UNIVERSITETET I BERGEN
Det medisinske fakultet

INNKALLING MØTE I PROGRAMUTVALG FOR FORSKERUTDANNING

Onsdag 17.04.24, kl. 14.00-15.30

Sted: Armauer Hansen Hus, møterom 437, 4.etg

Sak 05/24	**Godkjenning av innkalling og saksliste**
Sak 06/24	**Godkjenning av referat fra 07.02.2024**
Sak 07/24	**Fremdriftsrapportering 2023** Saksforelegg
Sak 08/24	**Ph.d.-utdanningsmelding 2023** Saksforelegg
Sak 09/24	**Årsmelding Forskerlinjen 2023** Saksforelegg
Sak 10/24	**Søknad om godkjenning av individuelt lesepensum** Saksforelegg
	Eventuelt

Martha Enger (s.) Havjin Jacob. (s.)

Postadresse
Postboks 7804
5020 Bergen

Telefon 55 58 00 00
Post@med.uib.no
www.uib.no/med
Org no. 874 789 542

Det medisinske fakultet
Telefon 55 58 20 86
Telefaks 55 58 96 82

Besøksadresse
AHH,
Haukelandsveien 28

REFERAT MØTE I PROGRAMUTVALG FOR FORSKERUTDANNING

Onsdag 07.02.24, kl. 14.00-15.30

Sted: Armauer Hansen Hus, møterom 437, 4.etg

Til stede: Martha C. Enger (leder), Harald Barsnes, Kristine Bærøe, Stian Knappskog, Åshild Johansen, Anne Berit Guttormsen, Stephanie Le Hellard, Kaia Nepstad og Havjin Jacob.

Sak 01/24	**Godkjenning av innkalling og saksliste** Godkjent.
Sak 02/24	**Godkjenning av referat fra 22.11.2023** En e-post ble sendt ut til utvalget som sirkulasjonssak. Denne sirkulasjonssaken skal inkluderes i referatet.
Sak 03/24	**Ph.d.-alumnusundersøkelsen 2023** Visedekan innledet med en presentasjon om bakgrunnen og oppsummeringen av ph.d.-alumnusundersøkelsen. Dette er en ledd i kvalitetssystemet for ph.d.-utdanningen, og det gjennomføres ph.d.-alumnusundersøkelser hvert tredje år. Den sykliske prosessen i kvalitetssikringssystemet er delt inn i fire faser: planlegge, utføre, evaluere (måle) og korrigere. Ph.d.-alumnusundersøkelsen ble for første gang gjennomført ved UiB våren 2023. Undersøkelsen ble sendt ut 890 personer som fullførte ph.d.-utdanningen ved UiB i perioden 2016-2019. Av disse har 335 personer svart på hele eller deler av undersøkelsen, hvorav 282 personer har svart hele og 53 personer har svart på deler av undersøkelsen.

	I undersøkelsen ble det samlet informasjon om hvordan ph.d.-utdanningen har gitt våre tidligere ph.d.-kandidater kompetanse og ferdigheter for den videre arbeidskarrieren deres. Alumnusundersøkelsen er delt inn i tre temaområder; arbeidslivstilknytning, forskningsopphold og oppnådd kompetanse, kunnskap og ferdigheter gjennom ph.d.-utdanningen. 87% av doktorer som har disputert ved MED er i relevant jobb innen ett år etter oppnådd grad. En overvekt av disse arbeider i dag ved et universitetssykehus, i tillegg til UH-sektoren. Doktorene melder at veiledning og undervisning er nødvendige og relevante kunnskaper og ferdigheter i deres nåværende stillinger. Det er verdt å merke seg at selv om omtrent halvparten av MEDs kandidater går videre til klinisk arbeid etter å ha fullført ph.d.-graden, er det likevel viktig å opprettholde disse ferdighetene innen undervisning og veiledning. Det kan være interessant tilnærming å inkludere undervisning som en del av ph.d.-løpet for klinikere. Utvalget diskuterte at besvarelsene i undersøkelsen er preget av to grupper: den ene gruppen som tar en ph.d.-grad og går direkte tilbake til klinikken, og den andre gruppen som er mer interessert i forskning. For å møte behovene til kandidatene som går tilbake til klinikken, bør fakultetet og sykehuset samarbeide tettere. Selv om sykehuset har opprettet en felles strategi og karriereløp, er det fortsatt ingen spesifikk innsats for å veilede klinikerne i hvordan de kan bruke den oppnådde ph.d.-graden i klinisk praksis. En interessant modell fra Oslo er å gi klinikerne en fridag fra klinikken for å drive forskning, slik at de kan balansere forskning og klinisk arbeid. Dette kan være en vei å utforske for å støtte klinikerne i å opprettholde sin forskningsinteresse samtidig som de er aktive i klinikken. UiB har et dobbelt ansvar når det gjelder ph.d.-kandidater: å utdanne dem for akademisk arbeid og samtidig oppfylle et samfunnsoppdrag. Realiteten

	er imidlertid at det ikke er nok akademiske stillinger til alle som fullfører en doktorgrad. Det er derfor forventet at mange havner andre steder enn i akademia. I besvarelsene fra MEDs doktorer kommer det frem at de ligger dårlig an når det gjelder utenlandske forskningsopphold. Utvalget diskuterte mulige årsaker, inkludert median alder blant kandidatene. Resultatene viser at MEDs doktorer er eldre sammenlignet med de andre fakulteter, og aler og familiesituasjon kan hindre dem i å reise utenlands. Det er interessant å merke seg at det ikke er noen direkte sammenheng mellom utenlandske forskningsopphold og kandidatenes tilfredshet. Derfor er det avgjørende å sikre kvalitet i utgangspunktet når man sender kandidatene på utenlandsopphold. For å forbedre utenlandske forskningsopphold, ble følgende tiltak foreslått av utvalget: 1. Formidle erfaringer: La ph.d.-kandidater dele sine erfaringer og nettverksmuligheter fra utenlandsopphold. 2. Studiepoeng: Kandidatene får allerede studiepoeng for utenlandsopphold (1,5 studiepoeng per uke). 3. Forlenget ph.d.-periode: Diskuter muligheten for å forlenge ph.d.-perioden når de er i utlandet, slik at de får bedre tid til å jobbe med prosjektet sitt. 4. Veiledning: utvikle veiledning om hvordan man søker om og tar kontakt med forskningsinstitusjoner i utlandet. 5. Oppdatere nettsider: gi informasjon om mulige finansieringskilder for utenlandsopphold på nettsidene. *Programutvalget tar ph.d.-alumnusundersøkelsen til etterretning.*
Sak 04/24	**Treårige emneevalueringer**

	Utvalget har gått gjennom to besvarelser fra treårige emneevalueringer for emner som ble undervist høsten 2023. **CCBIO901 (Junior Scientist Symposium)** Emneansvarlig skulle ha beskrevet og begrunnet pedagogiske valg, samt reflektert over kandidatenes læring som følge av disse valgene. Imidlertid har vedkommende kun kopiert og limt inn emnebeskrivelsen i evalueringsskjemaet. Kandidatevalueringen, som er en viktig del av emneevalueringene, er ikke inkludert, og det er heller ikke gitt noen begrunnelse for hvorfor den er utelatt. Antall deltakere på kurset har vært 23 siden høsten 2020, hvorav 5 kandidater har fullført kurset. Utvalget er klar over at dette kurset er en seminarserie, og deltakerne må ha oppfylt visse krav, inkludert deltakelse på minimum 4 symposium, skriving av totalt 4 vitenskapelige rapporter, og levering av en muntlig presentasjon i ett av seminarene. Emnebeskrivelsen indikerer at dette kan gjøres i løpet av to år, mens evalueringen nevner 23 påmeldte siden 2020. Utvalget ber emneansvarlige om å vurdere bærekraften av å opprettholde kurset for fem kandidater over en periode på 3,5 år. Selv om kurset også fungerer som en sosial sammenkomst for både ph.d.-kandidater, postdoktorer og andre, bør selve kursdelen vurderes på nytt, spesielt med tanke på det lave antallet som fullfører det. **CCBIO906 (Cancer Genomics):** Emneansvarlig har gjort en god jobbe med å skrive evalueringen. Kurset hadde totalt 41 deltakere høsten 2023, hvorav 15 kandidater hadde meldt seg på. Av disse har 9 bestått kurset. Aktivisering av kandidatene med gruppeoppgaver fungerer bra, og formatet på kurset slik det er nå, er tilfredsstillende. Likevel kan det være aktuelt å vurdere nye temaer eller metoder i undervisningen.

	Emnebeskrivelsen på nettsidene samsvarer ikke med det som emneansvarlige rapporterer. Nettsiden angir at kurset krevet 50-60 arbeidstimer, mens evalueringen gir en annen oppfatning. Dette bør rettes opp så snart som mulig. Diskusjonen om kurset skal være åpent for alle, ikke bare ph.d.-kandidater, er interessant. Fordelen med å inkludere andre deltakere, som postdoktorer, forskere og teknisk personale, er at de får muligheten til å lære mer om temaet. Ulempen er at ressursbruken kan bli feil, da forskerskolene primært mottar midler for å drive aktiviteter for ph.d.-kandidater. Utvalget er enige om at dette siste punktet krever en større diskusjon, og det kan tas opp ved evaluering av forskerskolene på et senere tidspunkt. *Programutvalget tar emneevalueringer til etterretning.*
	Eventuelt

Martha Enger (s.) Havjin Jacob. (s.)

DET MEDISINSKE FAKULTET

PROGRAMUTVALGET FOR FORSKERUTDANNING

MØTE 17.04.2024

SAK 07/24

Fremdriftsrapportering 2023

Hva saken gjelder

I tråd med ph.d.-forskriften § 8, skal alle aktive kandidater og deres hovedveiledere ved UiB hvert år levere separate, skriftlige framdriftsrapporter. Framdriftsrapporteringen er en del av den totale rapporteringen til UiB, og et ledd i institusjonens oppfølging av kandidatene.
Rapportene har siden 2010 blitt sendt ut av Forskningsadministrativ avdeling gjennom spørreskjemaverktøyet SurveyXact, men oppfølging av kandidatene skjer på instituttnivå.

Programutvalg for forskerutdanning skal på fakultetets vegne sikre at forskerutdanningen organiseres og gjennomføres med høy kvalitet. Som ledd i denne kvalitetssikringen leverer instituttene årlig en samlerapport for fremdriftsrapporteringen der det fremgår hvordan kandidater og veiledere har rapportert, og hvordan denne rapporteringen følges opp.

Programutvalget kan slik sikre seg at kandidatene får den oppfølgingen de har krav på. Samlerapportene skal ikke gi en detaljert fremstilling av de enkelte kandidaters situasjon, men synliggjøre hvordan instituttene jobber for å følge opp sine kandidater.

Programutvalget bes om å gå gjennom instituttenes samlerapporter og kommentere på forhold som bør følges opp videre, enten generelt eller på de enkelte instituttene.

Bestillingen til instituttene
Bestillingen til instituttene var noe endret for 2023 sammenlignet med foregående år. I tillegg til de vanlige spørsmålene om oppfølging av fremdriftsrapportene til kandidater og veiledere, ble instituttene også spurt om å beskrive sin oppfølging av pliktarbeid og medarbeider- og oppfølgingssamtaler. Dette ble flyttet fra bestillingen for ph.d.-utdanningsmelding til melding om oppfølging av fremdriftsrapportering for å inkludere all oppfølging av kandidatene i hele deres tilknytningsforhold til instituttet i samme melding. Instiutttenes rapporter omhandlet da oppfølging av fremdriftsrapport for alle i ph.d.-programmet, oppfølging av medarbeider- eller oppfølgingssamter for hhv. UiB-ansatte og eksternt ansatte kandidater og oppfølging av pliktarbeid for 4-årige UiB-ansatte stipendiater. De nye punktene er primært for fakultetets oppfølging av retningslinjene for samordning av pliktarbeid, som ble iverksatt i 2023.

Elementer fra instituttenes rapporter er også inkludert i fakultetet ph.d. utdanningsmelding, som behandles i sak 8/24.

Under følger et utdrag fra bestillingen som viser det nye i bestillingen (Bestillingen i sin helhet er vedlagt).

Oppfølging av 4-årige UiB-stipendiaters pliktarbeid

*Fra høsten 2023 iverksatte fakultetet retningslinjer for pliktarbeid for 4-årige UiB-stipendiater støttet av en anbefalt rutine (**Error! Reference source not found.**). I tillegg har Dekanatet bedt om en oversikt over gjennomført pliktarbeid de siste to årene.*

Oppfølging av pliktarbeid var et tema i dialogmøtene mellom fakultet og instituttene høsten 2023, og fakultetet har også fått lagt inn fakultetsspesifikke spørsmål om pliktarbeidet i spørreskjemaet for fremdriftsrapporten.

Oppdraget for beskrivelse av oppfølging av pliktarbeid er todelt.

Del 1: Oversikt over gjennomført pliktarbeid de siste to år

Fakultetet ber om en beskrivelse av:

a. *Hvor mange UiB-finansierte ph.d.-stipendiater har hatt pliktarbeid i 2022 og 2023?*
b. *Hvor mange timer pliktarbeid har hver av ph.d.-stipendiatene i pkt. a. gjennomført?*
c. *Hva er totalt antall timer gjennomført pliktarbeid for ph.d.-stipendiatene i perioden?*

Del 2: Praksis for oppfølging av pliktarbeid etter innføring av de nye retningslinjene for samordning av pliktarbeidet på fakultetet

Fakultetet ber om en beskrivelse av:

a. *Hvor mange UiB-stipendiater har pliktarbeid på instituttet høsten 2023?*
b. *Hvilke typer arbeidsoppgaver uføres?*
c. *Hvordan organiseres tildeling av kandidatenes pliktarbeid på instituttet?*
d. *Hvordan organiseres registrering av gjennomført pliktarbeid?*
e. *Hvordan følges pliktarbeid individuelt opp i løpet av forskerutdanningsløpet?*

Medarbeider- eller oppfølgingssamtaler

*I tråd med UiBs kvalitetssikringssystem for forskerutdanningen (**Error! Reference source not found.**) skal alle ph.d.-kandidater som er ansatt ved UiB ha en årlig medarbeidersamtale og alle ph.d.-kandidater som ikke har et arbeidsforhold ved UiB skal få et tilbud om en årlig oppfølgingssamtale (**Error! Reference source not found.**).*

Medarbeider- og oppfølgingssamtaler var et tema i dialogmøtene mellom fakultet og instituttene høsten 2023.

Fakultetet ber instituttene om en samlet vurdering av hvordan gjennomføring av samtalene har gått i 2023 med beskrivelse av eventuelle tiltak til forbedring i 2024. Hvor mange kandidater har fått tilbud og hvor mange har takket ja?

Instituttenes rapporter danner et viktig grunnlag for kvalitetsarbeidet forskerutdanningen ved MED og på UiB.

Forslag til vedtak

1. Programutvalg for forskerutdanning tar instituttenes samlerapporter for oppfølging av framdriftsrapportering, medarbeider- og oppfølgingssamtaler og pliktarbeid 2023 til etterretning.

Vedlegg

Samlerapporter fra instituttene, med unntak av IGS.

Bestilling til instituttene.

HEOM/10.04.2024

UNIVERSITETET I BERGEN
Det medisinske fakultet

Klinisk institutt 2
Institutt for global helse og samfunnsmedisin
Klinisk institutt 1
Institutt for biomedisin
Institutt for klinisk odontologi

Referanse	Dato
2023/16934-HEOM	04.12.2023

Fremdriftsrapportering i ph.d.-programmet og oppfølging av stipendiater og kandidater på instituttet: Bestilling til instituttene

Fremdriftsrapporter

Vi viser til utsendt fremdriftsrapportering i forskerutdanningen, med frist 1. november 2023 for alle kandidater og hovedveiledere ved UiB, og til tidligere korrespondanse i den forbindelse (2023/12788-HEOM, 01.09.2023).

Kandidat og hovedveileder skal hvert år levere separate og uavhengige rapporter om fremdriften av ph.d.-utdanningen (**Error! Reference source not found.**,**Error! Reference source not found.**). Fremdriftsrapporteringen er et ledd i den helhetlige kvalitetssikringen av forskerutdanningen og oppfølgingen av hver enkelt kandidat.

Alle instituttene har fått tilgang til rapportene, og melding om å starte oppfølging av kandidatene: https://skjema.app.uib.no/phdfremdriftsrapport2023.

Oppfølging av enkeltkandidater

Instituttene følger opp sine kandidater etter fremdriftsrapporteringen hvert år og Programutvalg for ph.d.-utdanning (PFU) går hvert år gjennom instituttenes samlerapporter som ledd i kvalitetssikringsarbeidet.

Under nevnes noen grupper som må følges særskilt opp:

- Kandidater som har svak progresjon, allerede er forsinket utover opprinnelig periode, eller rapporterer at de ikke vil bli ferdig innen rammene av sin avtaleperiode, må følges opp med tanke på å få avhandlingen levert innen rimelig tid. Det kan være aktuelt å be om særlig rapportering, kalle inn til møter, styrke/endre veiledergruppen, justere prosjektet eller ha samtaler med kandidat og veiledere.
- Kandidater som nærmer seg utløp av sin ph.d.-periode, må få melding om at de må søke om forlengelse av ph.d.-perioden dersom de fortsatt skal ha tilknytning til ph.d.-programmet. Slik søknad skal inneholde en realistisk plan for sluttføring og være attestert av veileder og instituttleder.
- Kandidater som har eller har hatt langtidsfravær bør følges opp særskilt for å sikre at prosjektet fortsatt er gjennomførbart, at kontakt mellom kandidat og

Det medisinske fakultet
Telefon 55582086
post.med@uib.no

Postadresse
Postboks 7804
5020 BERGEN

Besøksadresse
Realfagbygget, Allegt. 41
Bergen

Saksbehandler
Hege Ommedal
555833488

veiledere/institusjon opprettholdes, og for å hindre frafall.

- Kandidater som er svært forsinket eller som unnlater å følge opp sitt helhetlige rapporteringsansvar, kan som siste konsekvens skrives ut av forskerutdanningen (3). Anmodning om avslutning av ph.d.-studierett skal sendes til fakultetet. Grundig dokumentasjon på instituttets oppfølging av kandidaten må legges ved dersom kandidater skal skrives ut før avtalt tid.

Samlerapport til fakultetet

Det konkrete oppfølgingsansvaret er delegert til instituttene, men det er fakultetene som er ansvarlige for forskerutdanningen. Fakultetet ber derfor om en tilbakemelding fra instituttene i etterkant av fremdriftsrapporteringen.

Instituttene bes om å rapportere følgende:

- Kvantitative data (ved å generere samlerapporter direkte fra SurveyExact, både for kandidater og veiledere).
- Hvordan ph.d.-kandidatene følges opp av instituttet i etterkant av fremdriftsrapporteringen (se vedlagte mal).

Fakultetet skal ikke ha data på individnivå. Samlerapportene legges frem for PFU. Ikke all oppfølging må være sluttført før rapport sendes til fakultetet. Enkeltsaker må meldes til fakultetet separat gjennom ePhorte.

Oppfølging av 4-årige UiB-stipendiaters pliktarbeid

Fra høsten 2023 iverksatte fakultetet retningslinjer for pliktarbeid for 4-årige UiB-stipendiater støttet av en anbefalt rutine (4). I tillegg har Dekanatet bedt om en oversikt over gjennomført pliktarbeid de siste to årene.

Oppfølging av pliktarbeid var et tema i dialogmøtene mellom fakultet og instituttene høsten 2023, og fakultetet har også fått lagt inn fakultetsspesifikke spørsmål om pliktarbeidet i spørreskjemaet for fremdriftsrapporten.

Oppdraget for beskrivelse av oppfølging av pliktarbeid er todelt.

Del 1: Oversikt over gjennomført pliktarbeid de siste to år

Fakultetet ber om en beskrivelse av:

a. Hvor mange UiB-finansierte ph.d.-stipendiater har hatt pliktarbeid i 2022 og 2023?
b. Hvor mange timer pliktarbeid har hver av ph.d.-stipendiatene i pkt. a. gjennomført?
c. Hva er totalt antall timer gjennomført pliktarbeid for ph.d.-stipendiatene i perioden?

Del 2: Praksis for oppfølging av pliktarbeid etter innføring av de nye retningslinjene for samordning av pliktarbeidet på fakultetet

Fakultetet ber om en beskrivelse av:

a. Hvor mange UiB-stipendiater har pliktarbeid på instituttet høsten 2023?
b. Hvilke typer arbeidsoppgaver uføres?
c. Hvordan organiseres tildeling av kandidatenes pliktarbeid på instituttet?
d. Hvordan organiseres registrering av gjennomført pliktarbeid?
e. Hvordan følges pliktarbeid individuelt opp i løpet av forskerutdanningsløpet?

Medarbeider- eller oppfølgingssamtaler

I tråd med UiBs kvalitetssikringssystem for forskerutdanningen (5) skal alle ph.d.-kandidater som er ansatt ved UiB ha en årlig medarbeidersamtale og alle ph.d.-kandidater som ikke har et arbeidsforhold ved UiB skal få et tilbud om en årlig oppfølgingssamtale (6).

Medarbeider- og oppfølgingssamtaler var et tema i dialogmøtene mellom fakultet og instituttene høsten 2023.

Fakultetet ber instituttene om en samlet vurdering av hvordan gjennomføring av samtalene har gått i 2023 med beskrivelse av eventuelle tiltak til forbedring i 2024. Hvor mange kandidater har fått tilbud og hvor mange har takket ja?

Instituttenes rapporter danner et viktig grunnlag for kvalitetsarbeidet forskerutdanningen ved MED og på UiB.

Frist for instituttenes svar settes til **15. februar 2024.**

Ressurser – Nettsider og lenker:

1. Ph.d.-forskriften for Universitetet i Bergen, § 8: Rapportering
2. Programbeskrivelsen for Ph.d.-programmet ved Det medisinske fakultet, Pkt. 6.6: Fremdriftsrapportering
3. Ph.d.-forskriften for Universitetet i Bergen, § 5.5: Avslutning før avtalt tid
4. Vedtatte retningslinjer for 4-årige UiB-stipendiaters pliktarbeid og anbefalt rutine for oppfølging på instituttet. Se også ePhorte sak 2023/6423-1, dokument 3.
5. UiBs nettside for Kvalitetssikringssystemet for forskerutdanningen
6. Kvalitetssikringssystemet Kap. 4: Medarbeider- og oppfølgingssamtaler

Vennlig hilsen

Martha Enger
Visedekan for forskerutdanning

Tone Friis Hordvik
seksjonssjef

Vedlegg:
Skjema for samlerapport for fremdriftsrapportering i ph.d.-programmet 2023. Det medisinske fakultet.

Samlerapport og fremdriftsrapporteringen i ph.d.-programmet 2023

Instituttnavn: Institutt for biomedisin

1)

I hvor stor grad opplever instituttet at det er et problem at veiledere/kandidater ikke rapporterer?
- De fleste leverer, og vi opplever ikke at det er et problem.

Hva oppfatter instituttet at er årsakene til at enkelte ikke rapporterer?
- Det er som oftest en misforståelse/ledd i kommunikasjon som er årsaken.

Hva gjøres for å følge opp de som ikke rapporterer?
- Vi avklarer hva som er årsaken og følger opp kandidat eller veileder dersom det er hensiktsmessig.

2)

List opp hvilke typer tiltak som er iverksatt eller planlagt iverksatt på instituttet som del av oppfølgingen etter framdriftsrapportene for 2022, og anslå hvor mange kandidater det er aktuelt å følge opp med de enkelte tiltakene.

Type tiltak	**Antall kandidater**	**Merknad**
Møte med instituttledelsen (kandidat eller veileder)	2 stk.	Vi løste sakene på lavest mulig nivå
Møte med administrasjonen	4 stk.	Vi løste sakene må lavest mulig nivå
Kontakt med kandidat/veileder pga. store forskjeller mellom kandidat og veileders rapport	Ingen	
Endringer i pliktarbeid	Ingen	
Bytte av veileder/styrking av veiledergruppe	3 stk.	
Diskusjon med kandidat/veileder om justering av prosjekt for å bli ferdig innenfor avtaleperioden	1 stk.	
Melding om at kandidat må søke om forlengelse	9 stk.	Trenger mer tid på slutten før innlevering. Slutt på tilsetning og i ny jobb, det går sent å skrive
Melding om at kandidat må søke om permisjon eller	9 stk.	Eksterne kandidater som må melde til fakultetet om permisjoner/sykemeldinger.

annen endring av ph.d.-avtalen		Interne som må henvises til HR om å melde inn permisjoner/sykemeldinger.
Arbeid settes i gang med tanke på utskriving	1 stk.	Månedlig oppfølging, håper å levere uten utskriving.

Andre oppfølgingstiltak – fyll inn type tiltak og antall kandidater under:

Type tiltak	**Antall kandidater**	**Merknad**
Tilbakemeldinger om pliktarbeid	2 stk.	Ekstra samtale om best mulig tilrettelegging av pliktarbeid.

3)

Hvor mange kandidater oppgir at de er eller vil bli forsinket sammenlignet med opprinnelig ph.d.-periode?

- 9 stk. har søkt aktivt om forlengelse i ph.d.-programmet i 2023.
  9 stk. forlenget i tiden grunnet permisjon/sykemelding i 2023.

4)

Hvordan følger instituttet opp kandidater som er på overtid?

- Vi har som mål at alle kandidater skal stå med aktiv dato i ph.d. programmet.
- Tre måneder før opphør av dato i programmet minner vi om at de må søke om forlengelse om de ikke leverer innen aktiv dato.
- Vi skal ha søknaden om forlengelse inn og underskrevet ca. 1 måned før utløp i programmet.

Hvilke tiltak settes inn for å få disse til å levere en realistisk plan for gjennomføring, og hvordan følges slike individuelle planer opp fra instituttets side?

I søknaden om forlengelse ønsker vi at de legger ved et «gantt-chart» som viser realistisk arbeidsfremdrift. Vi følge så opp denne planen. Dette er særlig nyttig ved flere forlengelser, da kan vi lettere oppdage manglende fremdrift og følge opp kandidatene/veileder.

5)

Utfyllende (ikke personsensitive) opplysninger vedr. instituttets oppfølging. (Meld enkeltsaker inn i ePhorte).

-Ellers har vi åpen dør «policy» - kom inn og slå av en prat om en har spørsmål eller frustrasjoner.
-1-3 kommer innom hver uke.
- Vi får stort sett løst alle spørsmål ved raske avklaringer og kandidatene er mer fornøyd og føler seg ivaretatt.

Oppfølging av 4-årige UiB-stipendiaters pliktarbeid

1)

a) Hvor mange UiB-finansierte ph.d.-stipendiater har hatt pliktarbeid i 2022 og 2023?

2022: 23 stk. 2023: 25 stk.
Disse tallene gjelder for kandidater som er oppført med pliktarbeid, men pga kortere perioder hvor kandidatene er på utenlandsopphold, er i permisjon, eller er sykemeldte, så vil tallet enten vår eller høst være litt lavere (minus ca. 2-3).

b) Hvor mange timer pliktarbeid har hver av ph.d.-stipendiatene i pkt. a. gjennomført?

Generelt sett ser vi at undervisning og kurs dekker rundt 300 timer pliktarbeid pr år pr kandidat. Vi ser at den resterende tiden, rundt 200 timer i året, raskt blir dekket av pliktarbeid de utfører i forskningsgruppen de tilhører, da IBM har mange master studenter som skal veiledes og annet administrativt arbeid for forskningsgruppen.

c) Hva er totalt antall timer gjennomført pliktarbeid for ph.d.-stipendiatene i perioden?

Vi opererer ikke med nøyaktige tall, da det til nå har vært for tidkrevende å hente inn nøyaktige timetall for hver kandidat pr semester når undervisningen varierer vår vs. høst og kandidater kan være i permisjon/sykemeldt/på utenlandsopphold (se A).
Vi kan likevel regne ca. 20 stipendiater med pliktarbeid (se a)
x ca. 500 timer pliktarbeid pr år (se b) = ca. 10 000 timer pliktarbeid utført i 2023 hos IBM.
Vi har et godt samarbeid om undervisningen på IBM og kandidatene er pliktoppfyllende og kunnskapsrike. Kandidatene bruker Excel arket for «Oversikt over pliktarbeid» og vet at det kan komme kontroll av timeoppføring.

2)

a) Hvor mange UiB-stipendiater har pliktarbeid på instituttet høsten 2023?

Høst 2023: 25 stk er oppført med pliktarbeid (se også svar 1 a)

b) Hvilke typer arbeidsoppgaver uføres?

Pliktarbeidet på IBM er i hovedsak knyttet opp mot undervisning og kurs.
Kandidatene supplerer med pliktarbeid som forskningsgruppene har behov for som forskningsgruppe administrasjon, veiledning for master, osv.

c) Hvordan organiseres tildeling av kandidatenes pliktarbeid på instituttet?

I on-boarding prosessen informeres kandidatene om hva slags pliktarbeid vi har å tilby ved IBM, som er i hovedsak knyttet opp mot undervisning og kurs. Studieadministrasjonen har oversikt over kandidater som har pliktarbeid og fordeler kandidatene på timene som skal dekkes i undervisning og kurs. Ved on-boarding med ph.d.-koordinator og kontakt med studieadministrasjonen prøver vi å finne en god match mellom tildelt pliktarbeid og kandidatens kunnskapsbakgrunn.
Vi informerer også om hva slags pliktarbeid de kan supplere med, som varierer med hva de ulike forskningsgruppene har behov for (forskningsgruppe administrasjon, veiledning for master, osv.), slik at de oppnår kravet om 247,5 timer pr. semester.

d) Hvordan organiseres registrering av gjennomført pliktarbeid?

Kandidatene fører fortløpende opp pliktarbeidet, med faktor, i Excel arket «Oversikt over pliktarbeid» og senest ved fremdriftsrapportering hvert år, skal oversikten oppdateres.

e) Hvordan følges pliktarbeid individuelt opp i løpet av forskerutdanningsløpet?

Kandidatene blir oppfordret til å gi tilbakemelding om hvordan det går med utførelsen av pliktarbeidet, enten via fremdriftsrapporteringen eller ph.d.-koordinator. Dette er en del av vår «open-door» policy, hvor det er en lav tersker for å komme med tilbakemeldinger, slik at vi får gjort justeringer i pliktarbeidet om det trengs.
En frustrasjon som går igjen angående pliktarbeid på IBM er informasjon fra hoved-underviser (professor). Dette har vi hatt fokus på og vi ser at samarbeidet går litt bedre. Likevel har vi tilbakemeldinger fra høsten og nå i vår, at kandidatene ikke får nok informasjon (om f.eks. snitt i histologi), ikke blir introdusert, ikke har tilgang til MittUiB for emnet og de føler at undervisningen ikke blir så god som den kunne vært om de i større grad ble informert.

Medarbeider- eller oppfølgingssamtaler

Fakultetet ber instituttene om en samlet vurdering av hvordan gjennomføring av samtalene har gått i 2023 med beskrivelse av eventuelle tiltak til forbedring i 2024.

Medarbeidersamtalene for ph.d.-kandidatene utføres ute i forskningsgruppene. Vi har endel tilfeller av kandidater som (av ulike grunner) ikke har blitt tilbudt medarbeider samtale, eller at de takker nei til medarbeidersamtale.
Vi har gjort en omstrukturering av instituttet til større enheter av forskningsgrupper, og med en enhetsleder. Gjennomføring av medarbeidersamtaler vil bli tettere fulgt opp for 2024, og den nye enhetsstrukturen legger bedre til rette for ansvarsfordeling og oppfølging enn tidligere.

Hvor mange kandidater har fått tilbud og hvor mange har takket ja?

Medarbeidersamtale: 31 kandidater skal ha medarbeidersamtale ved IBM og 17 kandidater har gjennomført medarbeidersamtalen. Alle kandidatene skal ha fått tilbud om medarbeidersamtale, men vi utelukker ikke at det kan være noen avvik grunnet kommunikasjonssvikt.

Oppfølgingssamtale etter fremdriftsrapporteringen: her har to bedt om samtaler og vi er i kontinuerlig dialog med disse to kandidatene.

Samlerapport for fremdriftsrapporteringen i ph.d.-programmet 2023

Det medisinske fakultet

Instituttnavn: IKO

1) **I hvor stor grad opplever instituttet at det er et problem at veiledere/kandidater ikke rapporterer? Hva oppfatter instituttet at er årsakene til at enkelte ikke rapporterer? Hva gjøres for å følge opp de som ikke rapporterer?**

Institutt for klinisk odontologi opplever i liten grad at rapportering er en utfordring da alle kandidater og veiledere vanligvis leverer inn rapporter. To veiledere leverte ikke rapport i tide, men leverte til ph.d.-koordinator etter fristen. 1 kandidat leverte etter fristen til ph.d.-koordinator, 1 kandidat leverte ikke rapport.

2) **List opp <u>hvilke typer tiltak</u> som er iverksatt eller planlagt iverksatt på instituttet som del av oppfølgingen etter framdriftsrapportene for 2023, og anslå hvor mange kandidater det er aktuelt å følge opp med de enkelte tiltakene.**

Type tiltak	**Antall kandidater**	**Merknad**
Møte med instituttledelsen (kandidat eller veileder)	Alle, unntatt dem som har levert avhandling eller er i permisjon. Til sammen 23 kandidater.	Møte med forskningsleder, se merknadsfeltet under.
Møte med administrasjonen	23	IKO inviterer alle kandidatene, til medarbeider eller oppfølgingssamtaler i januar. Unntaket er dem som har levert/er i ferdmed å levere avhandling, eller er i permisjon. Forskningsleder, ph.d.-koordinator og kandidaten deltar på samtalene. Samtalene følger malene for ph.d.-medarbeider/oppfølgingssamtaler: System for quality assurance of the PhD education at UiB | Employee Pages | UiB Samtalene inkluderte bl.a. følgende: • Status i arbeidet, miljø, veilederforhold.

		• Info om å søke om godkjenning av opplæringsdelen • Praktisk info om sluttfasen for dem som nærmer seg levering. • Info om å søke om forlengelse for dem som ikke rekker å levere innen frist. • Fordelene med utenlandsopphold
Kontakt med kandidat/veileder pga. store forskjeller mellom kandidat og veileders rapport	0	Ikke aktuelt
Endringer i pliktarbeid	0	IKO har i løpet av desember/januar hatt egne Pliktarbeidsmøter med kandidat, hovedveileder, adm-sjef, forskningsleder og ph.d.-koordinator der vi har gjennomgått pliktarbeid utført så langt og plan for videre plikarbeid. 2 slike samtaler gjenstår pga. kandidater som er i permisjon. Pliktarbeid har derfor ikke vært tema på medarbeidersamtalene.
Bytte av veileder/styrking av veiledergruppe	9	Flere kandidater har endret på veiledergruppen i løpet av det siste året av diverse årsaker. En av årsakene er veiledere som har gått over til emeritus Dette har vært fulgt opp gjennom året og har også blitt tatt opp i medarbeidersamtalene.
Diskusjon med kandidat/veileder om justering av prosjekt for å bli ferdig innenfor avtaleperioden	1	Dette er resultat av midtveisevaluering gjennomført i januar, ikke av fremdriftsrapporten.
Melding om at kandidat må søke om forlengelse	0	På samtalene gir vi informasjon om å søke om forlengelse for dem som ikke rekker å levere innen frist.

Melding om at kandidat må søke om permisjon eller annen endring av ph.d.-avtalen	0	Vi har fortløpende kontakt med kandidatene om dette gjennom hele året, og det dukket ikke opp noe nytt i fremdriftssamtalene.
Arbeid settes i gang med tanke på utskriving	0	
Andre oppfølgingstiltak – fyll inn type tiltak og antall kandidater under:		

3) **Hvor mange kandidater oppgir at de er eller vil bli forsinket sammenlignet med opprinnelig ph.d.-periode?**

   6 kandidater svarer at planlagte innleveringstidspunkt ikke stemmer med opprinnelig plan for innleveringstidspunkt.

4) **Hvordan følger instituttet opp kandidater som er på overtid? Hvilke tiltak settes inn for å få disse til å levere en realistisk plan for gjennomføring, og hvordan følges slike individuelle planer opp fra instituttets side?**

   Vi mener å ha jevn kontakt med kandidatene som er på overtid. Vi har tradisjon for å kontakte kandidatene noen måneder før prosjektperioden deres går ut, og ber om dato for levering. Slik holder vi oversikt over kandidatene, og vi fanger opp forsinkelser. Kandidater som søker om forlengelse bes om å levere en statusrapport til ph.d.-koordinator hver måned.

5) **Utfyllende (ikke personsensitive) opplysninger vedr. instituttets oppfølging. (Meld enkeltsaker inn i ephorte)**

   Ingenting å tilføye.

Tilleggsspørsmål 2024:

Del 1: Oversikt over gjennomført pliktarbeid de siste to år

IKO har i løpet av høsten 2023 og januar 2024 gjennomført Pliktarbeidsmøter med kandidat, hovedveileder, administrasjonssjef, forskningsleder og ph.d.-koordinator hvor vi sammen har gått gjennom utført arbeid, sett på mulige nye oppgaver og laget en plan for tiden fremover.

Ettersom de nye rutinene for pliktarbeid ble innført i 2023, er det vanskelig å gi nøyaktige tall for utført pliktarbeid, men vi har gode rutiner for dette nå og under har vi listet foreløpige tall. Fakultetet ber om en beskrivelse av:

a. **Hvor mange UiB-finansierte ph.d.-stipendiater har hatt pliktarbeid i 2022 og 2023?**
   11
b. **Hvor mange timer pliktarbeid har hver av ph.d.-stipendiatene i pkt. a. gjennomført?**
   3 av kandidatene er i permisjon og vi har ikke tall for dem. Ca. tall for utført pliktarbeid for de resterende er:

Kandidat 1 393 timer
Kandidat 2 1448
Kandidat 3 1050
Kandidat 4 607
Kandidat 5 613
Kandidat 6 485
Kandidat 7 1650-1695
Kandidat 8 : 1300

c. **Hva er totalt antall timer gjennomført pliktarbeid for ph.d.-stipendiatene i perioden?**
Vi har ikke totalt antall ettersom vi mangler tilbakemelding på antall timer fra kandidatene som er i permisjon.

Del 2: Praksis for oppfølging av pliktarbeid etter innføring av de nye retningslinjene for samordning av pliktarbeidet på fakultetet

Fakultetet ber om en beskrivelse av:

a. **Hvor mange UiB-stipendiater har pliktarbeid på instituttet høsten 2023?**
11 – men to er i permisjon høsten 2023

b. **Hvilke typer arbeidsoppgaver uføres?**
Arbeidsoppgavene består både av oppgaver relatert til undervisning og administrative oppgaver.

c. **Hvordan organiseres tildeling av kandidatenes pliktarbeid på instituttet?**
Vi følger rutinen ang. pliktarbeid fra 2023 der pliktarbeid planlegges og tildeles på møte mellom kandidat, hovedveileder, admsjef, forskningsleder og ph.d.-koordinator.

d. **Hvordan organiseres registrering av gjennomført pliktarbeid?**
Kandidatene registrerer dette selv i skjemaet som ble innført ved nye retningslinjer for pliktarbeid i 2023.

e. **Hvordan følges pliktarbeid individuelt opp i løpet av forskerutdanningsløpet?**
Dette tas opp på samtaler med kandidater, både ved oppstart og i årlige medarbeidersamtaler.

Medarbeider- eller oppfølgingssamtaler

I tråd med UiBs kvalitetssikringssystem for forskerutdanningen (5) skal alle ph.d.-kandidater som er ansatt ved UiB ha en årlig medarbeidersamtale og alle ph.d.-kandidater som ikke har et arbeidsforhold ved UiB skal få et tilbud om en årlig oppfølgingssamtale (6).
Medarbeider- og oppfølgingssamtaler var et tema i dialogmøtene mellom fakultet og instituttene høsten 2023.

Fakultetet ber instituttene om en samlet vurdering av hvordan gjennomføring av samtalene har gått i 2023 med beskrivelse av eventuelle tiltak til forbedring i 2024. Hvor mange kandidater har fått tilbud og hvor mange har takket ja?

2023: IKO inviterer alle kandidatene, inkludert dem som startet etter første juli, til medarbeider eller oppfølgingssamtaler i januar. Unntak er kandidater som har levert avhandling/ er i ferd med å levere avhandling, nylig har hatt midtveisevaluering eller er i permisjon. Det forventes at alle kandidatene stiller på samtalene. Forskningsleder, ph.d.-koordinator og kandidaten deltar på samtalene. Samtalene inkluderte bl.a. følgende:

- Status i arbeidet, miljø, veilederforhold.
- Info om å søke om godkjenning av opplæringsdelen
- Praktisk info om sluttfasen for dem som nærmer seg levering.
- Info om å søke om forlengelse for dem som ikke rekker å levere innen frist.
- Fordelene med utenlandsopphold

I 2023, deltok 26 kandidater på samtalene.

2024: IKO inviterer alle kandidatene, med unntak av dem som er i permisjon eller har levert avhandling/er i ferd med å levere til medarbeidersamtale/oppfølgingssamtale i januar. Vi bruker malen under for samtalene og opplever at samtalene blir mer strukturert og dyptgående for alle kandidatene, men at vi bruker betraktelig mer tid på hver kandidat.

System for quality assurance of the PhD education at UiB | Employee Pages | UiB

Forskningsleder, ph.d.-koordinator og kandidaten deltar på samtalene.

Samlerapport for fremdriftsrapporteringen i ph.d.-programmet 2023

Det medisinske fakultet

Instituttnavn: Klinisk institutt 1

1) **I hvor stor grad opplever instituttet at det er et problem at veiledere/kandidater ikke rapporterer? Hva oppfatter instituttet at er årsakene til at enkelte ikke rapporterer? Hva gjøres for å følge opp de som ikke rapporterer?**

   Av i alt 117 kandidater er det 11 aktive kandidater og 10 veiledere som ikke har besvart. Nivået på rapportering ligger omtrent som foregående år. Alle med manglende rapporter har fått påminnelser, og tilsendte rapporter dokumenteres fortløpende i kandidatens mapper i ePhorte. Med unntak av enkelttilfeller, er det ikke de samme personene som unnlater å rapportere fra år til år, og den manglende rapporteringen framstår som noe tilfeldig.

2) **List opp <u>hvilke typer tiltak</u> som er iverksatt eller planlagt iverksatt på instituttet som del av oppfølgingen etter framdriftsrapportene for 2023, og anslå hvor mange kandidater det er aktuelt å følge opp med de enkelte tiltakene.**

Type tiltak	**Antall kandidater**	**Merknad**
Møte med instituttledelsen (kandidat eller veileder)	0	
Møte med administrasjonen	5	Oppfølgingsmøte gjennomført for alle som har bedt om det.
Kontakt med kandidat/veileder pga. store forskjeller mellom kandidat og veileders rapport	11	Forskjellene gjelder i hovedsak ulikt forventet innleveringstidspunkt/progresjon. For to kandidater tar vi kontakt på bakgrunn av innrapportert opplevelse av veiledningsforholdet.
Endringer i pliktarbeid	0	
Bytte av veileder/styrking av veiledergruppe	1	Dialog med 1 kandidat om ansvarsfordeling i veiledergruppen. Ellers ingen endringer direkte som følge av framdriftsrapporteringen.
Diskusjon med kandidat/veileder om justering av prosjekt for å bli ferdig innenfor avtaleperioden	0	
Melding om at kandidat må søke om forlengelse	7	Følges også opp kontinuerlig uavhengig av framdriftsrapportering.

Melding om at kandidat må søke om permisjon eller annen endring av ph.d.-avtalen	0	Følges opp kontinuerlig uavhengig av framdriftsrapportering.
Arbeid settes i gang med tanke på utskriving	0	Følges opp kontinuerlig uavhengig av framdriftsrapportering.
Andre oppfølgingstiltak – fyll inn type tiltak og antall kandidater under:		
Type tiltak	**Antall kandidater**	**Merknad**

3) Hvor mange kandidater oppgir at de er eller vil bli forsinket sammenlignet med opprinnelig ph.d.-periode?

54 kandidater har i skjemaet markert at innleveringstidspunkt ikke samsvarer med opprinnelig planlagt innleveringstidspunkt.

Dersom man sammenlikner oppgitt forventet innleveringspunkt med gjeldende sluttdato justert for permisjoner og forsinkelser, er det 23 kandidater hvor dette fortsatt ikke samsvarer, hvorav 12 kandidater har avvik på 4 måneder eller mindre. Kandidater i denne gruppen, og som er i siste fase av doktorgradsløpet, følges særlig opp.

4) Hvordan følger instituttet opp kandidater som er på overtid? Hvilke tiltak settes inn for å få disse til å levere en realistisk plan for gjennomføring, og hvordan følges slike individuelle planer opp fra instituttets side?

Instituttet følger med på kandidater som er nær eller over tiden, og igangsetter fortløpende dialog med kandidatene med tanke på eventuell forlengelse. Man er særlig oppmerksom på kandidater hvor opplæringsdelen ikke er fullført og hvor kandidaten allerede har fått maksimal forlengelse.

5) Utfyllende (ikke personsensitive) opplysninger vedr. instituttets oppfølging. (Meld enkeltsaker inn i ephorte).

Tilleggsspørsmål:

Del 1: Oversikt over gjennomført pliktarbeid de siste to år

a. Hvor mange UiB-finansierte ph.d.-stipendiater har hatt pliktarbeid i 2022 og 2023?

14 stipendiater.

Én av stipendiatene med pliktarbeid har vært i permisjon i aktuell periode.

b. Hvor mange timer pliktarbeid har hver av ph.d.-stipendiatene i pkt. a. gjennomført?

Fra perioden før innføring av nye retningslinjer er oversikten over gjennomført pliktarbeid hos kandidatene noe mangelfull, og egenrapportering i forbindelse med framdriftsrapportering er vår eneste kilde til opplysninger.

Noen av kandidatene som har rapportert i forgående år ser ikke ut til å ha tatt hensyn til forberedelsesfaktorer, og antall timer kan derfor ikke anses å være fullstendig.

Engelskspråklige kandidater rapporterer generelt om lavere antall timer, noe som kan skyldes utfordringer med å finne relevante undervisningsoppdrag.

Tre av kandidatene har i framdriftsrapporten oppgitt at de har hatt pliktarbeid, men har ikke spesifisert antall timer brukt på arbeidet.

c. Hva er totalt antall timer gjennomført pliktarbeid for ph.d.-stipendiatene i perioden?

4010 timer er dokumentert av kandidatene selv, men tallet er ikke fullstendig jf. kommentar i 1b).

Del 2: Praksis for oppfølging av pliktarbeid etter innføring av de nye retningslinjene for samordning av pliktarbeidet på fakultetet

a. Hvor mange UiB-stipendiater har pliktarbeid på instituttet høsten 2023?

13 stipendiater.

b. Hvilke typer arbeidsoppgaver uføres?

Undervisningsoppgaver inkl. assistentoppgaver, laboratorie- og tekniske oppgaver, veiledning av studenter, sensur/eksamensarbeid, andre formidlingsoppgaver, forskningsadministrasjon og annet administrativt arbeid, oppgaver knyttet til forskerskolene, seminarer og arrangementer, styreverv.

c. Hvordan organiseres tildeling av kandidatenes pliktarbeid på instituttet?

Tildeling av pliktarbeid skjer som regel av kandidatens veiledere eller forskningsgruppeledere. Instituttadministrasjonen bistår med å finne oppgaver etter behov, og legger vekt på planlegging av og informasjon om pliktarbeid og rapportering i onboardingsamtaler.

d. Hvordan organiseres registrering av gjennomført pliktarbeid?

Registrering av gjennomført pliktarbeid skjer ved egenrapportering gjennom excel-skjema tatt i bruk i forbindelse med nye retningslinjer. Rapportene sendes til ph.d.-koordinator ved instituttet som arkiverer disse,

e. Hvordan følges pliktarbeid individuelt opp i løpet av forskerutdanningsløpet

Pliktarbeid blir nå fulgt opp jf. nye retningslinjer, med innsamling av egenrapportert aktivitet, samt oppfølging av kandidater som trenger nye eller flere pliktarbeidsoppdrag.

Medarbeider- eller oppfølgingssamtaler

Oppfølging av medarbeidersamtaler for UiB-stipendiater er ved K1 delegert til de respektive seksjons- og forskningsgruppelederne, og alle kandidater skal få tilbud gjennom disse.

Instituttet har sett på muligheten for i tillegg å kunne gjennomføre oppfølgingssamtaler for de rundt 100 eksterne kandidatene, utover onboarding og oppfølgingssamtaler som følge av framdriftsrapporteringen. Det har i 2023 ikke vært realistisk å kunne organisere dette, på bakgrunn av instituttets ressurser og omfanget av samtalene.

Samlerapport for fremdriftsrapporteringen i ph.d.-programmet 2023

Det medisinske fakultet

Instituttnavn: Klinisk institutt 2

1) I hvor stor grad opplever instituttet at det er et problem at veiledere/kandidater ikke rapporterer? Hva oppfatter instituttet at er årsakene til at enkelte ikke rapporterer? Hva gjøres for å følge opp de som ikke rapporterer?

2) List opp hvilke typer tiltak som er iverksatt eller planlagt iverksatt på instituttet som del av oppfølgingen etter framdriftsrapportene for 2022, og anslå hvor mange kandidater det er aktuelt å følge opp med de enkelte tiltakene.

Type tiltak	**Antall kandidater**	**Merknad**
Møte med instituttledelsen (kandidat eller veileder)	4	
Møte med administrasjonen	0	
Kontakt med kandidat/veileder pga. store forskjeller mellom kandidat og veileders rapport	4	
Endringer i pliktarbeid	0	
Bytte av veileder/styrking av veiledergruppe	0	
Diskusjon med kandidat/veileder om justering av prosjekt for å bli ferdig innenfor avtaleperioden	0	
Melding om at kandidat må søke om forlengelse	20	

Melding om at kandidat må søke om permisjon eller annen endring av ph.d.-avtalen	2	
Arbeid settes i gang med tanke på utskriving	2	
Andre oppfølgingstiltak – fyll inn type tiltak og antall kandidater under:		
Type tiltak	**Antall kandidater**	**Merknad**

3) Hvor mange kandidater oppgir at de er eller vil bli forsinket sammenlignet med opprinnelig ph.d.-periode? 61

4) Hvordan følger instituttet opp kandidater som er på overtid? Hvilke tiltak settes inn for å få disse til å levere en realistisk plan for gjennomføring, og hvordan følges slike individuelle planer opp fra instituttets side?

Phd koordinator har løpende oppfølging av kandidater som har phd periode som utløper. De fleste blir bedt om å søke forlengelse.

5) Utfyllende (ikke personsensitive) opplysninger vedr. instituttets oppfølging. (Meld enkeltsaker inn i ephorte).

Framdriftsrapportering 2023

Klinisk institutt 2

Oppfølging av 4-årige stipendiater med pliktarbeid

Del 1: Oversikt over gjennomført pliktarbeid siste to år

a. Hvor mange UiB-finansierte phd stipendiater har hatt pliktarbeid i 2022 og 2023
   Vi har ikke tall for 2023, så har derfor tatt 2022 og 2021
   Til sammen 66 kandidater har hatt pliktarbeid i perioden.

b. Hvor mange timer pliktarbeid har hver av phd stipendiatene gjennomført?
   Vi har ingen gode tall på dette. Tallene er basert på hva som var rapportert i framdriftsrapporteringen. Kandidatene har meldt tilbake et snitt på 200 timer. Da innbefattet kandidater som har vært i permisjon osv.

c. Hva er totalt antall timer gjennomført pliktarbeid for phd stipendiatene i perioden? 13000

Del 2: Praksis for oppfølging av pliktarbeid etter innføring av nye retningslinjer for samordning av pliktarbeid på fakultetet

Instituttet jobber fremdeles med implementering av rutinene for pliktarbeid, og finne en god måte å følge dette opp på.

K2 hadde 28 stipendiater med pliktarbeid høsten 2023. Av disse var kun 10 norsktalende og kun 2 av disse er leger. Dette er utfordrende med tanke på undervisning instituttet er ansvarlig for (farmasi, medisin, odontologi) som foregår på norsk og krever bakgrunnskunnskap.

a. K2 hadde 28 stipendiater med pliktarbeid høsten 2023. Dette de med 100% stilling i 4 år. Av disse var kun 10 norsktalende. Kun 2 av disse er leger og kan brukes i undervisning.
   Dette er en utfordring for oss da store deler av behovet for pliktarbeid ligger innen undervisning. Vi tenker det kan være fornuftig å se på ordningen med pliktarbeid slik at vi får utnyttet denne ressursen bedre.

b. Arbeidsoppgaver som utføres er hovedsakelig undervisning, K2 nytt redaktør, arbeid i Forskerskolen for klinisk medisin, komitearbeid, arrangement av Junior/phd retreat.

c. Tildeling av pliktarbeid diskuteres ved oppstarts samtaler. Det blir også sendt ut forespørsel til kandidatene ved behov (f.eks for undervisning)

d. Det vil for vårsemesteret bli sendt ut registreringsskjema til alle kandidatene hvor pliktarbeid skal gjøres rede for.

e. Vi har for øyeblikket ikke rutiner for oppfølging av pliktarbeid underveis i phd løpet utover framdriftsrapporten.

Medarbeider – eller oppfølgingssamtaler

Medarbeidersamtaler med phd-stipendiatene er delegert forskningsgruppelederne ved instituttet. De har ansvar for gjennomføring gjennom sitt mandat. I vår årlige HMS rapportering er det meldt at 44 stipendiater skal ha medarbeidersamtaler. 16 har gjennomført. Det kan være feilmarginer her da ikke alle forskningsgruppene har meldt tilbake sine tall. Vi vil prøve å lage bedre rutiner for oppfølging av dette i fremtiden.

Gjennomføring av oppfølgingssamtaler har instituttet dessverre ikke hatt kapasitet til å ha fokus på. Vi vil se nærmere på dette for 2024.

DET MEDISINSKE FAKULTET

PROGRAMUTVALGET FOR FORSKERUTDANNING

MØTE 17.04.2024

SAK 08/24

Ph.d.-utdanningsmelding 2023

Hva saken gjelder

Vedlagt er årsmelding for ph.d.-utdanningsmelding. Fakultetet er blitt bedt om å rapportere for 2023 på følgende punkter:

- Oppfølging av foregående års ph.d.-utdanningsmelding og eventuelle tiltak som har blitt gjennomført.

- Nøkkeltall, herunder opptak, gjennomføring («seksårsmålet»), disputaser, underkjenninger, mindre omarbeiding og frafall. Beskrivelse av situasjonen og eventuelle tiltak for å bedre noen av disse parameterne. Kommenter særlig dersom det forekommer større endringer i noen av disse parameterne eller om tallene ikke er tilfredsstillende.

- Kandidatenes bidrag til forskningen ved fakultetene. Forhold som kan omtales:
    - graden av nyskapende forskning
    - hvordan kandidatene utfyller allerede eksisterende forskningsfelt
    - strategiske vurderinger som gjøres ved ansettelse og opptak av kandidater
    - strategiske vurderinger rundt BOA-aktivitet og hvordan fakultetet bruker eksternfinansierte prosjekter strategisk for å finansiere nye ph.d.-kandidater

- Fakultetenes systematiske kvalitetsarbeid. Hovedvekt bør legges på elementer der det har vært særlig utvikling eller aktivitet. Forhold som kan omtales:
    - oppfølging av fremdriftsrapporter
    - midtveisevaluering
    - gjennomføring av medarbeider- eller oppfølgingssamtaler
    - kvalitetssikring av opplæringsdelen, herunder emneevaluering og evaluering av emneporteføljen og rammene for opplæringsdelen
    - gjennomføring og oppfølging av programevaluering
    - oppfølging av undersøkelser

Andre forhold fakultetene ønsker å omtale, for eksempel veilederopplæring

Bestillingen til instituttene og fakultetet er vedlagt. Årets frist for ph.d.-meldingen til FIA er **3. mai 2024**.

Alle institutt har levert sine rapporter (vedlagt). På grunnlag av instituttenes rapporter, formulerer fakultetet en samlet ph.d.-utdanningsmelding. Vi gjør oppmerksom på at dette er et førsteutkast. Programutvalget bes om å gå gjennom instituttenes og fakultetets ph.d.-utdanningsmelding og komme med eventuelle kommentarer og endringsforslag.
Ph.d. utdanningsmelding skal også behandles i Fakultetsstyret 24. april.

Forslag til vedtak

Programutvalget for forskerutdanning tar ph.d.-utdanningsmeldingen til etterretning, eventuelt med de kommentarer/endringsforslag som fremkommer i møtet.

Vedlegg

1. Bestilling til fakultetene
2. Bestilling til instituttene
3. Instituttenes ph.d.-utdanningsmeldinger 2023
4. Fakultetets ph.d.-utdanningsmelding 2023

HEOM/10.04.2023

UNIVERSITETET I BERGEN
Forsknings- og innovasjonsavdelingen

Fakultetene

Referanse	Dato
2023/16725-MARIEI	24.11.2023

Ph.d.-utdanningsmeldingen 2023

Som en del av arbeidet med universitetets ph.d.-utdanningsmelding, ber vi om at det utarbeides meldinger for de syv fakultetsvise ph.d.-programmene. For forankring av ph.d.-programmenes meldinger, bør disse behandles av de respektive programstyrene.

Ph.d.-utdanningsmeldingene bør ikke være lengre enn fire sider. Vi ber om at punktene nedenfor omtales:

- Oppfølging av foregående års ph.d.-utdanningsmelding og eventuelle tiltak som har blitt gjennomført.

- Nøkkeltall, herunder opptak, gjennomføring («seksårsmålet»), disputaser, underkjenninger, mindre omarbeiding og frafall. Beskrivelse av situasjonen og eventuelle tiltak for å bedre noen av disse parameterne. Kommenter særlig dersom det forekommer større endringer i noen av disse parameterne eller om tallene ikke er tilfredsstillende.

- Kandidatenes bidrag til forskningen ved fakultetene. Forhold som kan omtales:
    - graden av nyskapende forskning
    - hvordan kandidatene utfyller allerede eksisterende forskningsfelt
    - strategiske vurderinger som gjøres ved ansettelse og opptak av kandidater
    - strategiske vurderinger rundt BOA-aktivitet og hvordan fakultetet bruker eksternfinansierte prosjekter strategisk for å finansiere nye ph.d.-kandidater

- Fakultetenes systematiske kvalitetsarbeid. Hovedvekt bør legges på elementer der det har vært særlig utvikling eller aktivitet. Forhold som kan omtales:
    - oppfølging av fremdriftsrapporter
    - midtveisevaluering
    - gjennomføring av medarbeider- eller oppfølgingssamtaler
    - kvalitetssikring av opplæringsdelen, herunder emneevaluering og evaluering av emneporteføljen og rammene for opplæringsdelen
    - gjennomføring og oppfølging av programevaluering
    - oppfølging av undersøkelser

- Andre forhold fakultetene ønsker å omtale, for eksempel veilederopplæring.

Ph.d.-utdanningsmeldingene danner et viktig grunnlag for kvalitetsarbeidet ved ph.d.-utdanningen ved UiB.

Forsknings- og innovasjonsavdelingen
Telefon +4755584980
post@uib.no

Postadresse
P.B. 7800
5020 BERGEN

Besøksadresse
Nygårdsgaten 5
Bergen

Saksbehandler
Marie Eide
55584963

Frist for fakultetenes ph.d.-utdanningsmeldinger settes til **3. mai 2024**. Ph.d.-utdanningsmeldingen vil bli behandlet i UiB sitt forskningsutvalg før endelig behandling i universitetsstyret.

Vennlig hilsen

Alette Gilhus Mykkeltvedt
avdelingsdirektør

Marie Eide
seniorrådgiver

Kopi
Universitetsdirektørens kontor

UNIVERSITETET I BERGEN
Det medisinske fakultet

Klinisk institutt 1
Institutt for global helse og samfunnsmedisin
Klinisk institutt 2
Institutt for biomedisin
Institutt for klinisk odontologi

Referanse	Dato
2023/16934-HEOM	04.12.2023

Ph.d. utdanningsmelding for 2023 (bestilling)

Vi viser til bestilling fra FIA i sak 2023/16725 der Det medisinske fakultet er bedt om å utarbeide ph.d.-utdanningsmelding for fakultetets ph.d.-program for 2023 (vedlagt). Informasjon fra instituttenes utdanningsmeldinger utgjør mye av grunnlagsmaterialet for fakultetets melding. Bestillingen til instituttene for ph.d.-utdanningsmelding for 2023 omfatter kvalitet i forskning, midtveisevaluering, opplæringsdelen og forskerskoler (nytt i 2023).

Utdanningsmeldingen er et strategisk viktig dokument som viser bredden og kvaliteten i forskerutdanningen ved MED. Meldingen danner et viktig grunnlag for kvalitetsarbeidet ved ph.d.-utdanningen ved UiB.

Fakultetet ber instituttene om innspill til deler av meldingen. Fakultetet vil selv hente ut kvantitative data og andre mer overordnede punkter som skal omtales i meldingen. Instituttene bes om å omtale følgende punkter:

- Oppfølging av ph.d.-utdanningsmelding for 2022 og eventuelle tiltak.
- Kandidatens bidrag til forskningen ved instituttet. Forhold som kan omtales:
    - Strategiske vurderinger som gjøres ved ansettelse og opptak av kandidater.
    - Hvordan kandidatene utfyller allerede eksisterende forskningsfelt.
    - Graden av nyskapende forskning og innovasjon.
- Midtveisevaluering.
- Opplæringsdelen:
    - Hvordan man vurderer instituttets kursportefølje.
    - Hvordan ph.d.-emner evalueres.
    - Instituttets registrering av emner til NorDoc-databasen.
    - Deltagelse av studenter fra andre institusjoner til egne NorDoc-tilgjengelige kurs.
- Forskerskoler:
    - Antall nye kandidater som har deltatt på instituttets forskerskole(r) i 2023.
    - En kort oppsummering av erfaringer med FS-registrering av deltagelse i forskerskolene.

Instituttene står fritt til å benytte anledningen til å kommentere andre forhold ved ph.d.-utdanningen dersom det er ønske om det.

Det medisinske fakultet
Telefon 55582086
post.med@uib.no

Postadresse
Postboks 7804
5020 BERGEN

Besøksadresse
Haukelandsveien 28
5009 BERGEN

Saksbehandler
Hege Ommedal
555833488

Frist for instituttenes svar settes til **15. februar 2024.** Innspillene vil bli lagt fram for Programutvalg for forskerutdanning i møte 20. mars 2024 og innarbeidet i fakultetets melding, som blir sendt til Forsknings- og innovasjonsmeldingen.

Vennlig hilsen

Martha Enger
Visedekan for forskerutdanning

Tone Friis Hordvik
seksjonssjef

Vedlegg:
Bestillingsbrev fra FIA
Oversikt over forskerskoler

Klinisk institutt 1 (K1) – Ph.d.-utdanningsmelding 2023

Oppfølging av ph.d.-utdanningsmeldingen 2022 og eventuelle tiltak

Det medisinske fakultet vedtok nye retningslinjer for gjennomføring av pliktarbeid for stipendiater, med virkning fra høsten 2023. Instituttet har startet arbeidet med å implementere nye rutiner, med særlig vekt på planlegging og tilrettelegging av pliktarbeid for nye kandidater. Alle kandidater med pliktarbeid, med unntak av kandidater i sluttfasen, er nå bedt om å rapportere oppgaver og omfang etter endt semester, og fullstendige data skal foreligge etter våren 2024.

Instituttet har i 2023 fortsatt gjennomføring av onboardingsamtaler for alle nye kandidater, og pliktarbeid er nå også et fast punkt i samtalene med kandidater ansatt i UiB-stilling.

I 2023 ble det gjennomført 17 disputaser ved K1. To av disputasene ble gjennomført heldigitalt. Alle øvrige disputaser ble gjennomført i Bergen.

Kandidatenes bidrag til forskningen ved instituttet

K1 inkluderer 18 fagområder hvor de aller fleste har aktiv forskning med ph.d.-rekruttering. Strategiske valg gjøres i forskningsgruppene, med rekruttering av kandidater til pågående prosjekter, og med søknader om ph.d.-stipendfinansiering til NFR, MED-UiB, Helse Vest, Kreftforeningen, DAM-stiftelsen og andre kilder for ekstern finansiering. I tillegg sendes det tilsvarende søknader ved utlysning av særlige satsningsområder som er tilpasset fagområdene ved instituttet.

Rekruttering av kandidater som kan søke ph.d.-utlysninger gjøres gjennom langsiktig satsning på å rekruttere forskerlinjestudenter, andre studenter som skriver særoppgaver, samt LIS-leger ved de kliniske avdelingene ved samarbeidende sykehus (HUS, SUS, FONNA, FØRDE, HDS, SØRLANDET etc.). Innenfor fagområdet ernæring rekrutteres ph.d.-kandidater særlig gjennom masterprogrammene.

Rekruttering av etablerte medisinere i ph.d.-prosjekter representerer tidvis utfordringer når det gjelder fordeling av tid til forskning og arbeid i klinikk.

Ved K1 er innovasjon satsingsområde i nært samarbeid med den nye EITRI-inkubatoren. Nye ph.d.-kandidater har mulighet til å få lab/kontorplass i EITRI dersom prosjektet deres ser ut til å lede mot innovasjoner.

Midtveisevaluering

Instituttet gjennomførte 25 midtveisevalueringer i 2023.

Midtveisevalueringene legges i hovedsak til mai og november, men instituttet legger også til rette for gjennomføring ved andre tidspunkter etter avtale med enkeltkandidater.

Instituttet jobber særlig med å fange opp kandidater som utsetter midtveisevalueringen, med mål om gjennomføring innen 2/3 av ph.d.-perioden jf. retningslinjene.

Forhold som krever oppfølging, og som avdekkes gjennom evalueringen, følges opp av ph.d.-koordinator ved instituttet, og eventuelt med instituttledelse.

Opplæringsdelen

Emneporteføljen på ph.d.-nivå ved K1 er i hovedsak knyttet opp mot forskerskolene instituttet har ansvar for. Disse tilbyr også kurs med generell interesse utover fagområdet forskerskolene representerer. Instituttet administrerer 19 enkeltemner, hvorav 15 er del av forskerskolene CCBIO og Neuro-SysMed. Sammen utgjør disse en viktig arena for ph.d.-kandidater, både ved K1 og andre institutter.

Et nytt ph.d.-emne i oral patologi ble opprettet i 2023, etter initiativ fra fagmiljøet. Emnet er ikke tilknyttet noen av forskerskolene, og ble gjennomført heldigitalt for kandidater ved samarbeidende institusjoner nasjonalt og internasjonalt.

Enkeltemner ved forskerskolene evalueres ved spørreskjema til deltakerne etter gjennomføring. Forskerskolen CCBIO har også i 2023 gjennomført egenevaluering av emnene CCBIO907 og CCBIO908. I tillegg er det gjort treårige emneevalueringer av CCBIO901 og CCBIO906. Evalueringene har vært behandlet i Programutvalg for forskerutdanning ved Det medisinske fakultet.

Med unntak av seminarseriene, blir de fleste ph.d.-emner tilhørende forskerskolene CCBIO og Neuro-SysMed tilbudt gjennom NorDoc-nettverket, med utlysning på nettverkets egne nettsider. Instituttet har i flere år tatt i bruk Søknadsweb med integrasjon mot FS for behandling av søknader om gjestestudierett for eksterne deltakere. Alle kurs har oppslutning både nasjonalt og internasjonalt, med deltakelse av relevante kandidater.

Forskerskoler

Følgende antall ph.d.-kandidater ble formelt registrert ved instituttets forskerskoler i 2023:

Forskerskolen CCBIO: 41 kandidater
Forskerskolen Neuro-SysMed: 2 kandidater
Forskerskolen i klinisk medisin: 20 kandidater

Registering av medlemmer i forskerskolene er gjort etter gjeldende kriterier. Det er fortsatt mange ubesvarte spørsmål knyttet til registrering og framgangsmåte, og enkelte sider ved registreringen oppleves som noe vilkårlig. Det er viktig å understreke at formelt registrerte medlemmer i forskerskolene ikke gir et fullstendig bilde av forskerskolenes aktivitet som helhet, da mange kurs og delaktiviteter kan ha ulik oppslutning fra år til år, og fordi seminarseriene også tenderer mot en noe mer uformell form, med deltakere og besøkende som ikke alltid registreres. Enkelte kurs ved forskerskolen Neuro-SysMed har stor oppslutning av både interne og eksterne deltakere, men er samtidig ikke kriterium for registrering av opptak til forskerskolen.

PhD UTDANNINGSMELDING 2023 FOR KLINISK INSTITUTT 2 (K2)

Oppfølging av ph.d. -utdanningsmeldingen 2022 og eventuelle tiltak

K2 har for tiden 149 phd.kandidater. Koronapandemien har fortsatt innvirkning på fremdriften til mange av kandidatene.
PhD koordinatorene jobber kontinuerlig med oppfølging og veiledning av kandidatene og er alltid tilgjengelige.
Vi bruker midtveisevalueringen som en viktig pekepinn/virkemiddel for å fange opp kandidater som trenger ekstra oppfølging.

Kandidatenes bidrag til forskningen ved instituttet
Strategiske vurderinger som gjøres ved ansettelse og opptak av kandidatene: Instituttet har ikke noen stillinger som kan brukes til dette formålet. Instituttet har ikke penger til å drive strategiske grep, fakultetet bør vurdere å avsette en pott til det.

Hvordan kandidatene utfyller allerede eksisterende forskningsfelt/grad av nyskapende forskning: De fleste stipendiatstillingene er ekstern finansiert, dvs PIene er ansvarlig for å velge ut kandidatene.

Når vi får en søknad om opptak til phd programmet hvor finansieringen er på plass aksepteres det etter vurdering av gjennomførbarhet, instituttet er ikke involvert i utvelgelse av kandidater eller prosjekter.

Våre PhD kandidater bidrar i stor grad til nyskapende forskning ved instituttet. Vi følger forskriften for graden ph.d ved UiB når det gjelder krav til avhandlingen, samt programbeskrivelsen og veiledning ved det medisinske fakultetet. PhD kandidatene bidrar i stor grad til antall publikasjonene som utgår fra K2.

Vi har hatt mindre fokus på innovasjon med tanke på patentering, og vi har tidligere opplevd en del problemer med næringslivs PhD når det gjelder resultater som ikke passer oppdragsgiveren og hvor kandidaten har hatt problemer med å publisere data.

Oppfølging av fremdriftsrapporter
Fremdriftsrapportene for 2023 viser at stipendiatene er stort sett fornøyde med sine veiledere. De tilfellene hvor en kandidat eller veileder har uttrykt misnøye med veilederforholdet følges opp basert på informasjonen i fremdriftsrapportene. Det store antallet fornøyde kandidater tyder på at veilederne ved instituttet er klar over det ansvaret det å være veileder medfører, og at de gjør jobben slik en kan forvente av dem. En del av våre stipendiater har veiledere i professor II stilling. Stort sett fungerer det bra. Alle veiledere oppfordres til å delta på veilederkursene organisert av Det medisinske fakultet, og informasjon om relevante andre veilederkurs og seminarer promoteres jevnlig gjennom vårt ukentlig nyhetsbrev.

Midtveisevaluering
Det gjennomføres midtveisevalueringer en gang per semester med påfølgende oppfølging hvor bekymringer/forsinkelser følges opp av PhD koordinator Irene Hjelmaas og evtl. leder for forskning, Silke Appel, hvis det trengs.

Medarbeidersamtaler
Ansvar for medarbeidersamtaler er delegert til forskningsgruppeledere. Instituttet får rapportert gjennomføring en gang per år. Medarbeidersamtalen skal ikke gjennomføres av veilederen.

Opplæringsdelen
Instituttet tilbyr forskerkurs hovedsakelig i regien av forskerskolene. Noen av kursene gir veldig få poeng, det er en fordel å ha noen kurs som gir litt større poengsum som for eksempel MEDMET900.

Evaluering av ph.d. emner: Dette er noe programmutvalget bør følge opp med emneansvarlige og forskerskolelederne direkte.

Disputaser
Hybride løsninger/strømming er forbeholdt de disputaser hvor opponentene ikke deltar fysisk. Vi kan ikke tilrettelegge for disputaser gjennomført andre steder enn Bergen og Stavanger som har nok UiB ansatte til å kunne organisere det lokalt. Hvis fakultetet ønsker å åpne for disputaser andre steder kan ikke vi være ansvarlig for gjennomføring, oppfølging og ikke minst finansiering.

Registrering av forskerskoler i FS
Vi har liten erfaring med registrering i FS. Det fremstår noe uklart hvem som skal registreres utover dem som har deltatt på forskerskolenes kurs. Vi ønsker dialog med fakutletet på hvordan dette bør løses.

Ph.d.-utdanningsmelding 2023.

Institutt for biomedisin, med frist 15. februar 2024

1. Oppfølging av foregående års ph.d.-utdanningsmelding og eventuelle tiltak

Opptak, gjennomstrømming, disputaser, underkjenninger og frafall – beskrivelse av situasjonen og eventuelle tiltak for å forbedre disse parameterne.

Hvis gjennomstrømmingen har vært svakere enn ønskelig, ber vi om at bakgrunnen for dette kommenteres 20

Opptak:
Ved Institutt for biomedisin har vi kontinuerlig opptak av ph.d. kandidater gjennom året. Studentene har alltid 1:1 onboarding-samtaler med ph.d.-koordinator ved oppstart, da det kan være spørsmål relatert til søknadskjema for opptak. Det er også den del spørsmål om valg av emner.
Vi hadde et greit opptak i 2023 med 9 nye kandidater.

Forlengelser:
Vi har hatt søknader om forlengelser både fra både 3-årige og 4-årge kandidater. 9 kandidater trenger mer tid i skriveperioden. Kullene som søker om forlengelse er fra 2018 (4), 2019 (1), 2020 (3) og 2021 (1). Så det virker som det er en «hale» av forlengelser fra pre- og post-korona.
Andre naturlige forsinkelser skyldes generelt foreldrepermisjoner og sykdom (her også 9 kandidater), fordelt på kull fra 2018-2021.

Gjennomstrømming og disputaser:
Antallet disputaser har vært noe lavere i 2023, 5 stk. mot 10 stk. i 2022, og kan ses i sammenheng med «halen» av forlengelser vi har sett for 2023, grunnet lengre tid til skriving. De fleste forlengede i denne «halen» vil levere i 2024, og vi regner med å ha alle igjennom og disputert ved utgangen av 2024. I 2023 var ingen disputaser digitale/hybride.

Frafall:
Vi hadde ingen underkjennelser i 2023. Vi har hatt to kandidater som har sluttet i sin ph.d.-stilling av personlige årsaker. Disse ble fulgt nøye opp for å se om det var enda noe mer instituttet kunne ha gjort for at de skulle ha fortsatt i stipendiatstillingen.

2. Kandidatens bidrag til forskningen ved fakultetet. Forhold som kan omtales

a. Strategiske vurderinger som gjøres ved ansettelse og opptak av kandidater:

Ved Institutt for biomedisin gjør de ulike forskningsgruppene strategiske vurderinger ved utlysning av ph.d.-stillinger, slik at prosjektene og arbeidsoppgavene passer inn i instituttets forskningsportefølje og behov. Vi følger standard oppsett for søknad- og ansettelses-prosess, med faglig vurdering av prosjekt (for eksterne søknader) og komite og godkjenning fra instituttets ledelse.

b. Hvordan kandidatene utfyller allerede eksisterende forskningsfelt

Kandidatene meddeler at de føler seg inkludert i det forskningsfeltet de tilhører, mye grunnet forskningsleder sitt engasjement, samt nasjonale og internasjonale kontakter. Instituttet tilstreber å ha en god arbeidsstedstilhørighet og inkludering i det eksisterende forskningsfeltet. Vi håper at med vår nye struktur på IBM, med større enheter, at ph.d.-kandidater i mindre forskningsgrupper skal få en større faglig omgangskrets. Vi vil oppmuntre enhetene til å arrangere flere felles møte gjennom året, hvor kandidatene kan presentere sin fremgang i prosjektet.

For å ytterligere styrke samarbeid for hele forskningsfeltet skulle vi ønske oss flere sosiale sammenkomster, gjerne på tvers av karrierestegene og instituttene. Vi ser frem til «felleslunsj» som er planlagt i Eitri denne våren, det blir flott.

Vi får fremdeles mange positive tilbakemeldinger på at forskerskolen bidrar til sosial tilhørighet for ph.d.-studentene. Kjennskap til hverandre i regi av forskerskolen gjør at studentene deler faglige erfaringer og lærer av hverandre, som igjen skaper grobunn for forskningssamarbeid senere i karrieren.

c. Graden av nyskapende forskning

De ulike forskningsgruppene ved instituttet dekker viktige områder innen biomedisinsk forskning, medisinsk forskning og medisinsk grunnforskning. Kandidatene ved Institutt for biomedisin bidrar i stor grad til nyskapende forskning og er, sammen med deres forskningsgruppe, en del av fronten i deres respektive forskningsfelt.

3. Midtveisevaluering

Kandidatene ved Institutt for biomedisin meddeler at de setter pris på midveisevalueringen og at denne er nyttig i deres ph.d.-forløp.
Administrasjonen på instituttet har søkelys på fremdrift under midtveisevaluering.
Vi har nå et team av professorer (IBM sitt «ph.d.-utvalg») som deler på deltakelse i alle midtveisevalueringene hver vår/høst og det er de samme som gjennomgår alle prosjektbeskrivelser ved opptak.
Videre vil Institutt for biomedisin fortsetter sitt opplegg for å forberede studentene på midveisevaluering, med søkelys på fremdrift og presentasjonen.
Vi har også et lengre seminar og diskusjonsgruppe om sluttføring av ph.d. prosjektet, avrunding, avhandling og disputas.

4. Opplæringsdelen
a. Hvordan man vurderer instituttets kursportefølje

Studieseksjonen ved instituttet følger opp instituttets kursportefølje.
Vi følger programmet for emneevaluering ved fakultetet.
Ph.d.-studentene er stort sett fornøyde med kursene som tilbys ved UiB (se også svar under om ph.d.-emner).

b. Hvordan ph.d.-emner evalueres.

Vi følger programmet for emneevaluering ved fakultetet.

Flere av våre undervisere på våre ph.d.-emner går av med pensjon og 2 emner går ikke denne våren. Vi skulle derfor ønske oss nye unge undervisere i ph.d.-emner og mer fremtidsrettede kurs/emner, særlig med forberedelse på et arbeidsliv utenfor akademia. Ph.d.-kandidatene ønsker selv også større kontakt med arbeidslivet for en bedre forståelse av hva slags arbeid de kan få etter ph.d.-graden.
Ph.d.-kandidatene er svært positivt til UiB Ferd og alle kjenner til hva Ferd tilbyr av kurs og oppfølging, og de benytter seg av tilbudet. Vår 2024 skal vi ha besøk av Ferd på instituttet.

c. Instituttets registrering av emner til NorDoc-databasen.

Vi har pr. nå dessverre ingen kurs som egner seg.
Vi registrerer gjerne kurs i fremtiden om vi har noen som egner seg.

d. Deltagelse av studenter fra andre institusjoner til egne NorDoc-tilgjengelige kurs.

NorDoc siden med emner virker lovende når vi presenterer det for våre kandidater, men mange vegrer seg da de ikke er sikre på å få godkjent kurset.
Noen kurs er oppført som «graduate» -hva betyr det versus de som er «ph.d.-kurs». Noen er oppført i listen med 25 ECTS, som klart må være feil andre med 1,1 eller 1,6 ECTS, som er litt underlig.
Kandidatene våre hadde helt klart benyttet seg av flere av Online kursene om det hadde kommet tydeligere frem at kurset ville bli godkjent i deres ph.d.-program.

NorDoc - Nordic Doctoral Training in Health Sciences. Home index (nordochealth.net)

5. Forskerskoler

a) Antall nye kandidater som har deltatt på instituttets forskerskole(r) i 2023.

Vi har 10 nye i BBRS, det siste året.
Alle IBM sine kandidater deltar så ofte som de kan i vår forskerskole (BBRS).
Det er et møtepunkt for informasjon om: midtveisevaluering; søknad og fullføring av forskningsopphold i utlandet; praktiske råd ved reise på konferanser; full gjennomgang av innlevering og disputas; et sted å komme med tilbakemeldinger til «fremdriftsrapportering» og «utdanningsmelding».
Det er også et sosialt møtepunkt og et sted hvor man får anledning til å presentere prosjektet sitt i en «low key» setting.

b) En kort oppsummering av erfaringer med FS-registrering av deltagelse i forskerskolene.

Vi oppdaterer FS-registrering hver vår, for nye deltagere i forskerskolen, samt skriver ut de som har disputert.

Ph.d.-koordinator ved Institutt for Biomedisin, Anne Mette Søviknes.

UNIVERSITETET I BERGEN
Institutt for global helse og samfunnsmedisin

Til orientering i instituttrådet sak xx/24

Forskerutdanningsmelding 2023 - Institutt for global helse og samfunnsmedisin

I 2023 ble det ved Institutt for global helse og samfunnsmedisin (IGS) avlagt 28 doktorgrader, fordelt på 19 kvinner og 9 menn. Av disse var 14 fra Senter for internasjonal helse (SIH) og deriblant 2 av kandidatene i dobbeltgraden med Hawassa University. Av de resterende var det 3 fra Fagområde for allmennmedisin, 4 fra Fagområde for helsevitenskap 1 fra Fagområde for eldremedisin, samfunnsfarmasi og tverrprofesjonell samarbeidslæring og 2 fra Fagområde for etikk og helseøkonomi, 3 fra Fagområde for epidemiologi og medisinsk statistikk og 1 fra Fagområde for samfunnsmedisin, arbeids- og miljømedisin, helseledelse. Av de 28 disputasene hadde alle bortsett fra 3 opponenter av begge kjønn. Ved utgangen av 2023 hadde IGS ca. 150 ph.d.-kandidater og 9 forskerlinjestudenter.

Oppfølging av forskerutdanningsmelding 2022 og tiltak som er gjort.
Ved IGS skal alle ph.d.-kandidater være tilknyttet en forskningsgruppe. Forskningsgruppene ved IGS har i sine strategiplaner at det er en prioritert oppgave å involvere og inkludere master- og forskerlinjestudenter og ph.d.-kandidater i sine aktiviteter.

Instituttet har to egne forskerskoler: Forskerskolen i samfunnsmedisin og CIH-CISMAC Research School. I tillegg til disse er det seks nasjonale tematiske forskerskoler som instituttet er/har vært en del av: EPINOR (For epidemiologi), NAFALM (for allmennmedisin), NFIF (for farmasi), NORBIS (for biostatistikk), NRSGH (for global helse) og MUNI-HEALTH-CARE (for kommunale helse- og omsorgstjenester). Av disse tok prosjektfinansieringen for NAFALM, NFIF og EPINOR slutt i løpet av 2021, og NRSGH hadde finansiering til juni 2023. NAFALM eksisterer fortsatt og har samme kursportefølje, men ingen finansiering for deltakende ph.d.-kandidater. IGS hadde 11 aktive medlemmer fordelt over 3 kull i 2023.

Forskerskolen i samfunnsmedisin hadde tre samlinger i 2023 med temaer som er relevante for ph.d.-kandidatene på tvers av fagområdene:

- Seminar om litteratursøk med bibliotekarer fra Universitetsbiblioteket
- Lunsj til lunsj kurs på Solstrand hvor SpeakLab ga trening i hvordan å snakke og formidle på god og tydelig måte.
- Finansieringsmuligheter i slutten av ph.d.-perioden hvor forskningsrådgiver Margareth Bittins introduserte mulige kilder for finansiering og Line Berge ga noen eksempler på hva hun har søkt på.

31 personer deltok på et eller flere av seminarene vi hadde og er registrert inn i FS under forskerskole i samfunnsmedisin.

CIH-CISMAC forskerskolen sitt program for 2023 bestod hovedsakelig av midtveis-evalueringer, samt CISMAC-webinarer og utvikling av metodekurs. Det var høy aktivitet i forskerskolen gjennom et stort antall disputaser dette året (15), og bidrag til flere kurs og arrangementer organisert gjennom den nasjonale forskerskolen (Norwegian Research School of Global Health), deriblant en 4-dagers ph.d.-konferanse i Lofoten. Mange av disputasene var online, og kandidatene ble oppfordret til å delta på disse for å lære hvordan disputasene foregår, i tillegg til å lære om tematikken under disputasene.

Oversikt over antall aktiviteter i 2023:
Midtveisevalueringer: 11
CISMAC webinar: 5

Forskerskolemøter: 4

CIH-CISMAC forskerskolen ledes av en vitenskapelig ansatt sammen med senterleder. Kriterie for registrere deltakelse i denne forskerskolen i FS er at man har holdt innlegg på et forskerskolemøtene. Dette oppdateres noen ganger i løpet av semesteret. Ved utgangen av 2023 var det registrert 11 ph.d.-kandidater som medlem i CIH/CISMAC forskerskolen. På webinarene har det vanligvis variert litt mellom 10 og 20 deltakere.

Forsinkelser
21% av kandidatene ved IGS rapporterer at det har skjedd endringer som har fått konsekvenser for fremdriften i forhold til opprinnelig plan. Dette samsvarer med veiledernes rapportering. 47% melder at de er blitt forsinket pga. korona. 55% svarer at planlagte innleveringstidspunkt samsvarer med opprinnelig plan.

Veiledning
86 % av kandidatene som har svart i framdriftsrapporteringen er svært fornøyd (56,5%) eller fornøyd (29,5%) med veiledningsforholdet, 8% av kandidatene er delvis fornøyd, og en kandidat melder delvis misfornøyd. Frekvensen på kontakt med veileder er det 90% som er svært fornøyd eller fornøyd, 8% som bare er delvis fornøyd, og 1 kandidat melder delvis misfornøyd og 1 misfornøyd. Vi har kontaktet alle som har meldt inn noen form for misnøye og følger dem opp.

Opplæringsdel
49% rapporterer at de har fullført opplæringsdelen.

Disputaser
I 2023 var det 28 disputaser, en kraftig oppgang fra året før. Deriblant var 8 av disputasene gjenomført digitalt på Zoom. Disse var alle tilknyttet internasjonale kandidater med tilhørighet til SIH. At de har vært digitale har gjort at deltakelsen har økt sammenlignet med før når de var holdt fysisk. Noen få av de fysiske disputasene i 2023 ble streamet pga. store fagmiljøer utenfor Bergen som ønsket å delta eller at en opponent ikke hadde anledning til å komme til Bergen.

Kandidatenes bidrag til forskning ved instituttet
Kandidatene ved IGS er som nevnt integrert i forskningsgrupper, og bidrar i stor grad til forskingsmiljøene og til framdrift i prosjektene de er engasjerte i. Forskningsgruppene vurderer hvor godt kandidatene utfyller eksisterende forskningsfelt i prosessene med å skaffe finansiering og stipend. Prosedyrene ved rekruttering er ulike ved ulike rekrutteringskilder, der de tre viktigste er prosjektfinansierte stipender, UiB stipend og Forskerlinjestipend. I alle tilfeller gjøres det strategiske vurderinger ved rekrutteringen, men i aller størst grad for prosjektfinansierte kandidater som skal løse relativt konkrete forskningsutfordringer. De strategiske vurderingene gjøres i hovedsak av prosjektledelse for disse kandidatene, mens UiB finansierte kandidater inkludert forskerlinjen blir vurdert av Utvidet Forskingsledelse ved fakultetet etter kriterier som inkluderer strategiske forhold ifht forskningsgruppe og veiledere. Både eksterne og interne finaniseringskilder legger stor vekt på at forskningen skal være nyskapende.

Oppfølging av framdriftsrapporter, midtvegsevalueringer
Vi er fornøyd med gjennomstrømmingen av ph.d.-kandidater. Instituttet har til enhver tid om lag 150 kandidater i programmet. Tilkomsten av nye kandidater følger antallet disputaser, med rundt 30 kandidater årlig.

Faglig oppfølging underveis er vesentlig for å sikre progresjon. Et viktig virkemiddel for oppfølging er midtveisevaluering, samt oppfølging av kandidater på bakgrunn av årlige framdriftsrapporter. Forskningsleder og/eller ph.d.-koordinator tar kontakt med kandidater og veiledere som har uttrykt misnøye på ett eller flere punkter i framdriftsrapporten. Vi har også innført en rutine hvor forskningsleder får tilsendt alle midtveisevalueringer for å lettere kunne følge opp kandidater som er vurdert som å ha en bekymringsverdig fremdrift. Oppfølging av kandidater som hører til CIH-CISMAC Research School blir av forskningsleder delegert til senterleder for SIH.

Gjennomføring av medarbeider- eller oppstartssamtaler med ph.d.-kandidatene utover ordinær veiledning
Ph.d.-kandidater med UiB-stipend skal som tilsatt innkalles til medarbeidersamtale. Ansvar for medarbeidersamtaler er delegert til fagområdelederne. Hvert fagområde har mulighet til å delegere medarbeidersamtaler til forskningsgruppeledere, men det kan ikke delegeres til veileder.

Instituttets kursportefølje og evaluering av disse:
Instituttet har en relativ stor ph.d.-kursportefølje der flere av ph.d.-kandidatene våre har deltatt:

- INTH914 Applied economic evaluation in health care
- INTH950 Equity and fairness in health - an applied approach to ethics
- INTH921 Experimental epidemiology
- INTH928 Global tuberculosis Epidemiology and Intervention
- INTH944 Migration and Health
- INTH956 Observational epidemiology
- MEDEPIDEM-A Epidemiologiske prinsipp og metodar
- MEDSTA2 Regression models in medical research
- MEDSTA3 Analysis of longitudinal and correlated data
- MEDKVFORSK2 Kvalitative forskningsmetoder - fordjuping i analysemetoder og tradisjonar

Oversikt over antall ph.d.-kandidater som deltok i emnene de siste fem årene:

Emnekode/semester	2023	2022	2021	2020	2019
INTH914/Vår	11	1	4	2	1
INTH950/Vår	14			1	
INTH921/Vår	6	5	2	7	4
INTH928/Vår	1	2	1		1
INTH944/Vår	1	4		1	
INTH956/Vår	11	7	5	8	5
MEDEPIDEM-A/Høst	24	17	18	14	16
MEDSTA2/Vår	21	21	32	32	32
MEDSTA3/Høst	13		7	11	6
MEDKVFORSK2/Vår	11	14	7	10	10

De formelle evalueringene av emnene har stort sett vært gode.

Andre forhold av relevans for meldingen

Forsknings- og innovasjonsstrategien til IGS oppmuntrer til internasjonalisering. Brorparten av instituttets ph.d.-kandidater deltar vanligvis på internasjonale kurs og seminarer. I instituttets strategi blir det lagt vekt på tilrettelegging for utveksling og forskningsopphold i utlandet, og for fortrinnsvis gjennom ekstern finansiering. Flere av ph.d.-kandidatene ved IGS bor og jobber i lavinntektsland. Avstand og reisetid, reisekostnader og klimahensyn krever vurdering av reiser. God bruk av digitale format innen utdanning og oppfølging av ph.d.-kandidater, samt god organisering og samkjøring av aktiviteter, kan spare tid og redusere behovet for reiser.

Utdanningsmelding for 2023- IKO

Oppfølging av ph.d.-utdanningsmeldingen 2022 og eventuelle tiltak.

Forskerskolen har økt sosial og faglig aktivitet blant kandidater ved IKO og andre institutt bl.a. vha. Journal Clubs og seminar. Seminarserien i oral helse, ODORS901 startet opp i 2022. Første mandag hver måned holdes det ODORS901-seminar for alle ph.d.-kandidatene. En vitenskapelig ansatt er tilstede på seminarene og ulike emner blir diskutert. Ph.d.-kandidatene får også mulighet til å diskutere arbeidet sitt på seminarene. ODORS901-seminarserien er særdeles vellykket og initierer gode faglige diskusjoner.

Kandidatens bidrag til forskningen ved instituttet.

(Forhold som kan omtales: Strategiske vurderinger som gjøres ved ansettelse og opptak av kandidater. Hvordan kandidatene utfyller allerede eksisterende forskningsfelt. Graden av nyskapende forskning).

Instituttets ph.d.-kandidater er vesentlig for forskning og publisering ved instituttet, og bidrar sterkt til den anerkjennelsen som forskningen ved IKO får. Kandidatene er del av instituttets forskningsgrupper og drar nytte av forskningsgruppenes aktiviteter og kompetanse. Forskningsgruppene utfører innovativ og translasjonell forskning der kandidatene er sterkt involvert.

Instittets forskingsstrategi fra 2019 viser at forskerutdanning er et av hovedtemaene. Instituttet har satt seg som et mål å ha et aktivt faglig læringsmiljø for tilegning av vitenskapelige, praktiske og analytiske ferdigheter for våre doktorgradskandidater, samtidig som alle vitenskapelig ansatte i faste stillinger skal være involvert i doktorgradsutdanningen ved instituttet. I strategien pekes det mot 11 konkrete tiltak som skulle iverksettes for å oppnå målene. Disse tiltakene var rettet både mot rekruttering av eksternfinansierte ph.d.-kandidater, profilering av ph.d.-kandidater, økning av deres kompetanse og ferdigheter, og styrking av veilederes engasjement og kompetanse. I løpet av perioden 2019-2023 har en rekke eksternfinansierte kandidater blitt ansatt hos IKO, men ingen øremerkede kandidater har blitt ansatt.

Fakultetets øremerkede ph.d.-stillinger for odontologiske kandidater er et viktig strategisk virkemiddel for å opprettholde god og innovativ odontologisk forskning. Det skal til enhver tid være 13 øremerkede ph.d.-stillinger for kandidater med odontologisk bakgrunn. Flere av disse blir ikke utlyst etter hvert som kandidater sluttfører. I fagmiljøet er det nå stor etterspørsel etter disse stillingene. For at odontologisk forskning ikke skal bli marginalisert må fakultetet utlyse disse øremerkede stillingene fortløpende.

Midtveisevaluering

Instituttet har god struktur på gjennomføringen av midtveisevalueringer. Komiteen til midtveisevaluering består av to vitenskapelige ansatte der en av dem er medlem i forskningsutvalget. I 2023 hadde IKO 6 midtveisevalueringer. Inntrykket vårt er at midtveisevalueringen er nyttig både for kandidatene og for instituttet.

Opplæringsdelen:

- **Hvordan man vurderer instituttets kursportefølje.**
- **Hvordan ph.d.-emner evalueres.**

Instituttet vurderer kontinuerlig behovet for endring og oppdatering av kursportefølje på ph.d.-nivå og evaluerer om endringer trengs. For tiden har instituttet følgende emner på ph.d.-nivå:

ODORS901	- Seminarserie i oral helse
ODFLA902	- Kurs i forløpsanalyse
ODBIO901	- Oral biologi

Følgende kurs under spesialisering i odontologi tas også ofte av ph.d.-kandidater ved IKO og andre institutt:

ODO-STAT1/06:	- Generell statistikk
ODO-STAT2/06	- Anvendt statistikk for den odontologiske videreutdanning -
ODO-FORSK/06	- Kurs i forskningsmetode
OD2ORBI:	- Oral biologi del 2
ODO-ORPAT/06:	- Oral patologi for spesialistutdanning i odontologi

Instituttets registrering av emner til NorDoc-databasen.
Deltagelse av studenter fra andre institusjoner til egne NorDoc-tilgjengelige kurs.

IKO sine kurs på ph.d.-nivå har ikke så langt blitt registrert i NorDoc-databasen, men vi vurderer fortløpende mulighetene.

Forskerskoler:

- **Antall nye kandidater som har deltatt på instituttets forskerskole(r) i 2023**.
  6 nye kandidater deltok på IKO sin forskerskole i 2023 i tillegg til de eksisterende kandidatene.

- **En kort oppsummering av erfaringer med FS-registrering av deltagelse i forskerskolene.**
  IKO har fulgt fakultetets retningslinjer for å registrere kandidater i forskerskolebildet i FS. Kandidatene registreres i Forskerskolen til IKO i FS når de har fullført kurset ODORS901.

Instituttene står fritt til å benytte anledningen til å kommentere andre forhold ved ph.d.-utdanningen dersom det er ønske om det.

Fakultetets øremerkede ph.d.-stillinger for odontologiske kandidater er et viktig strategisk virkemiddel for å opprettholde god og innovativ odontologisk forskning. Det skal til enhver tid være 13 øremerkede ph.d.-stillinger for kandidater med odontologisk bakgrunn. Flere av disse blir ikke utlyst etter hvert som kandidater sluttfører. I fagmiljøet er det nå stor etterspørsel etter disse stillingene. For at odontologisk forskning ikke skal bli marginalisert må fakultetet utlyse disse øremerkede stillingene fortløpende.

Ph.d.-utdanningsmelding 2023

UNIVERSITETET I BERGEN
Det medisinske fakultet

Innhold

Oppfølging av foregående års ph.d.-utdanningsmelding og eventuelle tiltak som har blitt gjennomført

I 2023 har Det medisinske fakultet (MED) arbeidet systematisk videre med kvaliteten i ph.d.-programmet i lys av Strategiplan for forskerutdanning[1], oppfølging etter ekstern programevaluering og oppfølging av punkter fra ph.d. utdanningsmelding for 2022.

- *Onboarding* er en helhetlig mottaksprosess av alle nye kandidater og inkluderer tildeling av pliktarbeid. Fakultetet utarbeidet og vedtok retningslinjer for samordning, registrering og oppfølging av pliktarbeid til 4-årige UiB-stipendiater i 2023[2]. Instituttene har startet arbeidet, men det er enda noe tidlig å evaluere effekten av tiltaket. Fakultetet følger opp i dialogmøter med instituttene i 2024.
- Innføring av retningslinjer for bruk av digitale elementer i disputaser, dvs, hybrid eller heldigital disputas[2].
- Økt veilederkompetanse: Fakultetet har søkelys på styrket veilederkompetanse og kvalitet i veiledningen gjennom målrettet veilederopplæring.
    - April 2023: Heldagsseminar for ph.d.-veiledere med temaet *Kvalitet i kappen for artikkelbasert avhandling* ved merittert underviser professor Rune Krumsvik, Institutt for pedagogikk ved Det psykologiske fakultet.
    - November 2023: Veilederfrokost med foredrag om *Publiseringsetikk* av professor Trine B. Haugen, vitenskapsombud ved Oslo MET og medlem av De nasjonale etiske komiteer.
    - Revisjon av det obligatoriske digitale veilederkurset er i avslutningsfasen. Kurset er oppdatert etter ny ph.d.-forskrift og fakultetets utfyllende retningslinjer og inkluderer undervisningsmoduler med forskningsbasert veiledningspedagogikk til pedagogisk påfyll for veilederkompetanse.
- Systematisk revisjon av grunnkurset i helsefaglig forskning av emneansvarlig professor Kristine Bærøe. MEDMED901 *Etikk og helseforskning* er utvidet til 7 studiepoeng og oppfyller kravet til vitenskapsteori og etikk i ph.d.-forskriften. Kurset omfatter forskningsetikk, vitenskapsfilosofi og innovasjon og er obligatorisk for alle som tas opp i ph.d.-programmet ved MED. Bærekraft i forskning inkluderes også ved at kandidatene gjennom gruppearbeid skal reflektere over og analysere ph.d.-prosjektet med tanke på konsekvenser forskningsresultatene kan ha for økonomi, miljø og sosial rettferdighet, og for nåværende og fremtidige generasjoner (i tråd med WHOs/Brundtlands konsept for bærekraft). MEDMET901 undervises første gang våren 2024.

- Synliggjøring av kvalitet i ph.d.-utdanningen ved utdeling av to priser for Årets ph.d.-arbeid for 2022. En av prisvinnerne delte sine erfaringer til glede og motivasjon for nye kandidater på det årlige oppstartseminaret i 2023.

Nøkkeltall

Opptak og gjennomføring

I 2023 tok fakultetet opp 78 nye kandidater. Opptakene fordelte seg slik: 14 ved Institutt for global helse og samfunnsmedisin, 18 ved Klinisk institutt 1, 34 ved Klinisk institutt 2, 10 ved Institutt for biomedisin og 2 ved Institutt for klinisk odontologi. Dette er en nedgang fra 2022 da vi tok opp 109 nye kandidater. Nedgang i opptakstall kan skyldes at (i) reduksjon i fakultetets egenfinansierte-stipendiatstillinger i 2023 av økonomiske grunner og at to store finansieringskilder ikke lyste ut midler, a) NFR lyste ikke ut FRIPRO-midler og b) Kreftforeningen finansierer ikke lenger stipendiatstillinger.

Totalt ble det avlagt 80 doktorgrader ved fakultetet med følgende fordeling: 28 ved Institutt for global helse og samfunnsmedisin hvorav 1 dr.philos., 17 ved Klinisk institutt 1, 25 ved Klinisk institutt 2, 5 ved Institutt for biomedisin og 5 ved Institutt for klinisk odontologi.

Dette er en nedgang fra 99 disputaser i 2022. Nedgang i disputaser kan skyldes at koronapandemien førte til forsinkelser for planlagt innlevering og disputas i 2023. Vi opplever imidlertid økt pågang av innleveringer for disputas tidlig i inneværende vårsemester, som kan indikere at antall innleveringer vil ta seg opp i 2024.

[1] Strategiplan for forskerutdanning | Det medisinske fakultet | UiB

[2] Reglement og retningslinjer | Det medisinske fakultet | UiB

Antall kandidater som disputerte innen 6 år ved utgangen av 2023 (startår 2017) var 84 (Figur 1a), tilsvarende 72% (n=117). Tilsvarende tall for startår 2016 ved utgangen av 2022 (n=108) var 76 stk/70%. For opptaksårene 2015 og 2014 var tallene hhv ca 66 og 81% (ikke vist). Fakultetet hadde et toppår i 2020 med 113 avlagte doktorgrader, hvor noe av bidraget kan ha kommet fra kandidater som var ferdig etter 6 år med oppstart i 2014.
Instituttene og fakultetet jobber systematisk for at kandidater som er på overtid skal levere avhandling og fullføre doktorgraden. Noen skrives ut av programmet etter at opplæringsdelen er fullført og leverer og disputerer på et senere tidspunkt. Andre får tett og individuell veiledning både faglig og menneskelig og står i programmet til innlevering og disputas. Vi ser at det gir resultater og vi har eksempler på dette blant dem som disputerte i 2023 hvor seks hadde stått i ph.d.-programmet i over 8 år (bruttotid). MED har kontinuerlig søkelys på kvalitet for gjennomføring og er glad for at tendensen er stigende de tre siste årene.

1 a) Andelen ph.d.-kandidater på MED som gjennomfører innen seks år

Indikator-år	Start-år	Disputert				
		Ja	Ja	Nei	Nei	
		Andel	Antall	Andel	Antall	Total
2023	2017	71,79 %	84,0	28,21 %	33,0	117,0
2022	2016	70,37 %	76,0	29,63 %	32,0	108,0
2021	2015	66,30 %	61,0	33,70 %	31,0	92,0

1 b) Avlagte doktorgrader MED - Gjennomstrømming alle

År	Termin		Finansieringskilde			
2023			ANDRE	EGEN	NFR	Total
	Høst	Avg. BRUTTO_ÅR	4,8	5,0	4,7	4,9
		Avg. NETTO_ÅR	3,2	3,8	4,1	3,4
		Antall	21,0	9,0	3,0	33,0
	Vår	Avg. BRUTTO_ÅR	5,70	5,30	4,10	5,40
		Avg. NETTO_ÅR	4,20	3,60	3,30	3,80
		Antall	23,0	17,0	6,0	46,0

Figur 1 . a) Andelen ph.d.-kandidater på MED som gjennomfører innen seks år [3]. b) Avlagte doktorgrader på MED - Gjennomstrømming alle[4].

Oppløsning av gjennomstrømmingstall for finansieringskilder fra Tableu (Fig. 1 b) - Egen utregning

Tabell 1: Oppløsning av finansieringskategoriene fra Tableu (Fig. 1 b) - Egen utregning

Finansieringskategori[5] Finansiering og ansettelse Disputaser vår og høst 2023	Antall	Gjennomsnittlig bruttotid (totaltid i doktorgradsprogrammet) (år)	Gjennomsnittlig nettotid (medgått tid til doktorgradsarbeidet) (år)
ANDRE Eksternt finansierte. Ansatt i sykehus og helseforetak.	25	4,6	2,9
ANDRE Eksternt finansierte kandidater. Ansatt i andre foretak inkl. kandidater fra andre kontinenter	19	6,2	4,7
EGEN UiB-ansatte	26	5,2	3,7
NFR UiB-ansatte	5	4,9	3,7
NFR Eksternt ansatte	4	3,6	3,4

ANDRE-finansierte kandidater

Kategorien *ANDRE* omfatter eksternt finansierte, eksternt ansatte kandidater, unntatt NFR[5]. For MED inkluderer dette ansatte i Helse Vest RHF og Helse Bergen, Haukeland Universitetssykehus, men også andre sykehus, UH-institusjoner, instituttsektoren, ideelle organisasjoner og internasjonale samarbeidsinstitusjoner.

[3] DBH - Andelen ph.d.-kandidater som gjennomfører innen seks år: Oversikt - Tableau Server (uhad.no)
[4] DBH - Avlagte doktorgrader og tidsbruk: Avlagte doktorgrader - gjstr. alle - Tableau Server (uhad.no)
[5] Database for statistikk om høyere utdanning - DBH (hkdir.no)

Når man bryter ned Tableu-tallene for gjennomstrømningstid (Tabell 1) for avlagte doktorgrader i 2023 (Figur 1 b) ser man at eksternt ansatte i norske sykehus og helseforetak har lavest nettotid, 2,9 år. Leger som tar doktorgrad på deltid, ofte med 50% progresjon, samtidig som de arbeider i klinikken er inkludert i denne gruppen. Hvis vi kun ser på de som har opptaksgrunnlag *Cand.med*. er brutto- og nettotid[6] hhv. 4,9 og 2,7 år (16 stk. ikke vist i Tabell 1). Tallene for nettotid tyder på at leger i doktorgradsprogrammet gjennomfører doktorgraden på normert tid i kombinasjon med klinisk arbeid.

Gjennomstrømningstiden for ANDRE-finansierte som ikke er ansatt i norske helseforetak og sykehus inkluderer også internasjonale kandidater finansiert gjennom NORAD, lånekassen, tuberkuloseprogrammer og internasjonale fellesgradsprogrammer. Disse hadde gjennomgående høy brutto- og nettotid. De fleste av disse internasjonale kandidatene disputerte om våren. Tilknytning og arbeidssted på kontinenter fjernt fra UiBs campus kan være utfordrende med tanke på kontinuitet i gjennomføring, for eksempel av geopolitiske årsaker. Global helse er ett av satsningsområdene på MED, som har eksellente forskningsmiljøer innen fagfeltet, som gir MED internasjonal kredibilitet og synlighet i dette feltet. De jobber gjerne med folkehelse eller påvirkning på regjeringsnivå, som gjør forskningen sårbar for geopolitiske spenninger. Vi er glade for at disse har lykkes i å fullføre sin doktorgrad.

Egenfinansierte ansatt på UiB har en gjennomsnittlig nettotid på 3,7 år, på linje med NFR-finansierte ansatt på UiB. Stipendiater med ett års pliktarbeid inngår i denne kategorien. Pliktarbeid øker bruttotiden på lik linje med andre ikke-rettighetsbaserte forlenginger. Tidligere forskerlinjestudenter, som ikke har pliktarbeid, inngår også i denne kategorien. Disse har i utgangspunktet en avkorting av tiden pga. tidligere fullført opplæringsdel. Men også her vil ikke-rettighetsbaserte forlenginger øke bruttotiden. Ulike omstendigheter påvirker, med andre ord, bruttotiden. For å se hvor mye tid som faktisk går med til doktorgradsarbeidet, må man se på nettotiden.

Doktorgradsløpet er normert til 3 år. Nettotiden påvirkes ikke av forlenginger som gis pga. rettighetsbasert fravær som sykdom og foreldrepermisjon. Dersom nettotiden øker, betyr det at det har vært behov forlenging av tiden til ferdigstillelse av doktorgradsarbeidet ut over normert tid. Gjennomsnittlig nettotid på 3,7 år antyder et behov for rundt 8 måneders ekstra tid (0,7 år) for å fullføre doktorgraden isolert sett. Dette understreker viktigheten av en god onboardingsprosess som gjør at kandidatene kommer fort i gang med ph.d.-arbeidet.

Mindre omarbeiding

Totalt fire avhandlinger fikk anbefaling om mindre omarbeiding. Dette er en reduksjon fra 12 i 2022. Vi har sett at anbefaling om mindre omarbeiding ofte skyldes svakheter i kappen. Vi er glade for å se en nedgang, som kan antyde at økt oppmerksomhet på kvalitet i kappen har gitt resultater.

Kandidatenes bidrag til forskningen ved fakultetet

Strategiske vurderinger som gjøres ved ansettelse og opptak av kandidater og hvordan kandidatene utfyller allerede eksisterende forskningsfelt

Det poengteres fra instituttene at kandidatene bidrar vesentlig til forskningen i sine respektive fagfelt, er i forskningsfronten og bidrar til fremdrift og anerkjennelse for forskningen ved instituttet.
Strategiske vurderinger ved ansettelse og opptak gjøres på instituttene og i forskningsgruppene i lys av instituttets forskningsstrategi og basert på utlysninger fra aktuelle eksterne finansieringskilder som NFR, Helse Vest, Kreftforeningen, DAM-stiftelsen, Tannhelsetjenestens kompetansesentre og tilfang av UiB-finansierte stipendiater. Innen flere fagområder, som ernæring og biomedisin, rekrutteres ph.d.-kandidater særlig gjennom masterprogrammene.

Rekrutteringsgrunnlaget varierer fra etablerte klinikere som tar doktorgrad på deltid parallelt med arbeid i sykehus eller tannhelsetjeneste til tilsatte UiB-stipendiater. De kliniske instituttene satser på rekruttering av LIS-leger lokalt og i distriktene inn i forskningsaktiviteten med tanke på framtidig rekruttering til ph.d. og kombinerte stillinger i akademia. Forskerlinjestudenter er viktig for rekruttering av unge, dyktige kandidater til forskerutdanningen. Det er et strategisk grep for (i) å få tidlig forskningskompetanse under legeutdanningen ihht. regjeringens langsiktige strategisk plan, (ii)

[6] 10 Tidskonto - Felles studentsystem: Nettotid er antall årsverk studenten har brukt på studiet minus alle permisjoner og annen tid som ikke er brukt til studiet.

senke alder for de som tar ph.d. og (iii) fullføre på kortere tid pga. tidligere løp i forskerlinjen. Seks av dem som disputerte i 2023 hadde bakgrunn fra forskerlinjen ved fakultetet.

Graden av nyskapende forskning og økt oppmerksomhet på innovasjon i forskerutdanningen

Kandidatene bidrar til nyskapende forskning. Innovasjon er et satsingsområde på fakultetet og i oktober 2023 arrangerte MED et halvdags oppstartseminar for nye kandidater hvor innovasjon var ett av hovedtemaene. Kandidater med innovasjonsrettede prosjekter kan få kontorplass i Eitri. Kandidatene oppmuntres til å inkludere et innslag i disputaspresentasjonen som viser hvordan deres ph.d.-prosjekt kan bidra til innovasjon i fagfeltet, hva som er nytt og verdifullt med prosjektet og hvilke utviklingsmuligheter som kan ligge i forskningsresultatene og føre til merverdi for sluttbruker.

Forskerskoler

MED har åtte lokale forskerskoler som bidrar med formell faglig kompetanse, overførbare ferdigheter og psykososialt velvære, og er forankret på instituttene. I tillegg deltar MED i en rekke nasjonale forskerskoler[7]. Kandidater deltar også på tvers av instituttenes forskerskoler. Alle kandidatene skal være tilknyttet en forskerskole for faglig og sosial forankring i forskningsmiljøene.

Fakultetet har 54 aktive ph.d.-emner. De fleste er knyttet til og administrert gjennom forskerskoler ved instituttene og noen få er administrert på fakultetet.

Fakultetets systematiske kvalitetsarbeid

Oppfølging av framdriftsrapporter

Instituttene følger opp fremdriftsrapportene og iverksetter tiltak overfor enkeltkandidater og veiledere der rapportene signaliserer slikt behov. Fakultetet har årlige dialogmøter med instituttledelsen om ph.d.-utdanningsmeldingen, framdriftsrapportene, utviklingsprosjekter i forskerutdanningen og oppfølging av enkeltkandidater. Disse møtene er en viktig arena for tilbakemeldinger og drøftinger mellom institutt og fakultet. Det er ulike utfordringer på de forskjellige instituttene ettersom de har svært ulikt omfang på forskerutdanningen. Derfor er det viktig å ha god dialog med instituttene for kontinuerlig kvalitetsutvikling.

Midtveisevaluering

Alle ph.d.-kandidater ved MED skal gjennomføre midtveisevaluering omtrent halvveis i sin avtaleperiode. Målet er å hjelpe kandidaten og veileder til å vurdere om kandidaten er i rute i henhold til oppsatt plan. Oppfølging etter midtveisevaluering tilpasses behovet til hver enkelt kandidat i regi av instituttet.

Gjennomføring av medarbeider- eller oppfølgingssamtaler

Det er ressurskrevende, spesielt for institutter med mange kandidater, å gjennomføre medarbeidersamtale for alle ansatte stipendiater og oppfølgingssamtaler for alle de eksternt ansatte kandidater. Ansvaret for gjennomføring av samtalene er i stor grad delegert til forskergruppene, men veileder skal ikke gjennomføre slike samtaler. Begrenset kapasitet skaper utfordringer for store institutter med mange eksternt ansatte kandidater, hvor det ikke har vært mulig å tilby slike samtaler i 2023. Vi ser likevel at instituttene har tett oppfølging kandidatene gjennom sine ph.d.-koordinatorer og at det iverksettes nødvendige tiltak i forbindelse med gjennomgang av fremdriftsrapportering hvor instituttledelsen involveres ved behov. To institutter melder at de arbeider internt for forbedring av rammene for samtaler i 2024. Institutt for klinisk odontologi kaller inn alle kandidater i ph.d.-programmet til samtale etter fremdriftsrapportering og har i 2023 utvidet samtalen ved å inkludere samtalepunktene fra mal for medarbeider- eller oppfølgingssamtaler[8]. Fakultetet anerkjenner instituttenes bestrebelser for best mulig oppfølging av dette punktet i kvalitetssikringssystemet for ph.d.-utdanningen.

Opplæringsdelen

Den obligatoriske delen av opplæringen består av MEDMET901 *Etikk og helseforskning* midtveisevaluering, formidling, og opplæring i bruk av forsøksdyr i medisinsk forskning der det er aktuelt. I den valgfrie delen tar kandidatene i stor

[7] Forskerskoler | Det medisinske fakultet | UiB.
[8] Kvalitetssystem for ph.d.-utdanningen ved UiB | Ansattsider | UiB

grad UiB-emner eller emner fra andre universiteter. Fakultetet er også medlem i NorDoc-samarbeidet og bidrar med ph.d.-emner til NorDoc emnedatabase[Feil! Bokmerke er ikke definert.]. Kandidater ved alle de nordiske universitetenes medisinske fakulteter har lik tilgang til å ta emner som finnes i databasen.

Emneevalueringer

Emneansvarlig skal gjennomføre egenvurdering og følge opp resultatene som videre danner grunnlag for fast evaluering av ph.d.-emner minst hvert tredje år eller etter hver tredje gang emnet er gjennomført. Programutvalget for forskerutdanningen godkjente våren 2022 en tidsplan for emneevaluering, som er fulgt opp med planlagt evaluering av 11 emner i 2023.

DET MEDISINSKE FAKULTET

PROGRAMUTVALGET FOR FORSKERUTDANNING

MØTE 17.04.2024

SAK 09/24

Årsmelding Forskerlinjen 2023

Hva saken gjelder

Forskerlinjeadministrasjonen har også i år utarbeidet Årsmelding for Forskerlinjen. Årsmeldingen tar for seg opptak, frafall, uteksaminerte studenter, aktiviteter, og vedtak med betydning for Forskerlinjen.

- Høsten 2023 fullførte den 212. studenten Forskerlinjen.
- Forskerlinjen tok opp 7 nye studenter i 2023.
- I 2023 var det to studenter som sluttet
- Opptak, undervisning og drift av Forskerlinjen er gjennomført i tråd med planer og vedtak.

Til Forskerlinjen er knyttet ett professorat 60 % stilling, en professor II i 20 % stilling, en 1. amanuensis II i 20% stilling og en rådgiver i 50 % stilling.

Forslag til vedtak

Programutvalg for forskerutdanning tar årsmeldingen til etterretning.

Vedlegg

- Årsmelding for Forskerlinjen 2023

MARSTI/03/24

Årsmelding 2023

Forskerlinjen ved Det medisinske fakultet, Universitet i Bergen

Foto: Anne Berit Guttormsen

INNHOLD

OPPSUMMERING 2023

- Høsten 2023 fullførte den 212. studenten Forskerlinjen.
- Forskerlinjen tok opp 7 nye studenter i 2023.
- I 2023 var det to studenter som sluttet
- Opptak, undervisning og drift av Forskerlinjen er gjennomført i tråd med planer og vedtak.

Til Forskerlinjen er knyttet ett professorat 60 % stilling, en professor II i 20 % stilling, en 1. amanuensis II i 20% stilling og en rådgiver i 50 % stilling.

INNLEDNING

Forskerlinjen drives etter den vedtatte studieplanen. Totalt er 338 studenter tatt opp i perioden 2002-2023, av disse har 57 studenter ikke fullført av ulike årsaker. Ved utgangen av 2023 hadde Forskerlinjen 76 aktive studenter, inkludert studenter tatt opp høsten 2023. Uttak av ordinære permisjoner, som for eksempel svangerskapspermisjoner, vil gjøre at antallet forskerlinjestudenter varierer.

Totalt 9 forskerlinjestudenter ble uteksaminert i 2023. Ledelsen ved forskerlinjen arbeider aktivt med å rekruttere forskningsgrupper som kan ivareta forskerlinjestudentene på en god måte. I starten kreves tett oppfølging og tilrettelegging av arbeidsoppgaver slik at studenten får maksimalt ut av fulltidspermisjonen som er ett år.

KORT OM FORSKERLINJEN

Forskerlinjen er et spesialtilbud for en gruppe lege- og tannlegestudenter som har interesse for forskning og som kan tenke seg en fremtidig forskerkarriere. Fra og med høsten 2010 ble det også åpnet opp for at tannlegestudenter kan søke om opptak, inntil to studenter årlig. Studentene ved Forskerlinjen følger ordinært lege- og tannlegestudium. Spesialtilbudet ved Forskerlinjen består i at studentene i tillegg får en organisert forskeropplæring og utfører egen forskningsaktivitet med skriving av en artikkel. Kravet er en artikkel som første – eller andreforfatter. Dette er arbeid som kan brukes i en framtidig doktorgrad.

Håpet er at mange vil fortsette på et ph.d.-arbeid i løpet av de første årene etter embetseksamen og avsluttet Forskerlinje.

Studentene blir i regelen opptatt ved Forskerlinjen andre høsten etter at de har startet på studiene i medisin eller odontologi (3. semester). Forskerlinjen innebærer at den normerte studietiden på henholdsvis seks og fem år forlenges med ett år. Forskerlinjen baserer seg på arbeid med forskning på full tid i ett år, samt forskning tilsvarende 0,2 årsverk per år parallelt med ordinært lege- eller tannlegestudium. Total mengde forskningsarbeid utgjør to årsverk, men for mange studenter tar forskningen mer tid på grunn av motivasjon og vilje til arbeid utover de formelle kravene.

Fullført Forskerlinje gir 120 studiepoeng i tillegg til lege- eller tannlegestudium; av dette er forskeropplæring 30 studiepoeng, forskningsoppgaven 30 studiepoeng og selve forskningen (det året de er i fulltidspermisjon fra studiet) 60 studiepoeng.

Forskerutdanningsprogrammet ved Forskerlinjen er identisk med utdanning for ph.d.-programmet. Det er ikke nødvendig med ytterligere formell forskeropplæring for ph.d.-kandidater som tidligere har fullført Forskerlinje.

I ph.d.-forskriften for Universitetet i Bergen er det gjort unntak fra §5.2 for forskerlinjestudenter når det gjelder kravet om at minimum ett år av forskningsprosjektet skal gjennomføres etter opptak i ph.d.-programmet. Videre er det gjort unntak for forskerlinjestudenter når det gjelder at

elementer i opplæringsdelen ikke bør være eldre enn 5 år ved opptaksdato (§7-2).

FORSKERLINJETEAMET

I 2023 var følgende personer tilknyttet Forskerlinjen:

Fast ansatte:

- Anne Berit Guttormsen, professor (60 %), leder av Forskerlinjen
- Marianne Stien, rådgiver (50 %)
- Astrid Olsnes Kittang, førsteamanuensis (20 %)
- Torbjørn Østvik Pedersen, førsteamanuensis, kontaktperson for odontologistudenter
- Stipendiater med arbeidsplikt på Forskerlinjen

OPPGAVEFORDELING MELLOM DE ANSATTE

Leder har, sammen med de vitenskapelige i bistilling, det faglige ansvaret for Forskerlinjen, for oppfølging av studentene og forskningsprosjektene samt kontakten med forskningsmiljøene og Fakultetet.

Rådgiveren har administrative oppgaver knyttet til Forskerlinjen, herunder søknader om stipend og permisjoner fra ordinært studium. Daglig kontakt med studentene via e-post eller samtaler er en viktig oppgave. Seniorkonsulenten deltar ellers under de fleste aktiviteter i regi av Forskerlinjen og har oppgaver knyttet til timeplanarbeid, arrangementer, midtveisevaluering, opptak og avslutning.

Stipendiater – det er opprettet 3-årige (-20 uker for fullført opplæringsdel) stipendiatstillinger øremerket Forskerlinjestudenter. Disse stipendene utlyses hver høst. Tilbud om øremerket stipend forutsetter bestått lege- eller tannlegestudium og at studenten begynner i stipendiatstilling umiddelbart etter endt studium. Disse kandidatene er pliktig å arbeide 20 timer årlig ved Forskerlinjen. Dette arbeidet er knyttet til rekruttering, informasjonsarbeid og lignende, etter avtale med ledelsen i Forskerlinjen.

Øremerkede ph.d stipend

Dette tiltaket ble startet opp i 2009. De studentene som har fått innvilget slikt stipend er:

2009**:** Miriam Nyberg (2011*), Lars Thore Fadnes (2011), Tor-Christian Johannessen (2011), Marit Ebbesen (2013) og Eli Fjeld (2011). ***2010*****:** Christian A. Moen (2013), Jochem Cuypers, Siri Herredsvela, Ida Wergeland (2017) og Karen Rebbestad. ***2011*****:** Tone Dolva Dahl (2015), Erik Helgeland (2014), Elisabeth Landaas (2013), Jobin Varughese (2012) og Erling T. Westlye. ***2012*****:** Jintana B. Andersen (2016), Anne Taraldsen Heldal (2014), Ida Wiig Sørensen (2015), Kari-Elise Veddegjærde, Simone B. Reiter (2016). ***2013*****:** Ashraf Fathpour, Arild A. Østhus (2013) og Kristoffer Evebø Sand (2015)**.** ***2014:*** Jan Roger Olsen (2017), Kristi Krüger (2017) og Karen Mauland (2017). ***2015*****:** Kristiane Tislevoll Eide (2020), Karl Erik Müller (2020), Kristine Husøy Onarheim (2018), Carl Tollef Solberg (2018), Sanjeevan Sriskandarajah (2018). ***2016*****:** Anna Therese Bjerkreim (2019), Tor-Arne Hegvik (2019), Henriette Aurora Selvik (2018), Sunniva Sakkestad (2021) og Ann Merethe Vågane. ***2017*****:** Fredrik Sævik (2020), Andrea Melberg (2020), Ingeborg Eskerud (2020), Hilde Renate Engerud (2020) og Magnus Bratteberg (2021) (odontologi). ***2018***: Dag Heiro Yi (2021), Aril Løge Håvik (2022), Eigir Einarsen (2020), Agnes Nystad (2020), Einar Marius Hjellestad Martinsen (2021), Jian Hao Liu (2023), Anneli Skjold (odontologi (2022), Elise Orvedal Leiten (2022), Benedicte Sjo Tislevoll (2023), Hanne Hegdahl (2023), Fredrik Hoel. ***2019***: Ane Gagnat (2020), Anders Aarebrot (2021), Helene Aarstad (2022), Philipp Strauss (2023), Torstein Frugård Habiger (2022), Tore Ivar Malmei Aarsland. ***2020***: Tormod Rebnord, Kordian Staniszewski (2023), Christiane Gjerde og Andrea Stautland. ***2021***: Vera Erchinger, Elisabet Kvadsheim (2022), Emily McLean. ***2022***: Sarah Svege,

Linda Xu, Solveig Løkhammer
2023: Nora Hatletvedt, Tamandeep Bharaj, Sehee Rim, Kaya Cetin, Agnes Jørgensen Eide

*Årstall for disputas

AKTIVITETER VÅREN 2023

Oppgavedag: Arrangementet ble denne gangen avholdt fysisk 9. februar 2023. Det kom inn 39 nye forskerlinjeoppgaver. Oppgavene var bredt fordelt mellom de ulike fagmiljøene.

Protokollskriving: For førsteårsstudentene og de av andreårsstudentene som ikke hadde deltatt tidligere, ble det i juni 2023 arrangert seminar i protokollskriving i regi av Eureka.

FLART2: Vi holder ett kurs per år og det er ca. 14 dager mellom hver samling. Totalt 10 samlinger a to timer. Kurset gir 2 studiepoeng. For å få kurset godkjent må studenten delta på 80% av samlingene. På grunn av dårlig påmelding ble kurset ikke avholdt i 2023.

AKTIVITETER HØSTEN 2023

Førsteårsstudentene (kull 2023) ble ønsket velkommen og informasjon om Forskerlinjen ble gitt ved flere anledninger. Første gang i introduksjonskurset som arrangeres de to første ukene av semesteret. Deretter har Forskerlinjen to timeplanfestede presentasjoner der det gis informasjon om Forskerlinjen. I tillegg til forskerlinjeadministrasjonens møter, har EUREKA egne rekrutteringsmøter med de to nyeste kullene.

Andreårsstudenter og tredjeårsstudenter som kan søke opptak på Forskerlinjen 1. oktober, gis praktisk, økonomisk og annen informasjon. Studentene blir i denne perioden fulgt opp enkeltvis om ønskelig, med tanke på søknadsskriving, utforming av faglig essay og lignende i forbindelse med søknaden.

25. august 2023 arrangerte vi oppstartsseminar for studenter som skal ha sin første fulltids forskningsperiode. For andre- og tredje års-studentene (andre år som forskerlinjestudent) arrangeres hvert år et faglig seminar på Voss over to dager. På grunn av fakultetets økonomiske situasjon, ble dette seminaret i 2023 holdt på campus. Dette ble gjennomført 24.-25. november og var svært vellykket. Studentene imponerer både med innsikt og kunnskap, og øver på presentasjonsteknikk. Vi hadde også med en ph.d.-kandidat som tidligere har vært forskerlinjestudent, for å komme med gode råd og tilbakemeldinger til de som presenterte.

Forskerlinjen har fra 2013 gjennomført midtveisevaluering etter fulltids forskningsår. Leder og rådgiver deltar på alle midtveisevalueringer. Studenter, veiledere, medlemmer av komiteene og Forskerlinjens ledelse rapporterer alle om gode erfaringer med denne ordningen.

OPPTAK AV NYE STUDENTER 2023

Ved søknadsfristens utløp 1. oktober hadde vi mottatt 8 søknader om opptak til Forskerlinjen.

Samtlige søkere ble innkalt til personlig samtale hvor forskningsprotokoll og motivasjon ble diskutert. Innkalt til samtalene var også tillitsvalgt/representant fra forskerlinjestudentene. Av de 8 søkerne fikk 7 tilbud om opptak og takket ja til plassen.

STUDENTER UTEKSAMINERT FRA FORSKERLINJEN

I 2023 fullførte følgende studenter Forskerlinjen:

Audun Brendbekken leverte oppgaven «Public participation: healthcare rationing in the newspaper media". Veileder: Ole Frithjof Norheim, Institutt for global helse og samfunnsmedisin.

Leo Larsen leverte oppgaven «Respiratory tract infections in Norwegian primary

care 2006–2015: a registry-based study". Veileder: Knut Erik Emberland, Institutt for global helse og samfunnsmedisin.

Madeleine Myrvold leverte oppgaven «Mismatch repair markers in preoperative and operative endometrial cancer samples; expression concordance and prognostic value". Veileder: Camilla Krakstad, Klinisk institutt 2.

Solveig Boland leverte oppgaven "Verified parental cardiovascular events for young and middle-agedn ischaemic stroke patients and controls". Veileder: Ulrike Waje-Andreassen, Klinisk institutt 1.

Torbjørn Nordrik leverte oppgaven "Sex and Age Differences among Opioid-Tolerant and Opioid-Naive Patients Referred to the Acute Pain Service: Evidence from a Registry-Based Study in Four Norwegian University Hospitals". Veileder: Lars Jørgen Rygh, Klinisk institutt 1.

Maria Olsen leverte oppgaven «Reducing regional health inequality: a subnational distributional cost-effectiveness analysis of community-based treatment of childhood pneumonia in Ethiopia". Veileder: Ole Frithjof Norheim, Institutt for global helse og samfunnsmedisin.

Sehee Rim leverte oppgaven «Dynamics of circulating lymphocytes responding to human experimental enterotoxigenic Escherichia coli infection". Veileder: Kurt Hanevik, Klinisk institutt 2.

Sigrid Nakken leverte oppgaven «AGAP2-AS1 as a prognostic biomarker in low-risk clear cell renal cell carcinoma patients with progressing disease". Veileder: Hans-Peter Marti, Klinisk institutt 1.

Trond Are Mannsåker leverte oppgaven « Cabozantinib Is Effective in Melanoma Brain Metastasis Cell Lines and Affects Key Signaling Pathways". Veileder: Frits Thorsen, Institutt for biomedisin.

STUDENTER SOM HAR SLUTTET PÅ FORSKERLINJEN

Det var to studenter som sluttet på Forskerlinjen i 2023.

TIDLIGERE FORSKERLINJESTUDENTER SOM FULLFØRTE SIN PH.D. I 2023

- Tilde Broch Østborg
- Johannes Just Hjertaas
- Jian Hao Liu
- Benedicte Sjo Tislevoll
- Hanne Keyser Hegdahl
- Philipp Strauss
- Kordian Staniszewski
- Håkon Skogrand Eliassen

FRAMPEIK 2023

Nok en gang reiste en stor delegasjon fra Bergen til den nasjonale forskerlinjekonferansen Frampeik. Denne gangen gikk turen til Trondheim, der den røde tråden var «bærekraftig forskning». Kunnskapsrike foredragsholdere åpnet konferansen med å reflektere rundt hva bærekraftig forskning innebærer. Spesielt paneldebatten om temaet engasjerte studentene i salen, med gjennomtenkte spørsmål og refleksjoner. Forskerlinjen ved UiB var representert med 21 deltakere på Frampeik i Trondheim. 19 hadde muntlige presentasjoner, mens 4 i tillegg presenterte postere. Ingen andre studiebyer kan skilte med så sterk deltakelse! Det ble delt ut en pris for beste poster og pris for beste foredrag i alle de ulike kategoriene; funksjonell forskning, klinisk forskning, translasjonell forskning og helseforskning. Deltakere fra UiB stakk av med 3 av 5 priser. Neda Valeckaite fikk pris for beste poster. Jonas Hodneland Sundfjord fikk pris for beste foredrag i kategorien translasjonell forskning. Annabeth Asphaug fikk pris for beste foredrag i kategorien funksjonell forskning. Forskerlinjestudentene returnerte med mer

kunnskap, større motivasjon og sterkere samhold i gruppen.

Frampeik vil neste år arrangeres av UiT.

STUDENTER VED FORSKERLINJEN OG STUDENTER FORDELT PÅ INSTITUTTER

Aktive studenter på Forskerlinjen pr 31.12.2023:

Kull 17 (12 studenter)

- Amdam, Håkon
- Augustsson, Mina
- Cetin, Kaya
- Eide, Agnes J.
- Enden, Marta R.
- Hatletvedt, Nora
- Skogvold, Thomas Nymo
- Sørbø, Sander T.
- Taule-Sivertsen, Peter
- Teige, Erica P.
- Teigen, David
- Wikerholmen, Tobias

Kull 18 (19 studenter)

- Bharaj, Tamandeep
- Bredin, Hanna
- Erikstad, Kjell Inge
- Gundersen, Jens
- Haugsøen, Jonas B.
- Hugaas, Ulrikke
- Johannessen, Håkon G.
- Khan, Ingela
- Khan, Michelle
- Leto, Nedim
- Lyngstad, Jenny
- Mikkelsen, Håvard
- Nyland, Harald
- Peter, David O.
- Rivedal, Mariell L.
- Saeed, Nazir
- Siyam, Diana
- Taule, Erlend M.
- Tegnander, Amalie F.

Kull 19 (15 studenter)

- Austgulen, Amalie
- Bedi, Ustat
- Bjørsvik, Ben R.
- Brudvik, Egil
- Hagenes, Jonar
- Khan, Ammal
- Kvamme, Amalie B.
- Nguyen, Jens L.
- Sindre, Rasmus B.
- Skjærseth, Idun G.
- Sættem, Magnus
- Søndenaa, Knut Q.
- Taranova, Evgenia
- Thorsdalen, Heidi
- Valeckaite, Neda

Kull 20 (14 studenter)

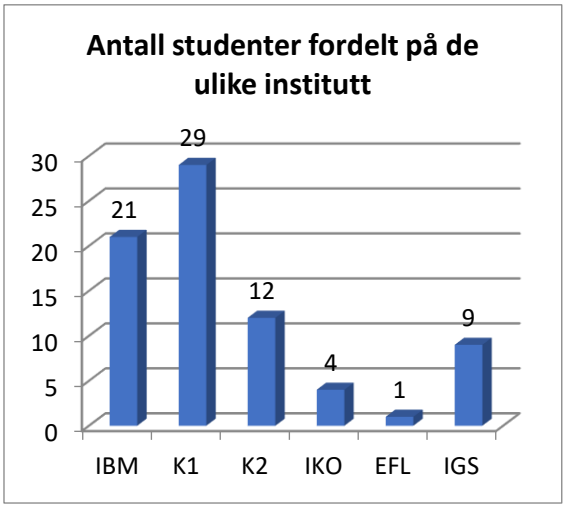
- Asphaug, Annabeth
- Evjenth, Edward
- Fevang, Hauk
- Fjeldstad, Karoline
- Herstad, Kornelia
- Høberg, André
- Kimo, Magnus
- Lauen, Enny S.
- Leerink, Christina
- Longva, Ariell
- Nupen, Tora H.
- Patil, Shridar A.
- Sundfjord, Jonas H.
- Velde, Christoffer D.

Kull 21 (13 studenter)

- Aga, Olav N.L.
- Brunsdon-Håland, Brynmor
- Bø, Maren A.
- Chowdhury, Risha
- Jakobsen, Lise S.
- Johnsen, Stine R.
- Nosal, Natalia
- Rajkumar, Jenny
- Stylianou Lerpold, Anna
- Amanda V. Winther-Waldau
- Magnus Fossum
- Miriam Grønberg
- Einar Naveen Møen

Kull 22 (3 studenter)

- Yuying Sammy Choi
- Marthe Alvestad
- Julia Saltyte Benth



IBM = Institutt for biomedisin

K1 = Klinisk institutt 1

K2 = Klinisk institutt 2

IKO = Institutt for klinisk odontologi

IGS = Institutt for global helse og samfunnsmedisin

EFL = Enhet for læring

ØKONOMI

Stipend til studentene

En forskerlinjestudent får via Fakultetet tildelt stipend fra NFR. Stipendene gjelder ett års forskning på heltid (oppdelt i to halvår) og to år der forskningen skjer på deltid parallelt med ordinært studium. Stipendene utbetales direkte til studenten uavhengig av instituttene.

Slike forskerlinjestipend er definert som «utdanningsstipend utenfor arbeidsforhold» og er følgelig ikke skattepliktig i henhold til skatte-bestemmelsene. I og med at stipendet ikke er lønn og ikke gir lønnsansiennitet, har verken NFR eller universitetet arbeidsgiveransvar for forskerlinje-studenten. Forskerlinjestudenten har verken rettigheter eller plikter som lønnsmottaker.

For tiden er stipendet kr 100.000, - for det hele året og kr 50.000, - x 2 for de to årene på deltid. I tillegg tildeles et sommerstipend på kr. 25.000, -. Dette innebærer 8 ukers fulltids forskning. Redusert sommerstipend kan også søkes for hhv 4 eller 6 uker.

Driftsmidler til instituttet

Fakultetet bevilger driftsmidler til bruk for hovedveileder via instituttet som har forskerlinjestudenten. Beløpet er for tiden kr. 10.000, - pr år i forskerlinjestudentens 3., 4. og 5. studieår. Driftsmidlene overføres instituttets driftsbevilgning (annuum) ved årets start. Vanlige regnskapsrutiner gjelder, som for eksempel bestilling, attestasjon og anvisning. Pengene disponeres av hovedveileder til utgifter med relevans for forskerlinjeprosjektet.

Driftstilskudd til studenten

Sammen med stipendene følger driftstilskudd, hhv kr 32.500, - for det hele året og 2 x 16.250, - for de to årene med deltidsforskning. Driftstilskuddet stilles til disposisjon for

veileder/biveileder og blir overført til instituttet i januar. Det vil si at det kan bli etterskuddsvis i forhold til søknad om eller påbegynt stipendperiode. Pengene skal brukes som "annuumsmidler" for studenten sitt prosjekt, som for eksempel pc, bøker, programvare, kursavgifter, kongressreiser, kjemikalier, og annet utstyr til forskningsprosjektet. Det er utarbeidet retningslinjer for bruk og vanlige regnskapsrutiner gjelder ved bestilling, attestasjon og anvisning.

INFORMASJONSARBEID

Forskerlinjen har utarbeidet nettsider som finnes via Fakultetet:

http://www.uib.no/med/65047/forskerlinjen-ved-det-medisinske-fakultet

På nettsidene finnes informasjon om bl.a.;

- Forskerlinjen generelt
- Studieforløp
- Opptak
- Studieplan
- Aspirantperiode
- Aktiviteter
- Studenter på Fl
- Økonomi
- Forskningsoppgaver
- Skjema
- Godkjenningsprosedyrer
- Veiledning
- Frampeik
- Frequently asked questions (FAQ)
- Tillitsvalgte/EUREKA!

Vi holdt også orienteringsmøte for nye veiledere i 2023.

Dessuten har Forskerlinjen mottatt et stort antall telefonhenvendelser samt henvendelser på e-post. I tillegg har vi hatt møter med potensielle søkere og med veiledere.

Innsamling av nye forskningsoppgaver foregår kontinuerlig. Forespørsel blir sendt ut til samtlige fast tilsatte ved fakultetet, vedlagt informasjon og skjemamal for beskrivelse av prosjektet.

FREMDRIFTSRAPPORTERING

Det skal leveres framdriftsrapport fra både forskerlinjestudent og veileder hvert år. Dette gjøres elektronisk og fungerer meget tilfredsstillende. Disse evalueringsskjemaene er tilnærmet lik de som brukes i ph.d.-programmet, men med små justeringer for Forskerlinjen. Fra 2016 har vi innkalt alle studentene til en samtale etter fremdriftsrapporteringen. Dette foregår på Teams. Dette er tidkrevende, men nyttig, og videreføres. Slike samtaler gjør at vi blir bedre kjent med studentene. Samtalene har i noen tilfeller også bidradd til å løse problemer i relasjonen mellom veileder og student. I samtalene gir vi også tips om karrierebygging for de studentene som ønsker dette.

75 % av studentene meldte om at det ikke hadde skjedd viktige endringer når det gjelder prosjektets arbeidstittel, design/metode, arbeidsplass og utstyr. 88 % meldte ingen endringer i veiledningsforholdet, og 95 % meldte at de var i rute i forhold til forskerutdanningsdelen. De som svarer at de ikke var i rute, følges opp individuelt.

Veilederne meldte at hos 98 % hadde det ikke skjedd viktige endringer når det gjaldt prosjektets arbeidstittel, design/metode, arbeidsplass og utstyr. 94 % meldte at studenten var i rute med forskerutdanningsdelen. Når det gjaldt progresjon vurderte veilederne den som meget bra eller bra i 79 % av tilfellene. 2% meldte om dårlig progresjon.

EUREKA!

Forskerlinjestudentene i Bergen startet i 2010 opp sin egen studentorganisasjon – EUREKA! EUREKA! er en sosial arena som gir Forskerlinjestudentene en tilhørighet og samhold, samtidig som organisasjonen vil være en faglig arena hvor studentene får

inspirasjon, ved blant annet å arrangere møter med inviterte foredragsholdere. De vil jobbe med rekruttering av nye studenter ved Forskerlinjen og forbedre Forskerlinjestudenters rettigheter og muligheter. EUREKA! har også en egen nettside, https://www.uib.no/med/116303/eureka. Eureka kan også følges på Instagram

Siden 2022 har forskerlinjestudentene hatt dynamiske arbeidsplasser på Eitri. Dette er blitt vel mottatt.

VEIEN VIDERE

Forskerlinjestudentene leverer! De ivaretar hverandre og mange yter faglig mer enn det som er kravet for å få forskerlinjen godkjent. På grunn av god progresjon ligger flere an til å kunne søke om opptak til forkortet ph.d. etter endt medisinstudium. Ledelsen i forskerlinjen er stabil, og vi jobber godt sammen med studentene. Studentene kommer til oss med små og store problem, og vår dør er åpen. Vi har god økonomisk styring. Vi griper inn når studenter sliter med motivasjon eller har manglende progresjon. I så måte er de årlige samtalene med hver enkelt student viktige. I disse samtalene bidrar vi også med karrieretips, noe vi har inntrykk av studentene setter pris på. Vi oppfordrer de studentene som ønsker å fortsette med forskning til å søke øremerkede stipend etter endt studium. Det er et viktig rekrutteringstiltak at øremerkede stipend opprettholdes. Gode muligheter for ph.d.-stipend etter endt studium er også en god måte å rekruttere dyktige, forskningsinteresserte studenter til forskerlinjen. Vi tror at mange av forskerlinjestudentene som har disputert ønsker å jobbe klinisk i tillegg til en UiB-stilling som førsteamanuensis.

Forskerlinjen i Bergen har kapasitet til 85 studenter (10 i odontologi og 75 i medisin) fordelt på fem kull. Konkurransen om studentenes oppmerksomhet er stor, og det må foreligge gode incentiver for at et tilstrekkelig antall studenter skal finne Forskerlinjen attraktiv. I 2023 har oppmøtet på rekrutteringsmøtene vært varierende, og det var kun 8 søkere til 17 plasser. Til gjengjeld var søkerne svært motiverte med gode og avgrensede prosjekter, noe som også gjenspeiler at veilederne er engasjerte og dedikerte. De siste årene har det vært god tilgang på forskningsprosjekter tilpasset disse yngste forskerne på fakultetet. Dedikasjon og systematisk arbeid gjør at disse ungdommene i framtiden kan bli gode forskningsledere.

DET MEDISINSKE FAKULTET

PROGRAMUTVALGET FOR FORSKERUTDANNING

MØTE 17.04.2024

SAK 10/24

Søknad om godkjenning av individuelt lesepensum

Hva saken gjelder

Søknaden om godkjenning av individuelt lesepensum for en ph.d.-kandidat ble behandlet i PFU på møtet 22.11.2023 (sak 36/23), men ble ikke godkjent som en del av kandidatens opplæringsdel.

Nå søker kandidaten på nytt om godkjenning av individuelt lesepensum og har vedlagt følgende informasjon:

- Innholdet: En oversikt over hva lesepensumet inneholder.
- Læringsutbytte: Hva kandidaten forventer å lære gjennom dette spesialpensumet.
- Formatet på arbeidet med spesialpensumet
- Vurderingsformen: Dette inkluderer en skriftlig oppgave

Det søkes om godkjenning med 3 studiepoeng.

Søknaden anbefales av hovedveileder og utdanningsleder ved instituttet.

Forslag til vedtak

1. Programutvalget godkjenner søknaden om individuelt pensum som del av kandidatens opplæringsdel. Studiepoeng angis dersom denne fraviker foreslått studiepoengtildeling.
2. Programutvalget godkjenner ikke søknaden om individuelt pensum som en del av kandidatens opplæringsdel. Avslag begrunnes.

Vedlegg

- Søknad om godkjenning av individuelt lesepensum
- Beskrivelse og innhold av spesialpensumet
- Anbefaling ved instituttet

HAJA/27.02.2024

DET MEDISINSKE FAKULTET

PROGRAMUTVALGET FOR FORSKERUTDANNING

MØTE 17.04.2024

SAK 09/24

Årsmelding Forskerlinjen 2023

Hva saken gjelder

Forskerlinjeadministrasjonen har også i år utarbeidet Årsmelding for Forskerlinjen. Årsmeldingen tar for seg opptak, frafall, uteksaminerte studenter, aktiviteter, og vedtak med betydning for Forskerlinjen.

- Høsten 2023 fullførte den 212. studenten Forskerlinjen.
- Forskerlinjen tok opp 7 nye studenter i 2023.
- I 2023 var det to studenter som sluttet
- Opptak, undervisning og drift av Forskerlinjen er gjennomført i tråd med planer og vedtak.

Til Forskerlinjen er knyttet ett professorat 60 % stilling, en professor II i 20 % stilling, en 1. amanuensis II i 20% stilling og en rådgiver i 50 % stilling.

Forslag til vedtak

Programutvalg for forskerutdanning tar årsmeldingen til etterretning.

Vedlegg

- Årsmelding for Forskerlinjen 2023

MARSTI/03/24

REQUEST FOR APPROVAL OF AN INDIVIDUAL CURRICULUM FOR A PHD STUDENT DUE TO EXCEPTIONAL CIRCUMSTANCES

PhD student Alberto Rovetta has limited chemoinformatics and relational databases experience. These skills are essential to successfully complete his PhD project, but, unfortunately, there is no course that combines all these topics either at UiB or at other universities.

According to the UiB training component regulations, "the training component should include training in scientific dissemination, theory of science and ethics, as well as training in topics relevant to the candidate's project". This syllabus has therefore been created to specifically address a topic relevant to the candidate's project: to build the theoretical and practical knowledge required to work with molecules in Python and to perform relational databases searches, which are highly relevant and required in Alberto's project. The syllabus includes two IBM courses (see below), that are aimed at beginners. Considering that Alberto did not have any programming courses in his previous degree, we believe that they are at the right level for a PhD course for him to set the stage, considering the interdisciplinary nature of his PhD project. Combined with the RDKit elements of the course (see below) this special syllabus goes clearly beyond about what is taught in lower-level courses and addresses the knowledge needs that he has for his PhD project. To start with the basics and then to proceed to more advanced levels is also common with other programming courses offered to interdisciplinary PhD students in Norway, e. g. the BioCat course "Data visualization with Python programming software", or "BIO-8027 Scientific Programming with Python in the life sciences" offered at UiT.

I therefore strongly recommend that PhD student Alberto Rovetta receives approval from the program committee to accept this special syllabus so that he can learn the required skills.

Below is a description of the special curriculum content and learning outcomes.

INDUVIDUAL SYLLABUS FOR PHD STUDENT ALBERTO ROVETTA

Given the scope of this syllabus, the number of ECTS that can be earned in this course is set to 3, calculated on 25 h/ECTS.

Objective:

The goal of this course is to introduce MySQL databases and how to work with molecules in Python using RDKit, an Open Source Chemoinformatics Software, which can be used to perform a very wide range of chemoinformatics tasks.

Content:

The syllabus covers the most important topics in chemoinformatics (types of molecular files, working with molecules, substructure searching, chemical transformations, maximum common substructure, fingerprints and molecular similarity) and relational databases (analyze data within a database using SQL and Python, create a relational database on Cloud and using DDL commands, work with tables write SQL statements including SELECT, INSERT, UPDATE, and DELETE, and using DDL commands, construct basic to intermediate level SQL queries using DML commands, compose more powerful queries with advanced SQL techniques like views, transactions, stored procedures, and joins).

Learning Outcomes:

Upon successful completion of the course, the student will possess the following learning outcomes, defined in terms of knowledge and skills.

► **Knowledge:**

The student will learn the principles of how relational databases and RDKit Python module can be combined and used to solve computational chemoinformatics problems.

▸ **Skills**:

- implement python scripts to perform different analyses on sets of molecules.
- use available program libraries with a special focus on the RDKit library.
- extend and adapt code written by other programmers to specific need.
- create, modify and use relational databases to store and retrieve chemoinformatics data.

Forms of work and teaching:
The course is composed of two modules:
1. RDKit – an Open-Source toolkit for cheminformatics (1.5 ECTS)
Within this module, the student reads a comprehensive introduction to RDKit which will allow him to learn the most used functionalities of the package (see "Introduction to RDKit with Python" and "RDKit tutorials and video lectures"). This will be underpinned by studying selected video lectures. In addition, a list of tutorials will be provided, and the student will work through these.

2. SQL databases for data science with Python (1.5 ECTS)
The student will be asked to complete two online courses provided by IBM on Coursera. Specifically:

- SQL: A Practical Introduction for Querying Databases
- Databases and SQL for Data Science with Python

These courses emphasize hands-on practical learning, each section starts with a theoretical part composed of video lectures and reading material. Then the student works with real database systems, uses real tools, and real-world datasets, as well as creates a database instance in the cloud. Practice on building and running SQL queries is done through hands-on labs. The acquired skills are evaluated at the end of each section through specific quizzes.
At course end, the acquired skills will be applied and demonstrated in a final project. The SQL skills learnt in this course can be applicable to a variety of RDBMSes such as MySQL, PostgreSQL, IBM Db2, Oracle, SQL Server and others. For the detailed content of each course section see "SQL courses details".

Form of assessment:
The course concludes with a written examination.
The student will be asked to write a program that performs specific tasks related to knowledge acquired during the course. The program must be well documented and executable.

Grading Scale: Pass/Fail

Bergen, 31.01.24

Prof. Dr. Ruth Brenk
University of Bergen
Department of Biomedicine

Introduction to RDKit with Python

Introduction

RDKit is a OpenSource toolkit extensively used in chemoinformatics as allows to perform an large variety of operation on molecules.
This document has been written based on Getting Started with the RDKit with Python webpage to give an introduction to the most important functionalities implemented in RDKit, such as types of molecular files, working with molecules, substructure searching, chemical transformations, maximum common substructure, fingerprints and molecular similarity, and other relevant topics.

Table of content

1. **Reading, drawing and writing molecules**: reading single molecules, drawing and saving molecule images, reading sets of molecules, writing molecules, writing sets of molecules

2. **Working with molecules**: looping over atoms and bonds, ring information, modifying molecules, working with 2D molecules, working with 3D molecules, preserving molecules, drawing molecules, metadata in molecules images

3. **Substructure searching**: stereochemistry in substructure matches, atom map indices in SMARTS, advanced substructure matching

4. **Chemical transformations**: substructure-based transformations, Murcko decomposition
5. **Maximum common substructure**: FindMCS, RascalMCES, clustering with Rascal
6. **Fingerprint and molecular similarity**: RDKit (topological) fingerprints, atom pairs and topological torsions, Morgan fingerprints, MACCS keys, explaining bits from fingerprints (Morgan and RDKit fingerprints), generating images of fingerprint bits, picking diverse molecules using fingerprints, generating similarity maps using fingerprints
7. **Descriptor calculation**: calculating all descriptors, calculating partial charges, visualization of descriptors
8. **Chemical reactions**: drawing chemical reactions, advanced reaction functionality (protecting atoms), Recap implementations, BRICS implementation, other fragmentation approaches
9. **Chemical features and pharmacophores**: chemical features, 2D pharmacophore fingerprints
10. **Molecular fragments**
11. **R-Group Decomposition**
12. **Non-chemical functionality**: bit vectors
13. **Getting help**
14. **Advanced topics/warnings**: editing molecules
15. **Miscellaneous tips and hints**: Chem vs AllChem, the SSSR problem
16. **List of available descriptors**
17. **List of available 3D descriptors**

Important notes

This document is based on available information on RDKit webpage, and referes to RDKit 2023.09 release.

Beginning with the 2019.03 release, RDKit is no longer supporting Python 2. If you need to continue using Python 2, please stick with a release from the 2018.09 release cycle.

This document is intended to provide an overview of how one can use RDKit functionality from Python.

Reading, drawing and writing molecules

Reading single molecules

The majority of the basic molecular functionality is found in module rdkit.Chem.

Individual molecules can be contructed using a variety of approaches, depending on the input format:

```
# First import RDKit.Chem module
>>> from rdkit import Chem
# Different ways of creating molecules
>>> m = Chem.MolFromSmiles('Cc1ccccc1')
>>> m = Chem.MolFromMolFile('data/input.mol')
>>> stringWithMolData=open('data/input.mol','r').read()
>>> m = Chem.MolFromMolBlock(stringWithMolData)
```

All these functions return a rdkit.Chem.rdchem.Mol object on success, or None on failure:

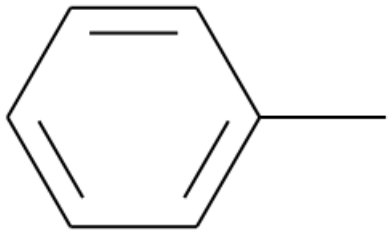
```
>>> m
<rdkit.Chem.rdchem.Mol object at 0x...>


>>> m_invalid = Chem.MolFromMolFile('data/invalid.mol')
>>> m_invalid
None
```

Drawing and saving molecule images

Molecules can be displayed or saved using rdkit.Chem.Draw. With Draw.MolToImage() it is possible to display Mol objects, whereas images can be saved using Draw.MolToFile() function.

```
>>> from rdkit.Chem import Draw
# Display mol object
```

```
>>> img = Draw.MolToImage(m)
```



```
# Save molecule image
>>> Draw.MolToFile(m, "molecule.png")
```

An attempt is made to provide sensible error messages:

```
>>> m1 = Chem.MolFromSmiles('CO(C)C')
```

[12:18:01] Explicit valence for atom # 1 O greater than permitted

```
>>> m2 = Chem.MolFromSmiles('c1cc1')
```

[12:20:41] Can't kekulize mol

Reading sets of molecules

Groups of molecules are read using a Supplier (for example, an rdkit.Chem.SDMolSupplier or a rdkit.Chem.SmilesMolSupplier):

```
>>> suppl = Chem.SDMolSupplier('data/5ht3ligs.sdf')
>>> for mol in suppl:
...     print(mol.GetNumAtoms())
20
24
24
26
```

You can easily produce lists of molecules from a Supplier:

```
>>> mols = [x for x in suppl]
>>> len(mols)
```

```
4
```

or just treat the Supplier itself as a random-access object:

```
>>> suppl[0].GetNumAtoms()
```

```
20
```

Two good practices when working with Suppliers are to use a context manager and to test each molecule to see if it was correctly read before working with it:

```
>>> with Chem.SDMolSupplier('data/5ht3ligs.sdf') as suppl:
...     for mol in suppl:
...         if mol is None: continue
...         print(mol.GetNumAtoms())
20
24
24
26
```

An alternate type of Supplier, the rdkit.Chem.ForwardSDMolSupplier can be used to read from file-like objects:

```
>>> inf = open('data/5ht3ligs.sdf','rb')
>>> with Chem.ForwardSDMolSupplier(inf) as fsuppl:
...     for mol in fsuppl:
...        if mol is None: continue
...        print(mol.GetNumAtoms())
20
24
```

```
24

26
```

This means that they can be used to read from compressed files:

```
>>> import gzip
>>> inf = gzip.open('data/actives_5ht3.sdf.gz')
>>> with Chem.ForwardSDMolSupplier(inf) as gzsuppl:
...     ms = [x for x in gzsuppl if x is not None]
>>> len(ms)
180
```

Note that ForwardSDMolSuppliers cannot be used as random-access objects:

```
>>> inf = open('data/5ht3ligs.sdf','rb')
>>> with Chem.ForwardSDMolSupplier(inf) as fsuppl:
...     fsuppl[0]
```

TypeError: 'ForwardSDMolSupplier' object does not support indexing

For reading Smiles or SDF files with large number of records concurrently, MultithreadedMolSuppliers can be used like this:

```
>>> i = 0
>>> with Chem.MultithreadedSDMolSupplier('data/5ht3ligs.sdf') as sdSupl:
...     for mol in sdSupl:
...         if mol is not None:
...             i += 1
>>> print(i)
4
```

By default a single reader thread is used to extract records from the file and a single writer thread is used to process them. Note that due to multithreading the output may not be in the expected order. Furthermore, the MultithreadedSmilesMolSupplier and the MultithreadedSDMolSupplier cannot be used as random-access objects.

Writing molecules

Single molecules can be converted to text using several functions present in the rdkit.Chem module.

For example, for SMILES:

```
>>> m = Chem.MolFromMolFile('data/chiral.mol')
>>> Chem.MolToSmiles(m)
'C[C@H](O)c1ccccc1'
>>> Chem.MolToSmiles(m,isomericSmiles=False)
'CC(O)c1ccccc1'
```

Note that the SMILES provided is canonical, so the output should be the same no matter how a particular molecule is input:

```
>>> Chem.MolToSmiles(Chem.MolFromSmiles('C1=CC=CN=C1'))
'c1ccncc1'
>>> Chem.MolToSmiles(Chem.MolFromSmiles('c1cccnc1'))
'c1ccncc1'
>>> Chem.MolToSmiles(Chem.MolFromSmiles('n1ccccc1'))
'c1ccncc1'
```

If you'd like to have the Kekule form of the SMILES, first Kekulize the molecule, then use the "kekuleSmiles" option:

```
>>> Chem.Kekulize(m)
>>> Chem.MolToSmiles(m,kekuleSmiles=True)
'C[C@H](O)C1=CC=CC=C1'
```

Note: as of this writing (Aug 2008), the smiles provided when one requests kekuleSmiles are not canonical. The limitation is not in the SMILES generation, but in the kekulization itself.

MDL Mol blocks are also available:

```
>>> m2 = Chem.MolFromSmiles('C1CCC1')
>>> print(Chem.MolToMolBlock(m2))
'''

     RDKit          2D

  4  4  0  0  0  0  0  0  0  0999 V2000
    1.0607    0.0000    0.0000 C   0  0  0  0  0  0  0  0  0  0  0  0
   -0.0000   -1.0607    0.0000 C   0  0  0  0  0  0  0  0  0  0  0  0
   -1.0607    0.0000    0.0000 C   0  0  0  0  0  0  0  0  0  0  0  0
    0.0000    1.0607    0.0000 C   0  0  0  0  0  0  0  0  0  0  0  0
  1  2  1  0
  2  3  1  0
  3  4  1  0
  4  1  1  0
M  END
'''
```

To include names in the mol blocks, set the molecule's "_Name" property:

```
m2.SetProp("_Name","cyclobutane")
print(Chem.MolToMolBlock(m2))
'''
cyclobutane
     RDKit          2D

  4  4  0  0  0  0  0  0  0  0999 V2000
    1.0607    0.0000    0.0000 C   0  0  0  0  0  0  0  0  0  0  0  0
   -0.0000   -1.0607    0.0000 C   0  0  0  0  0  0  0  0  0  0  0  0
   -1.0607    0.0000    0.0000 C   0  0  0  0  0  0  0  0  0  0  0  0
    0.0000    1.0607    0.0000 C   0  0  0  0  0  0  0  0  0  0  0  0
```

```
  1  2  1  0
  2  3  1  0
  3  4  1  0
  4  1  1  0
M  END
'''
```

In order for atom or bond stereochemistry to be recognised correctly by most software, it's essential that the mol block have atomic coordinates. It's also convenient for many reasons, such as drawing the molecules.

Generating a mol block for a molecule that does not have coordinates will, by default, automatically cause coordinates to be generated. These are not, however, stored with the molecule. Coordinates can be generated and stored with the molecule using functionality in the rdkit.Chem.AllChem module (see the Chem vs AllChem section for more information). You can either include 2D coordinates (i.e. a depiction):

```
>>> from rdkit.Chem import AllChem
>>> AllChem.Compute2DCoords(m2)
```

0

```
>>> print(Chem.MolToMolBlock(m2))
'''
cyclobutane
     RDKit          2D

  4  4  0  0  0  0  0  0  0  0999 V2000
    1.0607   -0.0000    0.0000 C   0  0  0  0  0  0  0  0  0  0  0  0
   -0.0000   -1.0607    0.0000 C   0  0  0  0  0  0  0  0  0  0  0  0
   -1.0607    0.0000    0.0000 C   0  0  0  0  0  0  0  0  0  0  0  0
    0.0000    1.0607    0.0000 C   0  0  0  0  0  0  0  0  0  0  0  0
  1  2  1  0
```

```
  2  3  1  0
  3  4  1  0
  4  1  1  0
M  END
'''
```

Or you can add 3D coordinates by embedding the molecule (this uses the ETKDG method, which is described in more detail below). Note that we add Hs to the molecule before generating the conformer. This is essential to get good structures:

```
>>> m3 = Chem.AddHs(m2)
>>> AllChem.EmbedMolecule(m3,randomSeed=0xf00d)
# optional random seed for reproducibility
```

0

```
>>> print(Chem.MolToMolBlock(m3))
'''
cyclobutane
     RDKit          3D

 12 12  0  0  0  0  0  0  0  0999 V2000
    1.0256    0.2491   -0.0964 C   0  0  0  0  0  0  0  0  0  0  0  0
   -0.2041    0.9236    0.4320 C   0  0  0  0  0  0  0  0  0  0  0  0
   -1.0435   -0.2466   -0.0266 C   0  0  0  0  0  0  0  0  0  0  0  0
    0.2104   -0.9922   -0.3417 C   0  0  0  0  0  0  0  0  0  0  0  0
    1.4182    0.7667   -0.9782 H   0  0  0  0  0  0  0  0  0  0  0  0
    1.8181    0.1486    0.6820 H   0  0  0  0  0  0  0  0  0  0  0  0
   -0.1697    1.0826    1.5236 H   0  0  0  0  0  0  0  0  0  0  0  0
   -0.5336    1.8391   -0.1051 H   0  0  0  0  0  0  0  0  0  0  0  0
   -1.6809   -0.0600   -0.8987 H   0  0  0  0  0  0  0  0  0  0  0  0
   -1.6501   -0.6194    0.8220 H   0  0  0  0  0  0  0  0  0  0  0  0
```

```
    0.4659   -1.7768    0.3858 H   0  0  0  0  0  0  0  0  0  0  0  0
    0.3439   -1.3147   -1.3988 H   0  0  0  0  0  0  0  0  0  0  0  0
  1  2  1  0
  2  3  1  0
  3  4  1  0
  4  1  1  0
  1  5  1  0
  1  6  1  0
  2  7  1  0
  2  8  1  0
  3  9  1  0
  3 10  1  0
  4 11  1  0
  4 12  1  0
M  END
'''
```

If we don't want the Hs in our later analysis, they are easy to remove:

```
>>> m3 = Chem.RemoveHs(m3)
>>> print(Chem.MolToMolBlock(m3))
'''
cyclobutane
     RDKit          3D

  4  4  0  0  0  0  0  0  0  0999 V2000
    1.0256    0.2491   -0.0964 C   0  0  0  0  0  0  0  0  0  0  0  0
   -0.2041    0.9236    0.4320 C   0  0  0  0  0  0  0  0  0  0  0  0
   -1.0435   -0.2466   -0.0266 C   0  0  0  0  0  0  0  0  0  0  0  0
    0.2104   -0.9922   -0.3417 C   0  0  0  0  0  0  0  0  0  0  0  0
  1  2  1  0
```

```
  2  3  1  0
  3  4  1  0
  4  1  1  0
M  END
'''
```

If you'd like to write the molecule to a file, use Python file objects:

```
>>> print(Chem.MolToMolBlock(m2),file=open('data/foo.mol','w+'))
```

Writing sets of molecules

Multiple molecules can be written to a file using an rdkit.Chem.rdmolfiles.SDWriter object:

```
>>> with Chem.SDWriter('data/foo.sdf') as w:
...     for m in mols:
...         w.write(m)
```

An SDWriter can also be initialized using a file-like object:

```
>>> from rdkit.six import StringIO
>>> sio = StringIO()
>>> with Chem.SDWriter(sio) as w:
...     for m in mols:
...         w.write(m)
>>> print(sio.getvalue())
'''
mol-295
     RDKit          3D

 20 22  0  0  1  0  0  0  0  0999 V2000
    2.3200    0.0800   -0.1000 C   0  0  0  0  0  0  0  0  0  0  0  0
```

```
    1.8400   -1.2200    0.1200 C   0  0  0  0  0  0  0  0  0  0  0  0
...
  1  3  1  0
  1  4  1  0
  2  5  1  0
M  END
'''
```

Other available Writers include:

- rdkit.Chem.rdmolfiles.SmilesWriter
- rdkit.Chem.rdmolfiles.TDTWriter.

Working with molecules

Looping over Atoms and Bonds

Once you have a molecule, it's easy to loop over its atoms and bonds:

```
>>> m = Chem.MolFromSmiles('C1OC1')
>>> for atom in m.GetAtoms():
...     print(atom.GetAtomicNum())
6
8
6
>>> print(m.GetBonds()[0].GetBondType())
SINGLE
```

You can also request individual bonds or atoms:

```
>>> m.GetAtomWithIdx(0).GetSymbol()
'C'
>>> m.GetAtomWithIdx(0).GetExplicitValence()
2
>>> m.GetBondWithIdx(0).GetBeginAtomIdx()
0
>>> m.GetBondWithIdx(0).GetEndAtomIdx()
1
>>> m.GetBondBetweenAtoms(0,1).GetBondType()
rdkit.Chem.rdchem.BondType.SINGLE
```

Atoms keep track of their neighbors:

```
>>> atom = m.GetAtomWithIdx(0)
>>> [x.GetAtomicNum() for x in atom.GetNeighbors()]
```

```
[8,6]
>>> len(atom.GetNeighbors()[-1].GetBonds())
2
```

Ring Information

Atoms and bonds both carry information about the molecule's rings:

```
>>> m = Chem.MolFromSmiles('OC1C2C1CC2')
>>> m.GetAtomWithIdx(0).IsInRing()
False
>>> m.GetAtomWithIdx(1).IsInRing()
True
>>> m.GetAtomWithIdx(2).IsInRingSize(3)
True
>>> m.GetAtomWithIdx(2).IsInRingSize(4)
True
>>> m.GetAtomWithIdx(2).IsInRingSize(5)
False
>>> m.GetBondWithIdx(1).IsInRingSize(3)
True
>>> m.GetBondWithIdx(1).IsInRing()
True
```

But note that the information is only about the smallest rings:

```
>>> m.GetAtomWithIdx(1).IsInRingSize(5)
False
```

More detail about the smallest set of smallest rings (SSSR) is available:

```
>>> ssr = Chem.GetSymmSSSR(m)
>>> len(ssr)
```

```
2
>>> list(ssr[0])
[1, 2, 3]
>>> list(ssr[1])
[4, 5, 2, 3]
```

As the name indicates, this is a symmetrized SSSR; if you are interested in the number of "true" SSSR, use the GetSSSR function (note that in this case there's no difference).

```
>>> len(Chem.GetSSSR(m))
2
```

The distinction between symmetrized and non-symmetrized SSSR is discussed in more detail below in the section The SSSR Problem. For more efficient queries about a molecule's ring systems (avoiding repeated calls to Mol.GetAtomWithIdx), use rdkit.Chem.rdchem.RingInfo:

```
>>> m = Chem.MolFromSmiles('OC1C2C1CC2')
>>> ri = m.GetRingInfo()
>>> ri.NumAtomRings(0)
0
>>> ri.NumAtomRings(1)
1
>>> ri.NumAtomRings(2)
2
>>> ri.IsAtomInRingOfSize(1,3)
True
>>> ri.IsBondInRingOfSize(1,3)
True
```

Modifying molecules

Normally molecules are stored in the RDKit with the hydrogen atoms implicit (e.g. not explicitly present in the molecular graph. When it is useful to have the hydrogens explicitly present, for example when generating or optimizing the 3D geometry, the rdkit.Chem.AddHs function can be used:

```
>>> m=Chem.MolFromSmiles('CCO')
>>> m.GetNumAtoms()
3
>>> m2 = Chem.AddHs(m)
>>> m2.GetNumAtoms()
9
```

The Hs can be removed again using the rdkit.Chem.RemoveHs() function:

```
>>> m3 = Chem.RemoveHs(m2)
>>> m3.GetNumAtoms()
3
```

RDKit molecules are usually stored with the bonds in aromatic rings having aromatic bond types. This can be changed with the rdkit.Chem.rdmolops.Kekulize() function:

```
>>> m = Chem.MolFromSmiles('c1ccccc1')
>>> m.GetBondWithIdx(0).GetBondType()
rdkit.Chem.rdchem.BondType.AROMATIC
>>> Chem.Kekulize(m)
>>> m.GetBondWithIdx(0).GetBondType()
rdkit.Chem.rdchem.BondType.DOUBLE
>>> m.GetBondWithIdx(1).GetBondType()
rdkit.Chem.rdchem.BondType.SINGLE
```

By default, the bonds are still marked as being aromatic:

```
>>> m.GetBondWithIdx(1).GetIsAromatic()
True
```

because the flags in the original molecule are not cleared (clearAromaticFlags defaults to False). You can explicitly force or decline a clearing of the flags:

```
>>> m = Chem.MolFromSmiles('c1ccccc1')
>>> m.GetBondWithIdx(0).GetIsAromatic()
True
>>> m1 = Chem.MolFromSmiles('c1ccccc1')
>>> Chem.Kekulize(m1, clearAromaticFlags=True)
>>> m1.GetBondWithIdx(0).GetIsAromatic()
False
```

Bonds can be restored to the aromatic bond type using the rdkit.Chem.rdmolops.SanitizeMol() function:

```
>>> Chem.SanitizeMol(m)
rdkit.Chem.rdmolops.SanitizeFlags.SANITIZE_NONE
>>> m.GetBondWithIdx(0).GetBondType()
rdkit.Chem.rdchem.BondType.AROMATIC
```

The value returned by *SanitizeMol()* indicates that no problems were encountered.

Working with 2D molecules: Generating Depictions

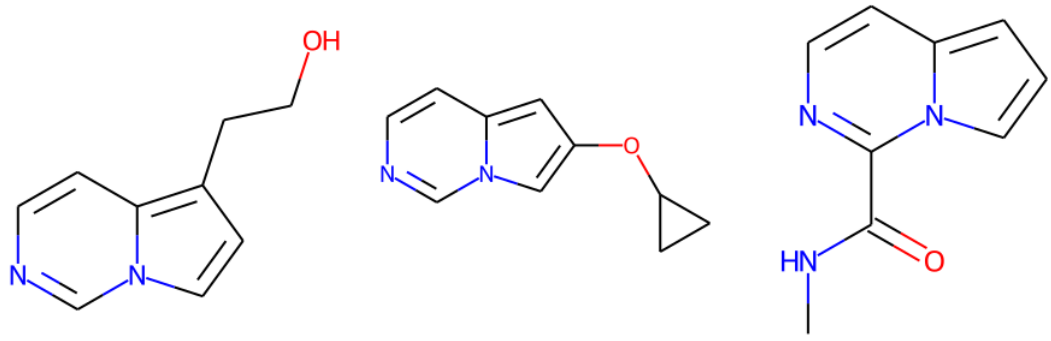
The RDKit has a library for generating depictions (sets of 2D) coordinates for molecules. This library, which is part of the AllChem module, is accessed using the Chem.Compute2DCoords() function:

```
>>> m = Chem.MolFromSmiles('c1nccc2n1ccc2')
>>> AllChem.Compute2DCoords(m)
0
```

The 2D conformer is constructed in a canonical orientation and is built to minimize intramolecular clashes, i.e. to maximize the clarity of the drawing. If you have a set of molecules that share a common template and you'd like to align them to that template, you can do so as follows:

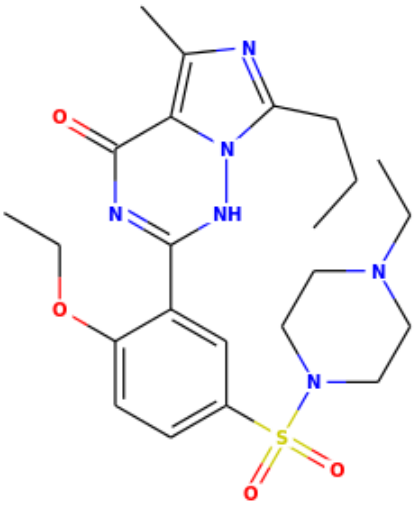
```
>>> template = Chem.MolFromSmiles('c1nccc2n1ccc2')
>>> AllChem.Compute2DCoords(template)
0
>>> ms = [Chem.MolFromSmiles(smi) for smi in ('OCCc1ccn2cnccc12',
                                              'C1CC1Oc1cc2ccncn2c1',
                                              'CNC(=O)c1nccc2cccn12')
          ]
>>> for m in ms:
...     _ = AllChem.GenerateDepictionMatching2DStructure(m,template)
```

Running this process for the molecules above gives:



Another option for Compute2DCoords allows you to generate 2D depictions for molecules that closely mimic 3D conformers. This is available using the function GenerateDepictionMatching3DStructure() available in Chem.AllChem pacakage.

Here is an illustration of the results using the ligand from PDB structure 1XP0:



Using the core function Chem.Compute2DCoordsMimicDistmat() a more fine-grained control can be obtained, but that is beyond the scope of this document. See the implementation of GenerateDepictionMatching3DStructure in AllChem.py for an example of how it is used.

Working with 3D Molecules

The RDKit can generate conformers for molecules using two different methods. The original method used distance geometry. The algorithm followed is:

1. The molecule's distance bounds matrix is calculated based on the connection table and a set of rules.

2. The bounds matrix is smoothed using a triangle-bounds smoothing algorithm.

3. A random distance matrix that satisfies the bounds matrix is generated.

4. This distance matrix is embedded in 3D dimensions (producing coordinates for each atom).

5. The resulting coordinates are cleaned up somewhat using a crude force field and the bounds matrix.

Note that the conformers that result from this procedure tend to be fairly ugly. They should be cleaned up using a force field. This can be done within the RDKit using its implementation of the Universal Force Field (UFF). [2] More recently, there is an implementation of the ETKDG method of Riniker and Landrum [18] which uses torsion angle preferences from the Cambridge Structural Database (CSD) to correct the conformers after distance geometry has been used to generate them. With this method, there should be no need to use a minimisation step to clean up the structures. More detailed information about the conformer generator and the parameters controlling it can be found in the "RDKit Book". Since the 2018.09 release of the RDKit, ETKDG is the default conformer generation method. The full process of embedding a molecule is easier than all the above verbiage makes it sound:

```
>>> m2=Chem.AddHs(m)
>>> AllChem.EmbedMolecule(m2)
0
```

The RDKit also has an implementation of the MMFF94 force field available. Please note that the MMFF atom typing code uses its own aromaticity model, so the aromaticity flags of the molecule will be modified after calling MMFF-related methods. Here's an example of using MMFF94 to minimize an RDKit-generated conformer:

```
>>> m = Chem.MolFromSmiles('C1CCC1OC')
>>> m2=Chem.AddHs(m)
>>> AllChem.EmbedMolecule(m2)
0
>>> AllChem.MMFFOptimizeMolecule(m2)
0
```

Note the calls to *Chem.AddHs()* in the examples above. By default RDKit molecules do not have H atoms explicitly present in the graph, but they are important for getting realistic geometries, so they generally should be added. They can always be removed afterwards

if necessary with a call to *Chem.RemoveHs()*. With the RDKit, multiple conformers can also be generated using the different embedding methods. In both cases this is simply a matter of running the distance geometry calculation multiple times from different random start points. The option *numConfs* allows the user to set the number of conformers that should be generated. Otherwise the procedures are as before. The conformers so generated can be aligned to each other and the RMS values calculated.

```
>>> m = Chem.MolFromSmiles('C1CCC1OC')
>>> m2=Chem.AddHs(m)
# run ETKDG 10 times
>>> cids = AllChem.EmbedMultipleConfs(m2, numConfs=10)
>>> print(len(cids))
10
>>> rmslist = []
>>> AllChem.AlignMolConformers(m2, RMSlist=rmslist)
>>> print(len(rmslist))
9
```

rmslist contains the RMS values between the first conformer and all others. The RMS between two specific conformers (e.g. 1 and 9) can also be calculated. The flag prealigned lets the user specify if the conformers are already aligned (by default, the function aligns them).

```
>>> rms = AllChem.GetConformerRMS(m2, 1, 9, prealigned=True)
```

If you are interested in running MMFF94 on a molecule's conformers (note that this is often not necessary when using ETKDG), there's a convenience function available:

```
>>> res = AllChem.MMFFOptimizeMoleculeConfs(m2)
```

The result is a list a containing 2-tuples: *(not_converged, energy)* for each conformer. If not_converged is 0, the minimization for that conformer converged.

By default *AllChem.EmbedMultipleConfs()* and *AllChem.MMFFOptimizeMoleculeConfs()* run single threaded, but you can cause them to use multiple threads simultaneously for these embarassingly parallel tasks via the *numThreads* argument:

```
>>> cids = AllChem.EmbedMultipleConfs(m2, numThreads=0)
>>> res = AllChem.MMFFOptimizeMoleculeConfs(m2, numThreads=0)
```

Setting *numThreads* to zero causes the software to use the maximum number of threads allowed on your computer. *Disclaimer/Warning*: Conformer generation is a difficult and subtle task. The plain distance-geometry 2D to3D conversion provided with the RDKit is not intended to be a replacement for a "real" conformer analysis tool; it merely provides quick 3D structures for cases when they are required. We believe, however, that the newer ETKDG method is suitable for most purposes.

Preserving Molecules

Molecules can be converted to and from text using Python's pickling machinery:

```
>>> m = Chem.MolFromSmiles('c1ccncc1')
>>> import pickle
>>> pkl = pickle.dumps(m)
>>> m2=pickle.loads(pkl)
>>> Chem.MolToSmiles(m2)
'c1ccncc1'
```

The RDKit pickle format is fairly compact and it is much, much faster to build a molecule from a pickle than from a Mol file or SMILES string, so storing molecules you will be working with repeatedly as pickles can be a good idea. The raw binary data that is encapsulated in a pickle can also be directly obtained from a molecule:

```
>>> binStr = m.ToBinary()
```

This can be used to reconstruct molecules using the Chem.Mol constructor:

```
>>> m2 = Chem.Mol(binStr)
>>> Chem.MolToSmiles(m2)
'c1ccncc1'
>>> len(binStr)
130
```

Note that this is smaller than the pickle:

```
>>> len(binStr) < len(pkl)
True
```

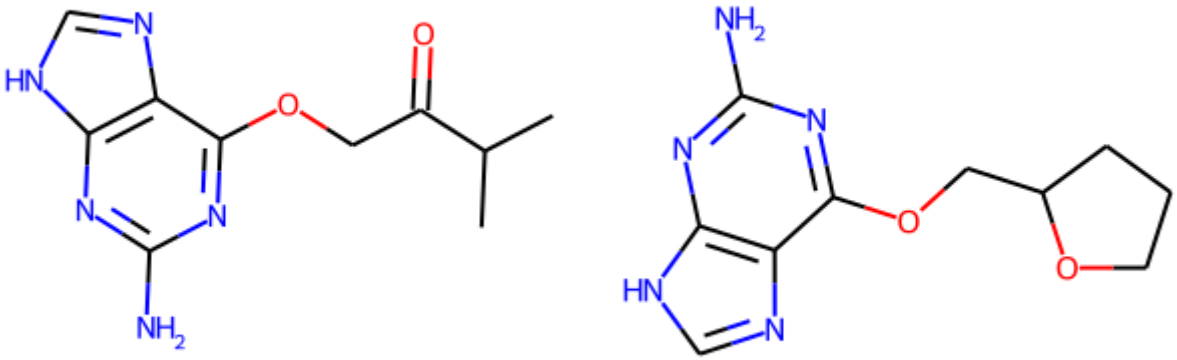
The small overhead associated with python's pickling machinery normally doesn't end up making much of a difference for collections of larger molecules (the extra data associated with the pickle is independent of the size of the molecule, while the binary string increases in length as the molecule gets larger). *Tip*: The performance difference associated with storing molecules in a pickled form on disk instead of constantly reparsing an SD file or SMILES table is difficult to overstate. In a test I just ran on my laptop, loading a set of 699 drug-like molecules from an SD file took 10.8 seconds; loading the same molecules from a pickle file took 0.7 seconds. The pickle file is also smaller – 1/3 the size of the SD file – but this difference is not always so dramatic (it's a particularly fat SD file).

Drawing Molecules

The RDKit has some built-in functionality for creating images from molecules found in the rdkit.Chem.Draw package:

```
>>> with Chem.SDMolSupplier('data/cdk2.sdf') as suppl:
...     ms = [x for x in suppl if x is not None]
>>> for m in ms: tmp=AllChem.Compute2DCoords(m)
>>> from rdkit.Chem import Draw
>>> Draw.MolToFile(ms[0],'images/cdk2_mol1.o.png')
>>> Draw.MolToFile(ms[1],'images/cdk2_mol2.o.png')
```

Producing these images:



(a) CDK2 mol. 1 (b) CDK2 mol. 2

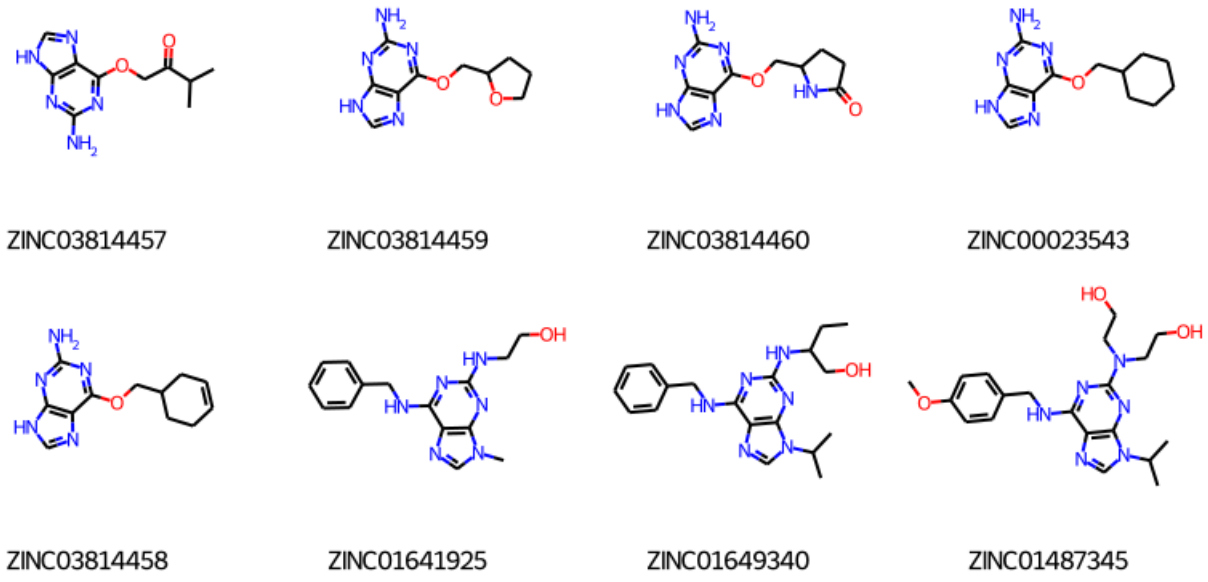
It's also possible to produce an image grid out of a set of molecules:

```
>>> img=Draw.MolsToGridImage(ms[:8],
                             molsPerRow=4,
                             subImgSize=(200,200),
                             legends=[x.GetProp("_Name")
                             for x in ms[:8]]
                             )
```

This returns a PIL image, which can then be saved to a file:

```
>>> img.save('images/cdk2_molgrid.o.png')
```

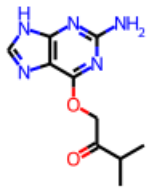
The result looks like this:



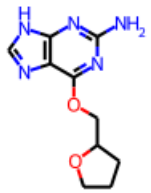
These would of course look better if the common core were aligned. This is easy enough to do:

```
>>> p = Chem.MolFromSmiles('[nH]1cnc2cncnc21')
>>> subms = [x for x in ms if x.HasSubstructMatch(p)]
>>> len(subms)
14
>>> AllChem.Compute2DCoords(p)
0
>>> for m in subms:
...     _ = AllChem.GenerateDepictionMatching2DStructure(m,p)
>>> img=Draw.MolsToGridImage(subms,
                             molsPerRow=4,
                             subImgSize=(200,200),
                             legends=[x.GetProp("_Name")
                             for x in subms])
>>> img.save('images/cdk2_molgrid.aligned.o.png')
```

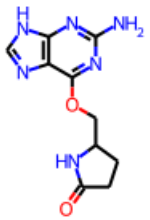
The result looks like this:



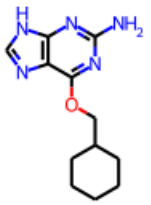
ZINC03814457



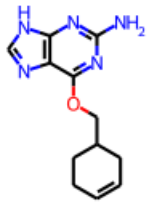
ZINC03814459



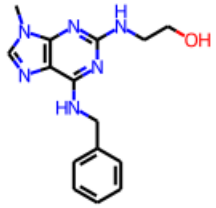
ZINC03814460



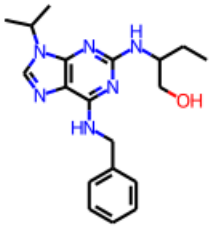
ZINC00023543



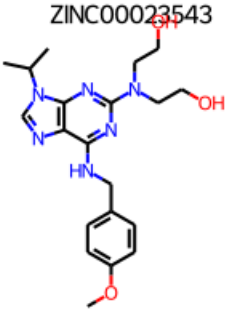
ZINC03814458



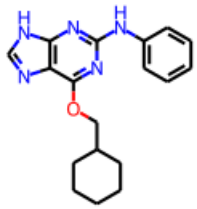
ZINC01641925



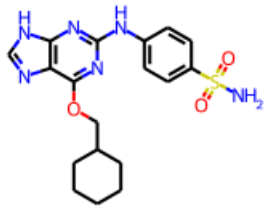
ZINC01649340



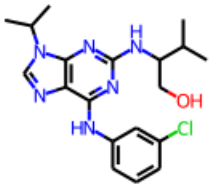
ZINC01487345



ZINC03814462



ZINC00603011



ZINC00582575

ZINC03814437

ZINC03814439

ZINC03591113

Atoms in a molecule can be highlighted by drawing a coloured solid or open circle around them, and bonds likewise can have a coloured outline applied. An obvious use is to show atoms and bonds that have matched a substructure query:

```
>>> from rdkit.Chem.Draw import rdMolDraw2D
>>> smi = 'c1cc(F)ccc1Cl'
>>> mol = Chem.MolFromSmiles(smi)
>>> patt = Chem.MolFromSmarts('ClccccF')
>>> hit_ats = list(mol.GetSubstructMatch(patt))
>>> hit_bonds = []
>>> for bond in patt.GetBonds():
...     aid1 = hit_ats[bond.GetBeginAtomIdx()]
...     aid2 = hit_ats[bond.GetEndAtomIdx()]
...     hit_bonds.append(mol.GetBondBetweenAtoms(aid1,aid2).GetIdx())
>>> d = rdMolDraw2D.MolDraw2DSVG(500, 500) # MolDraw2DCairo to get PNGs
>>> rdMolDraw2D.PrepareAndDrawMolecule(d, mol, highlightAtoms=hit_ats,
                                       highlightBonds=hit_bonds)
```

will produce:

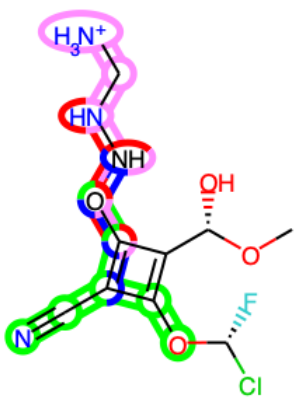
It is possible to specify the colours for individual atoms and bonds:

```
>>> colours = [(0.8,0.0,0.8),(0.8,0.8,0),(0,0.8,0.8),(0,0,0.8)]
>>> atom_cols = {}
>>> for i, at in enumerate(hit_ats):
```

```
...     atom_cols[at] = colours[i%4]
>>> bond_cols = {}
>>> for i, bd in enumerate(hit_bonds):
...     bond_cols[bd] = colours[3 - i%4]
>>> d = rdMolDraw2D.MolDraw2DCairo(500, 500)
>>> rdMolDraw2D.PrepareAndDrawMolecule(d, mol, highlightAtoms=hit_ats,
                                       highlightAtomColors=atom_cols,
                                       highlightBonds=hit_bonds,
                                       highlightBondColors=bond_cols)
```

to give:

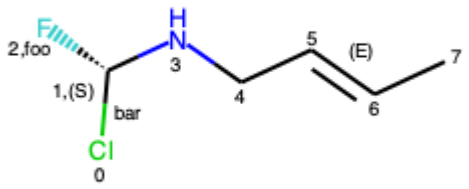
Atoms and bonds can also be highlighted with multiple colours if they fall into multiple sets, for example if they are matched by more than 1 substructure pattern. This is too complicated to show in this simple introduction, but there is an example in data/test_multi_colours.py, which produces the somewhat garish



As of version 2020.03, it is possible to add arbitrary small strings to annotate atoms and bonds in the drawing. The strings are added as properties *atomNote* and *bondNote* and they will be placed automatically close to the atom or bond in question in a manner intended to minimise their clash with the rest of the drawing. For convenience, here are 3 flags in *MolDraw2DOptions* that will add stereo information (R/S to atoms, E/Z to bonds) and atom and bond sequence numbers.

```
>>> mol = Chem.MolFromSmiles(r'Cl[C@H](F)NC\C=C\C')
>>> d = rdMolDraw2D.MolDraw2DCairo(250, 200) # or MolDraw2DSVG to get SVGs
>>> mol.GetAtomWithIdx(2).SetProp('atomNote', 'foo')
>>> mol.GetBondWithIdx(0).SetProp('bondNote', 'bar')
>>> d.drawOptions().addStereoAnnotation = True
>>> d.drawOptions().addAtomIndices = True
>>> d.DrawMolecule(mol)
>>> d.FinishDrawing()
>>> d.WriteDrawingText('atom_annotation_1.png')
```

will produce

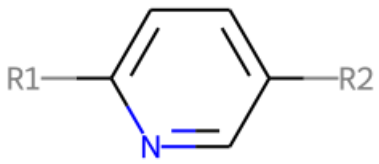


If atoms have an atomLabel property set, this will be used when drawing them:

```
>>> smi = 'c1nc(*)ccc1* |$;;;R1;;;;R2$|'
>>> mol = Chem.MolFromSmiles(smi)
>>> mol.GetAtomWithIdx(3).GetProp("atomLabel")
```

```
'R1'
>>> mol.GetAtomWithIdx(7).GetProp("atomLabel")
'R2'
>>> d = rdMolDraw2D.MolDraw2DCairo(250, 250)
>>> rdMolDraw2D.PrepareAndDrawMolecule(d,mol)
>>> d.WriteDrawingText("./images/atom_labels_1.png")
```

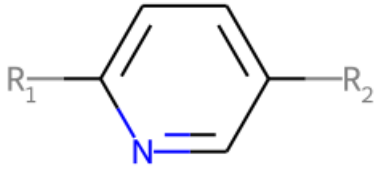
gives:



Since the *atomLabel* property is also used for other things (for example in CXSMILES as demonstrated), if you want to provide your own atom labels, it's better to use the *_displayLabel* property:

```
>>> smi = 'c1nc(*)ccc1* |$;;;R1;;;;R2$|'
>>> mol = Chem.MolFromSmiles(smi)
>>> mol.GetAtomWithIdx(3).SetProp("_displayLabel","R<sub>1</sub>")
>>> mol.GetAtomWithIdx(7).SetProp("_displayLabel","R<sub>2</sub>")
>>> d = rdMolDraw2D.MolDraw2DCairo(250, 250)
>>> rdMolDraw2D.PrepareAndDrawMolecule(d,mol)
>>> d.WriteDrawingText("./images/atom_labels_2.png")
```

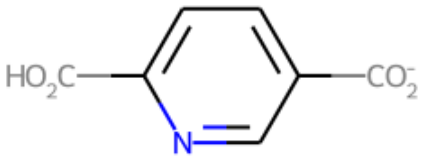
this gives:



Note that you can use $< sup >$ and $< sub >$ in these labels to provide super- and subscripts. Finally, if you have atom labels which should be displayed differently when the bond comes into them from the right (the West), you can also set the *_displayLabelW* property:

```
>>> smi = 'c1nc(*)ccc1* |$;;;R1;;;;R2$|'
>>> mol = Chem.MolFromSmiles(smi)
>>> mol.GetAtomWithIdx(3).SetProp("_displayLabel","CO<sub>2</sub>H")
>>> mol.GetAtomWithIdx(3).SetProp("_displayLabelW","HO<sub>2</sub>C")
>>> mol.GetAtomWithIdx(7).SetProp("_displayLabel","CO<sub>2</sub><sup>-</sup>")
>>> mol.GetAtomWithIdx(7).SetProp("_displayLabelW","<sup>-</sup>OOC")
>>> d = rdMolDraw2D.MolDraw2DCairo(250, 250)
>>> rdMolDraw2D.PrepareAndDrawMolecule(d,mol)
>>> d.WriteDrawingText("./images/atom_labels_3.png")
```

this gives:

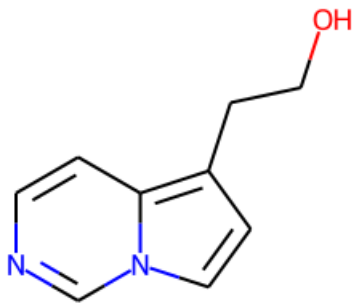


Metadata in Molecule Images

The PNG files generated by the *MolDraw2DCairo* class by default include metadata about the molecule(s) or chemical reaction included in the drawing. This metadata can be used later to reconstruct the molecule(s) or reaction.

```
>>> template = Chem.MolFromSmiles('c1nccc2n1ccc2')
>>> AllChem.Compute2DCoords(template)
0
>>> ms = [Chem.MolFromSmiles(smi) for smi in ('OCCc1ccn2cnccc12',
                                              'C1CC1Oc1cc2ccncn2c1',
                                              'CNC(=O)c1nccc2cccn12')]
>>> _ = [AllChem.GenerateDepictionMatching2DStructure(m,template) for m in ms]
>>> d = rdMolDraw2D.MolDraw2DCairo(250, 200)
>>> d.DrawMolecule(ms[0])
>>> d.FinishDrawing()
>>> png = d.GetDrawingText()
>>> mol = Chem.MolFromPNGString(png)
>>> Chem.MolToSmiles(mol)
'OCCc1c2ccncn2cc1'
```

The molecular metadata is stored using standard metadata tags in the PNG and is, of course, not visible when you look at the PNG:



If the PNG contains multiple molecules we can retrieve them all at once using *MolsFromP-*

NGString() from Chem package:

```
>>> from rdkit.Chem import Draw
>>> png = Draw.MolsToGridImage(ms,returnPNG=True)
>>> mols = Chem.MolsFromPNGString(png)
>>> for mol in mols:
...     print(Chem.MolToSmiles(mol))
'OCCc1c2ccncn2cc1'
'c1cc2cc(OC3CC3)cn2cn1'
'CNC(=O)c1nccc2cccn12'
```

Substructure Searching

Substructure matching can be done using query molecules built from SMARTS:

```
>>> m = Chem.MolFromSmiles('c1ccccc1O')
>>> patt = Chem.MolFromSmarts('ccO')
>>> m.HasSubstructMatch(patt)
True
>>> m.GetSubstructMatch(patt)
(0, 5, 6)
```

Those are the atom indices in *m*, ordered as *patt*'s atoms. To get all of the matches:

```
>>> m.GetSubstructMatches(patt)
((0, 5, 6), (4, 5, 6))
```

This can be used to easily filter lists of molecules:

```
>>> patt = Chem.MolFromSmarts('c[NH1]')
>>> matches = []
>>> with Chem.SDMolSupplier('data/actives_5ht3.sdf') as suppl:
...     for mol in suppl:
...         if mol.HasSubstructMatch(patt):
...             matches.append(mol)
>>> len(matches)
22
```

We can write the same thing more compactly using Python's list comprehension syntax:

```
>>> with Chem.SDMolSupplier('data/actives_5ht3.sdf') as suppl:
...     matches = [x for x in suppl if x.HasSubstructMatch(patt)]
>>> len(matches)
22
```

Substructure matching can also be done using molecules built from SMILES instead of SMARTS:

```
>>> m = Chem.MolFromSmiles('C1=CC=CC=C1OC')
>>> m.HasSubstructMatch(Chem.MolFromSmarts('CO'))
True
>>> m.HasSubstructMatch(Chem.MolFromSmiles('CO'))
True
```

But don't forget that the semantics of the two languages are not exactly equivalent:

```
>>> m.HasSubstructMatch(Chem.MolFromSmiles('COC'))
True
>>> m.HasSubstructMatch(Chem.MolFromSmarts('COC'))
False
>>> m.HasSubstructMatch(Chem.MolFromSmarts('COc')) #<- need an aromatic C
True
```

Stereochemistry in substructure matches

By default information about stereochemistry is not used in substructure searches:

```
>>> m = Chem.MolFromSmiles('CC[C@H](F)Cl')
>>> m.HasSubstructMatch(Chem.MolFromSmiles('C[C@H](F)Cl'))
True
>>> m.HasSubstructMatch(Chem.MolFromSmiles('C[C@@H](F)Cl'))
True
>>> m.HasSubstructMatch(Chem.MolFromSmiles('CC(F)Cl'))
True
```

But this can be changed via the *useChirality* argument:

```
>>> m.HasSubstructMatch(Chem.MolFromSmiles('C[C@H](F)Cl'),
                        useChirality=True)
True
>>> m.HasSubstructMatch(Chem.MolFromSmiles('C[C@@H](F)Cl'),
                        useChirality=True)
False
>>> m.HasSubstructMatch(Chem.MolFromSmiles('CC(F)Cl'),
                        useChirality=True)
True
```

Notice that when *useChirality* is set a non-chiral query **does** match a chiral molecule. The same is not true for a chiral query and a non-chiral molecule:

```
>>> m.HasSubstructMatch(Chem.MolFromSmiles('CC(F)Cl'))
True
>>> m2 = Chem.MolFromSmiles('CCC(F)Cl')
>>> m2.HasSubstructMatch(Chem.MolFromSmiles('C[C@H](F)Cl'),
                         useChirality=True)
False
```

Atom Map Indices in SMARTS

It is possible to attach indices to the atoms in the SMARTS pattern. This is most often done in reaction SMARTS (see Chemical Reactions), but is more general than that. For example, in the SMARTS patterns for torsion angle analysis, indices are used to define the four atoms of the torsion of interest. This allows additional atoms to be used to define the environment of the four torsion atoms, as in *[cH0:1][c:2]([cH0])!@[CX3!r:3]=[NX2!r:4]* for an aromatic C=N torsion. We might wonder in passing why they didn't use recursive SMARTS for this, which would have made life easier, but it is what it is. The atom lists

from *GetSubstructureMatches* are guaranteed to be in order of the SMARTS, but in this case we'll get five atoms so we need a way of picking out, in the correct order, the four of interest. When the SMARTS is parsed, the relevant atoms are assigned an atom map number property that we can easily extract:

```
>>> qmol = Chem.MolFromSmarts('[cH0:1][c:2]([cH0])!@[CX3!r:3]=[NX2!r:4]')
>>> ind_map = {}
>>> for atom in qmol.GetAtoms() :
...     map_num = atom.GetAtomMapNum()
...     if map_num:
...         ind_map[map_num-1] = atom.GetIdx()
>>> ind_map
{0: 0, 1: 1, 2: 3, 3: 4}
>>> map_list = [ind_map[x] for x in sorted(ind_map)]
>>> map_list
[0, 1, 3, 4]
```

Then, when using the query on a molecule you can get the indices of the four matching atoms like this:

```
>>> mol = Chem.MolFromSmiles('Cc1cccc(C)c1C(C)=NC')
>>> for match in mol.GetSubstructMatches( qmol ) :
...     mas = [match[x] for x in map_list]
...     print(mas)
[1, 7, 8, 10]
```

Advanced substructure matching

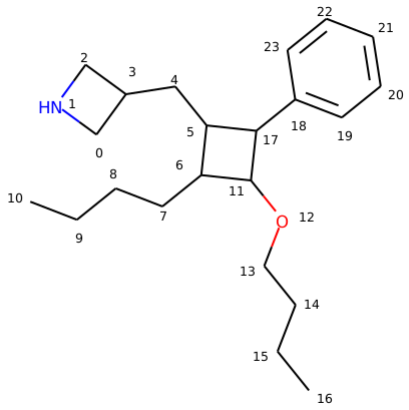
Starting with the 2020.03 release, the RDKit allows you to provide an optional function that is used to check whether or not a possible substructure match should be accepted. This function is called with the molecule to be matched and the indices of the matching

atoms. Here's an example of how you can use the functionality to do "Markush-like" matching, requiring that all atoms in a sidechain are either carbon (type "all_carbon") or aren't aromatic (type "alkyl"). We start by defining the class that we'll use to test the sidechains:

```
from rdkit import Chem
class SidechainChecker(object):
    matchers = {
                'alkyl': lambda at: not at.GetIsAromatic(),
                'all_carbon': lambda at: at.GetAtomicNum() == 6
                }
    def __init__(self, query, pName="queryType"):
            # identify the atoms that have the properties we care about
        self._atsToExamine = [(x.GetIdx(), x.GetProp(pName)) for x in
                              query.GetAtoms() if x.HasProp(pName)]
        self._pName = pName
    def __call__(self, mol, vect):
        seen = [0] * mol.GetNumAtoms()
        for idx in vect:
            seen[idx] = 1
        # loop over the atoms we care about:
        for idx, qtyp in self._atsToExamine:
            midx = vect[idx]
            stack = [midx]
            atom = mol.GetAtomWithIdx(midx)
            # now do a breadth-first search from that atom, checking
            # all of its neighbors that aren't in the substructure
            # query:
            stack = [atom]
```

```
            while stack:
                atom = stack.pop(0)
                if not self.matchers[qtyp](atom):
                    return False
                seen[atom.GetIdx()] = 1
                for nbr in atom.GetNeighbors():
                    if not seen[nbr.GetIdx()]:
                        stack.append(nbr)
        return True
```

Here's the molecule we'll use:



And the default behavior:

```
>>> m = Chem.MolFromSmiles('C2NCC2CC1C(CCCC)C(OCCCC)C1c2ccccc2')
>>> p = Chem.MolFromSmarts('C1CCC1*')
>>> p.GetAtomWithIdx(4).SetProp("queryType", "all_carbon")
>>> m.GetSubstructMatches(p)
((5, 6, 11, 17, 18), (5, 17, 11, 6, 7),
 (6, 5, 17, 11, 12), (6, 11, 17, 5, 4))
```

Now let's add the final check to filter the results:

```
>>> params = Chem.SubstructMatchParameters()
>>> checker = SidechainChecker(p)
>>> params.setExtraFinalCheck(checker)
>>> m.GetSubstructMatches(p,params)
((5, 6, 11, 17, 18), (5, 17, 11, 6, 7))
```

Repeat that using the 'alkyl' query:

```
>>> p.GetAtomWithIdx(4).SetProp("queryType", "alkyl")
>>> checker = SidechainChecker(p)
>>> params.setExtraFinalCheck(checker)
>>> m.GetSubstructMatches(p,params)
((5, 17, 11, 6, 7),
 (6, 5, 17, 11, 12),
 (6, 11, 17, 5, 4))
```

Chemical Transformations

The RDKit contains a number of functions for modifying molecules. Note that these transformation functions are intended to provide an easy way to make simple modifications to molecules. For more complex transformations, use the Chemical Reactions functionality.

Substructure-based transformations

There's a variety of functionality for using the RDKit's substructure-matching machinery for doing quick molecular transformations. These transformations include deleting substructures:

```
>>> m = Chem.MolFromSmiles('CC(=O)O')
>>> patt = Chem.MolFromSmarts('C(=O)[OH]')
>>> rm = AllChem.DeleteSubstructs(m,patt)
>>> Chem.MolToSmiles(rm)
'C'
```

replacing substructures:

```
>>> repl = Chem.MolFromSmiles('OC')
>>> patt = Chem.MolFromSmarts('[$(NC(=O))]')
>>> m = Chem.MolFromSmiles('CC(=O)N')
>>> rms = AllChem.ReplaceSubstructs(m,patt,repl)
>>> rms
(<rdkit.Chem.rdchem.Mol object at 0x...>,)
>>> Chem.MolToSmiles(rms[0])
'COC(C)=O'
```

as well as simple SAR-table transformations like removing side chains:

```
>>> m1 = Chem.MolFromSmiles('BrCCc1cncnc1C(=O)O')
>>> core = Chem.MolFromSmiles('c1cncnc1')
```

```
>>> tmp = Chem.ReplaceSidechains(m1,core)
>>> Chem.MolToSmiles(tmp)
'[1*]c1cncnc1[2*]'
```

and removing cores:

```
>>> tmp = Chem.ReplaceCore(m1,core)
>>> Chem.MolToSmiles(tmp)
'[1*]CCBr.[2*]C(=O)O'
```

By default the sidechains are labeled based on the order they are found. They can also be labeled according by the number of that core-atom they're attached to:

```
>>> m1 = Chem.MolFromSmiles('c1c(CCO)ncnc1C(=O)O')
>>> tmp=Chem.ReplaceCore(m1,core,labelByIndex=True)
>>> Chem.MolToSmiles(tmp)
'[1*]CCO.[5*]C(=O)O'
```

rdkit.Chem.rdmolops.ReplaceCore() returns the sidechains in a single molecule. This can be split into separate molecules using rdkit.Chem.rdmolops.GetMolFrags():

```
>>> rs = Chem.GetMolFrags(tmp,asMols=True)
>>> len(rs)
2
>>> Chem.MolToSmiles(rs[0])
'[1*]CCO'
>>> Chem.MolToSmiles(rs[1])
'[5*]C(=O)O'
```

Murcko Decomposition

The RDKit provides standard Murcko-type decomposition of molecules into scaffolds:

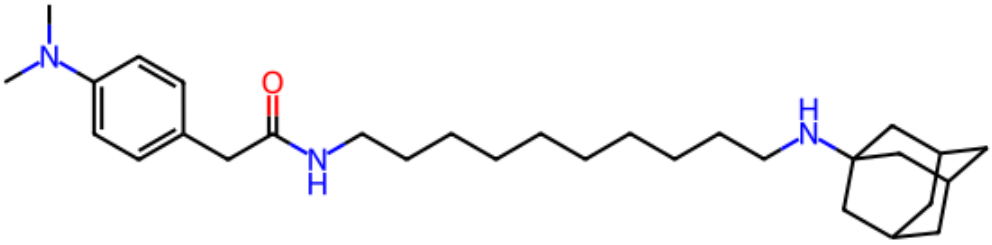
```
>>> from rdkit.Chem.Scaffolds import MurckoScaffold
>>> with Chem.SDMolSupplier('data/cdk2.sdf') as cdk2mols:
...     m1 = cdk2mols[0]
>>> core = MurckoScaffold.GetScaffoldForMol(m1)
>>> Chem.MolToSmiles(core)
'c1ncc2nc[nH]c2n1'
```

or into a generic framework:

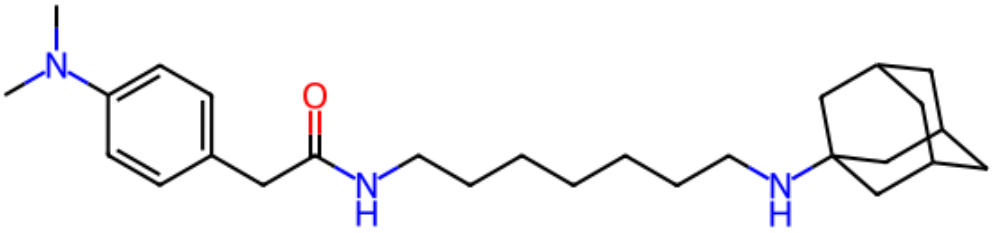
```
>>> fw = MurckoScaffold.MakeScaffoldGeneric(core)
>>> Chem.MolToSmiles(fw)
'C1CCC2CCCC2C1'
```

Maximum Common Substructure

There are 2 methods for finding maximum common substructures. The first, FindMCS, finds a single fragment maximum common substructure (MCS) of two or more molecules: The second, RascalMCES, finds the maximum common edge substructure (MCES) between two molecules and can return a multi-fragment MCES. The difference is demonstrated with the following pair of molecules:



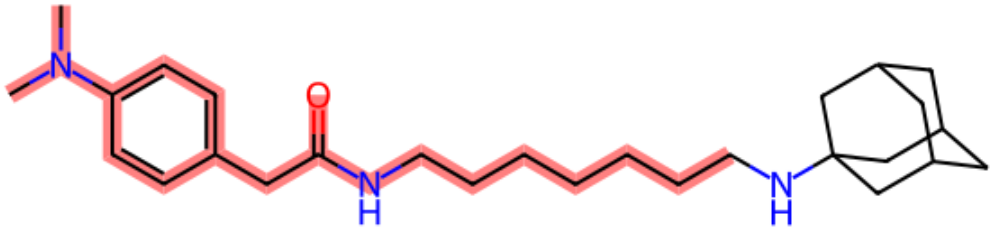
CHEMBL153934



CHEMBL152361

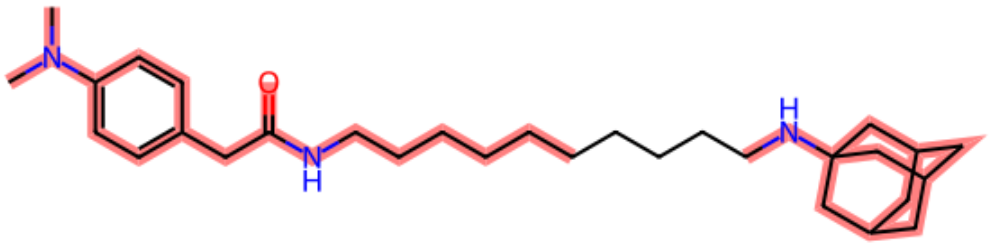
FMCS gives this maximum common substructure:

CHEMBL153934



CHEMBL152361

Whereas RascalMCES gives:



CHEMBL153934

CHEMBL152361

FindMCS

FindMCS operates on 2 or more molecules:

```
>>> from rdkit.Chem import rdFMCS
>>> mol1 = Chem.MolFromSmiles("O=C(NCc1cc(OC)c(O)cc1)CCCC/C=C/C(C)C")
>>> mol2 = Chem.MolFromSmiles("CC(C)CCCCCC(=O)NCC1=CC(=C(C=C1)O)OC")
>>> mol3 = Chem.MolFromSmiles("c1(C=O)cc(OC)c(O)cc1")
>>> mols = [mol1,mol2,mol3]
>>> res=rdFMCS.FindMCS(mols)
>>> res
<rdkit.Chem.rdFMCS.MCSResult object at 0x...>
>>> res.numAtoms
10
>>> res.numBonds
10
>>> res.smartsString
'[#6]1(-[#6]):[#6]:[#6](-[#8]-[#6]):[#6](:[#6]:[#6]:1)-[#8]'
>>> res.canceled
False
```

It returns an MCSResult instance with information about the number of atoms and bonds

in the MCS, the SMARTS string which matches the identified MCS, and a flag saying if the algorithm timed out. If no MCS is found then the number of atoms and bonds is set to 0 and the SMARTS to ". By default, two atoms match if they are the same element and two bonds match if they have the same bond type. Specify atomCompare and bondCompare to use different comparison functions, as in:

```
>>> mols = (Chem.MolFromSmiles('NCC'),
            Chem.MolFromSmiles('OC=C'))
>>> rdFMCS.FindMCS(mols).smartsString
'[#6]'
>>> rdFMCS.FindMCS(mols,
                   atomCompare=AtomCompare.CompareAny).smartsString
'[#7,#8]-[#6]'
>>> rdFMCS.FindMCS(mols,
                   bondCompare=BondCompare.CompareAny).smartsString
'[#6]-,=[#6]'
```

The options for the atomCompare argument are: CompareAny says that any atom matches any other atom, CompareElements compares by element type, and CompareIsotopes matches based on the isotope label. Isotope labels can be used to implement user-defined atom types. A bondCompare of CompareAny says that any bond matches any other bond, CompareOrderExact says bonds are equivalent if and only if they have the same bond type, and CompareOrder allows single and aromatic bonds to match each other, but requires an exact order match otherwise:

```
>>> mols = (Chem.MolFromSmiles('c1ccccc1'),
            Chem.MolFromSmiles('C1CCCC=C1'))
>>> rdFMCS.FindMCS(mols,
                   bondCompare=BondCompare.CompareAny).smartsString
'[#6]1:,-[#6]:,-[#6]:,-[#6]:,-[#6]:,=[#6]:,-1'
```

```
>>> rdFMCS.FindMCS(mols,
                   bondCompare=BondCompare.CompareOrderExact).smartsString
'[#6]'
>>> rdFMCS.FindMCS(mols,
                   bondCompare=BondCompare.CompareOrder).smartsString
'[#6](:,-[#6]:,-[#6]:,-[#6]):,-[#6]:,-[#6]'
```

A substructure has both atoms and bonds. By default, the algorithm attempts to maximize the number of bonds found. You can change this by setting the *maximizeBonds* argument to False. Maximizing the number of bonds tends to maximize the number of rings, although two small rings may have fewer bonds than one large ring. You might not want a 3-valent nitrogen to match one which is 5-valent. The default *matchValences* value of False ignores valence information. When True, the atomCompare setting is modified to also require that the two atoms have the same valency.

```
>>> mols = (Chem.MolFromSmiles('NC1OC1'),
            Chem.MolFromSmiles('C1OC1[N+](=O)[O-]'))
>>> rdFMCS.FindMCS(mols).numAtoms
4
>>> rdFMCS.FindMCS(mols, matchValences=True).numBonds
3
```

It can be strange to see a linear carbon chain match a carbon ring, which is what the *ringMatchesRingOnly* default of False does. If you set it to True then ring bonds will only match ring bonds.

```
>>> mols = [Chem.MolFromSmiles("C1CCC1CCC"),
            Chem.MolFromSmiles("C1CCCCCC1")]
>>> FindMCS(mols).smartsString
'[#6](-[#6]-[#6])-[#6]-[#6]-[#6]-[#6]'
>>> FindMCS(mols, ringMatchesRingOnly=True).smartsString
```

```
'[#6&R](-&@[#6&R]-&@[#6&R])-&@[#6&R]'
```

Notice that the SMARTS returned now include ring queries on the atoms and bonds. You can further restrict things and require that partial rings (as in this case) are not allowed. That is, if an atom is part of the MCS and the atom is in a ring of the entire molecule then that atom is also in a ring of the MCS. Setting *completeRingsOnly* to True toggles this requirement.

```
>>> mols = [Chem.MolFromSmiles("CCC1CC2C1CN2"),
            Chem.MolFromSmiles("C1CC2C1CC2")]
>>> rdFMCS.FindMCS(mols).smartsString
'[#6]1-[#6]-[#6](-[#6]-1-[#6])-[#6]'
>>> rdFMCS.FindMCS(mols, ringMatchesRingOnly=True).smartsString
'[#6]1-&@[#6]-&@[#6](-&@[#6]-&@1)-&@[#6&R]'
>>> rdFMCS.FindMCS(mols, completeRingsOnly=True).smartsString
'[#6]1-&@[#6]-&@[#6]-&@[#6]-&@1'
```

Of course the two options can be combined with each other:

```
>>> ms = [Chem.MolFromSmiles(x) for x in ('CC1CCC1','CCC1CC1',)]
>>> rdFMCS.FindMCS(ms,ringMatchesRingOnly=True).smartsString
'[#6&!R]-&!@[#6&R](-&@[#6&R])-&@[#6&R]'
>>> rdFMCS.FindMCS(ms,completeRingsOnly=True).smartsString
'[#6]-&!@[#6]'
>>> rdFMCS.FindMCS(ms,
                   ringMatchesRingOnly=True,
                   completeRingsOnly=True).smartsString
'[#6&!R]-&!@[#6&R]'
```

The MCS algorithm will exhaustively search for a maximum common substructure. Typically this takes a fraction of a second, but for some comparisons this can take minutes or longer. Use the *timeout* parameter to stop the search after the given number of seconds

(wall-clock seconds, not CPU seconds) and return the best match found in that time. If timeout is reached then the *canceled* property of the MCSResult will be True instead of False.

```
>>> mols = [Chem.MolFromSmiles("Nc1ccccc1"*10),
            Chem.MolFromSmiles("Nc1ccccccccc1"*10)]
>>> rdFMCS.FindMCS(mols, timeout=1).canceled
True
```

(The MCS after 50 seconds contained 511 atoms.)

RascalMCES

RascalMCES can only work on 2 molecules at a time:

```
>>> from rdkit.Chem import rdRascalMCES
>>> mol1 = Chem.MolFromSmiles(
                "CN(C)c1ccc(CC(=O)NCCCCCCCCCCNC23CC4CC(C2)CC(C3)C4)cc1
                 CHEMBL153934")
>>> mol2 = Chem.MolFromSmiles(
                "CN(C)c1ccc(CC(=O)NCCCCCCCNC23CC4CC(C2)CC(C3)C4)cc1
                CHEMBL152361")
>>> res = rdRascalMCES.FindMCES(mol1, mol2)
>>> res[0].smartsString
'CN(-C)-c1:c:c:c(-CC(=O)-NCCCCCCC):c:c:1.NC12CC3CC(-C1)-CC(-C2)-C3'
>>> len(res[0].bondMatches())
33
```

It returns a list of RascalResult objects. Each RascalResult contains the 2 molecules that the result pertains to, the SMARTS string of the MCES, the lists of atoms and bonds in the two molecules that match, the Johnson similarity between the 2 molecules, the number of fragments in the MCES, the number of atoms in the largest fragment and whether the run

timed out or not. There is also the method largestFragmentOnly(), which cuts the MCES down to the largest single fragment. This is a non-reversible change, so if you want both results, take a copy first. By default, the MCES algorithm returns the first result it finds of maximum size. Because of symmetry, there may be other equivalent solutions with the same number of atoms and bonds, but with different equivalent bonds matched to each other. If you want to see all MCESs of maximum size, you can use the option allBestMCESs = True. This will increase the run time, partly because more branches in the search tree must be examined, but mostly because sorting the multiple results is quite time-consuming. The results are returned in a consistent order sorted by number of bond matches, then number of fragments (fewer first), then largest fragment size and so on. Some of these aren't trivial to compute. The adamantane example above is particularly extreme because not only is there extensive symmetry about the adamantane end and 2-fold symmetry at the phenyl end but also several points of breaking the matching alkyl chain all of which give rise to valid MCESs of the same size. In this case, sorting into a consistent order takes significantly longer than determining the MCESs in the first place. The MCES differs from a conventional MCS in that it is the maximum common substructure based on bonds rather than atoms. Often the result is the same, but not always. The Johnson similarity is akin to a Tanimoto similarity, but expressed in terms of the atoms and bonds in the MCES. It is the square of the sum of the number of atoms and bonds in the MCES divided by the product of the sums of the numbers of atoms and bonds in the 2 input molecules. It has values between 0.0 (no MCES between the molecules) and 1.0 (the molecules are identical). A key source of efficiency in the RASCAL algorithm is a fast and correct prediction of a maximum value for the Johnson similarity between 2 molecules and hence the maximum size of the MCES. The first step in the algorithm is then a screening, whereby the full MCES determination is not performed if the predicted similarity is less than some desired threshold. The final similarity between the 2 molecules may be less than the threshold, but it will never be higher than the predicted upper bound. RASCAL stems from RApid Similarity CALulation. The default settings for RascalMCES are good for general use, but

they may be altered by passing an optional RascalOptions object:

```
>>> mol1 = Chem.MolFromSmiles('Oc1cccc2C(=O)C=CC(=O)c12')
>>> mol2 = Chem.MolFromSmiles('O1C(=O)C=Cc2cc(OC)c(O)cc12')
>>> results = rdRascalMCES.FindMCES(mol1, mol2)
>>> len(results)
0
>>> opts = rdRascalMCES.RascalOptions()
>>> opts.similarityThreshold = 0.5
>>> results = rdRascalMCES.FindMCES(mol1, mol2, opts)
>>> len(results)
1
>>> f'{results[0].similarity:.2f}'
'0.37'
>>> results[0].smartsString
'Oc1:c:c:c:c:c:1.[#6]=O'
>>> opts.minFragSize = 3
>>> results = rdRascalMCES.FindMCES(mol1, mol2, opts)
>>> len(results)
1
>>> f'{results[0].similarity:.2f}'
'0.25'
>>> results[0].smartsString
'Oc1:c:c:c:c:c:1'
```

In this case, the upper bound on the similarity score is below the default threshold of 0.7, so no results are returned. Setting the threshold to 0.5 produces the second result although, as can be seen, the final similarity is substantially below the threshold. This example also shows a disadvantage of the MCES method, which is that it can produce small fragments in the MCES which are rarely helpful. The option minFragSize can be

used to over-ride the default value of -1, which means no minimum size. Like FindMCS, there is a ringMatchesRingOnly option, and also there's completeAromaticRings, which is True by default, and means that MCESs won't be returned with partial aromatic rings matching:

```
>>> mol1 = Chem.MolFromSmiles('C1CCCC1c1ccncc1')
>>> mol2 = Chem.MolFromSmiles('C1CCCC1c1ccccc1')
>>> results = rdRascalMCES.FindMCES(mol1, mol2, opts)
>>> f'{results[0].similarity:.2f}'
'0.27'
>>> results[0].smartsString
'C1CCCC1-c'
>>> opts.completeAromaticRings = False
>>> results = rdRascalMCES.FindMCES(mol1, mol2, opts)
>>> f'{results[0].similarity:.2f}'
'0.76'
>>> results[0].smartsString
'C1CCCC1-c(:c:c):c:c'
```

This result may look a bit odd, with a single aromatic carbon in the first SMARTS string. This is a consequence of the fact that the MCES works on matching bonds. A better, atom-centric, representation might be C1CCC[$(C-c)]1. When the completeAromaticRings option is set to False, a larger MCES is found, with just the pyridine nitrogen atom not matching the corresponding phenyl carbon atom.

Clustering with Rascal

There are 2 clustering methods available using the Johnson metric. The first, Rascal-Cluster, is a fuzzy method described in 'A Line Graph Algorithm for Clustering Chemical Structures Based on Common Substructural Cores', JW Raymond, PW Willett (). The

second, RascalButinaCluster, uses the Butina sphere-exclusion algorithm. Because of the time-consuming nature of the MCES determination, these clustering methods can be slow to run, so are best used on small sets (no more than a few hundred molecules) of small molecules.

Fingerprinting and Molecular Similarity

The RDKit has a variety of built-in functionality for generating molecular fingerprints and using them to calculate molecular similarity. The most straightforward and consistent way to get fingerprints is to create a FingeprintGenerator object for your fingerprint type of interest and then use that to calculate fingerprints. Fingerprint generators provide a consistent interface to all the supported fingerprinting methods and allow easy generation of fingerprints as:

- bit vectors : *fpgen.GetFingerprint*
- sparse (unfolded) bit vectors : *fpgen.GetSparseFingerprint*
- count vectors : *fpgen.GetCountFingerprint*
- sparse (unfolded) count vectors : *fpgen.GetSparseCountFingerprint*

Note that there are older, legacy methods of generating fingerprints with the RDKit which are still supported, but these will not be covered here.

RDKit (Topological) Fingerprints

```
>>> from rdkit import DataStructs
>>> ms = [Chem.MolFromSmiles('CCOC'), Chem.MolFromSmiles('CCO'),
>>> Chem.MolFromSmiles('COC')]
>>> fpgen = AllChem.GetRDKitFPGenerator()
>>> fps = [fpgen.GetFingerprint(x) for x in ms]
>>> DataStructs.TanimotoSimilarity(fps[0],fps[1])
0.6...
>>> DataStructs.TanimotoSimilarity(fps[0],fps[2])
0.4...
>>> DataStructs.TanimotoSimilarity(fps[1],fps[2])
0.25
```

The examples above used Tanimoto similarity, but one can use different similarity metrics:

```
>>> DataStructs.DiceSimilarity(fps[0],fps[1])
0.75
```

Available similarity metrics include Tanimoto, Dice, Cosine, Sokal, Russel, Kulczynski, McConnaughey, and Tversky. More details about the algorithm used for the RDKit fingerprint can be found in the "RDKit Book". The default set of parameters used by the fingerprinter is:

- minimum path size: 1 bond
- maximum path size: 7 bonds
- fingerprint size: 2048 bits
- number of bits set per hash: 2

You can control these when calling **AllChem.GetRDKitFPGenerator()**:

```
>>> fpgen = AllChem.GetRDKitFPGenerator(maxPath=2,fpSize=1024)
>>> fps = [fpgen.GetFingerprint(x) for x in ms]
>>> DataStructs.TanimotoSimilarity(fps[0],fps[2])
0.5
```

Atom Pairs and Topological Torsions

Atom-pair descriptors are available in several different forms. The standard form is as fingerprint including counts for each bit instead of just zeros and ones:

```
>>> ms = [Chem.MolFromSmiles('C1CCC1OCC'),
          Chem.MolFromSmiles('CC(C)OCC'),
          Chem.MolFromSmiles('CCOCC')]
>>> fpgen = AllChem.GetAtomPairGenerator()
>>> pairFps = [fpgen.GetSparseCountFingerprint(x) for x in ms]
```

Because the space of bits that can be included in atom-pair fingerprints is huge, they are stored in a sparse manner. We can get the list of bits and their counts for each fingerprint as a dictionary:

```
>>> pairFps[-1].GetNonzeroElements()
{541732: 1, 558113: 2, 558115: 2, 558146: 1, 1606690: 2, 1606721: 2}
```

Unlike most other fingerprint types, descriptions of the bits are directly available:

```
>>> from rdkit.Chem.AtomPairs import Pairs
>>> Pairs.ExplainPairScore(558115)
(('C', 1, 0), 3, ('C', 2, 0))
```

The above means: C with 1 neighbor and 0 pi electrons which is 3 bonds from a C with 2 neighbors and 0 pi electrons The usual metric for similarity between atom-pair fingerprints is Dice similarity:

```
>>> from rdkit import DataStructs
>>> DataStructs.DiceSimilarity(pairFps[0],pairFps[1])
0.333...
>>> DataStructs.DiceSimilarity(pairFps[0],pairFps[2])
0.258...
>>> DataStructs.DiceSimilarity(pairFps[1],pairFps[2])
0.56
```

It's also possible to get atom-pair descriptors encoded as a standard bit vector fingerprint.

```
>>> pairFps = [fpgen.GetFingerprint(x) for x in ms]
>>> DataStructs.DiceSimilarity(pairFps[0],pairFps[1])
0.352...
>>> DataStructs.DiceSimilarity(pairFps[0],pairFps[2])
0.266...
>>> DataStructs.DiceSimilarity(pairFps[1],pairFps[2])
0.583...
```

By default the atom pair bit vector fingerprints use a scheme which simulates counts in the bit vectors (described in detail in the "RDKit Book"), but this can be disabled:

```
>>> fpgen = AllChem.GetAtomPairGenerator(countSimulation=False)
>>> pairFps = [fpgen.GetFingerprint(x) for x in ms]
>>> DataStructs.DiceSimilarity(pairFps[0],pairFps[1])
0.5
>>> DataStructs.DiceSimilarity(pairFps[0],pairFps[2])
0.4
>>> DataStructs.DiceSimilarity(pairFps[1],pairFps[2])
0.625
```

Topological torsion descriptors [4] are calculated in essentially the same way:

```
>>> fpgen = AllChem.GetTopologicalTorsionGenerator()
>>> tts = [fpgen.GetSparseCountFingerprint(x) for x in ms]
>>> DataStructs.DiceSimilarity(tts[0],tts[1])
0.166...
```

Topological torsion fingerprints, like atom-pair fingerprints, use a count simulation scheme by default when generating bit vector fingerprints

Morgan Fingerprints (Circular Fingerprints)

This family of fingerprints, better known as circular fingerprints, is built by applying the Morgan algorithm to a set of user-supplied atom invariants. When generating Morgan fingerprints, the radius of the fingerprint can also be provided (the default is 3):

```
>>> from rdkit.Chem import AllChem
>>> fpgen = AllChem.GetMorganGenerator(radius=2)
>>> m1 = Chem.MolFromSmiles('Cc1ccccc1')
>>> fp1 = fpgen.GetSparseCountFingerprint(m1)
```

```
>>> fp1
<rdkit.DataStructs.cDataStructs.ULongSparseIntVect object at 0x...>
>>> m2 = Chem.MolFromSmiles('Cc1ncccc1')
>>> fp2 = fpgen.GetSparseCountFingerprint(m2)
>>> DataStructs.DiceSimilarity(fp1,fp2)
0.55...
```

Morgan fingerprints, like atom pairs and topological torsions, are often used as counts, but it's also possible to calculate them as bit vectors, the default fingerprint size is 2048 bits:

```
>>> fp1 = fpgen.GetFingerprint(m1)
>>> fp1
<rdkit.DataStructs.cDataStructs.ExplicitBitVect object at 0x...>
>>> len(fp1)
2048
>>> fp2 = fpgen.GetFingerprint(m2)
>>> DataStructs.DiceSimilarity(fp1,fp2)
0.51...
```

The default atom invariants use connectivity information similar to those used for the well known ECFP family of fingerprints. Feature-based invariants, similar to those used for the FCFP fingerprints, can also be used by creating the fingerprint generator with an appropriate atom invariant generator. The feature definitions used are defined in the section Feature Definitions Used in the Morgan Fingerprints. At times this can lead to quite different similarity scores:

```
>>> m1 = Chem.MolFromSmiles('c1ccccn1')
>>> m2 = Chem.MolFromSmiles('c1ccco1')
>>> fpgen = AllChem.GetMorganGenerator(radius=2)
>>> fp1 = fpgen.GetSparseCountFingerprint(m1)
>>> fp2 = fpgen.GetSparseCountFingerprint(m2)
```

```
>>> invgen = AllChem.GetMorganFeatureAtomInvGen()
>>> ffpgen = AllChem.GetMorganGenerator(radius=2,
                                        atomInvariantsGenerator=invgen)
>>> ffp1 = ffpgen.GetSparseCountFingerprint(m1)
>>> ffp2 = ffpgen.GetSparseCountFingerprint(m2)
>>> DataStructs.DiceSimilarity(fp1,fp2)
0.36...
>>> DataStructs.DiceSimilarity(ffp1,ffp2)
0.90...
```

When comparing the ECFP/FCFP fingerprints and the Morgan fingerprints generated by the RDKit, remember that the 4 in ECFP4 corresponds to the diameter of the atom environments considered, while the Morgan fingerprints take a radius parameter. So the examples above, with radius=2, are roughly equivalent to ECFP4 and FCFP4.

The user can also provide their own atom invariants using the optional *customAtomInvariants* argument to the *GetFingerprint()* call. Here's a simple example that uses a constant for the invariant; the resulting fingerprints compare the topology of molecules:

```
>>> m1 = Chem.MolFromSmiles('Cc1ccccc1')
>>> m2 = Chem.MolFromSmiles('Cc1ncncn1')
>>> fpgen = AllChem.GetMorganGenerator(radius=2)
>>> fp1 = fpgen.GetFingerprint(m1,customAtomInvariants=[1]*m1.GetNumAtoms())
>>> fp2 = fpgen.GetFingerprint(m2,customAtomInvariants=[1]*m2.GetNumAtoms())
>>> fp1==fp2
True
```

Note that bond order is by default still considered:

```
>>> m3 = Chem.MolFromSmiles('CC1CCCCC1')
>>> fp3 = fpgen.GetFingerprint(m3,customAtomInvariants=[1]*m3.GetNumAtoms())
```

```
>>> fp1==fp3
False
```

But this can also be turned off:

```
>>> fpgen = AllChem.GetMorganGenerator(radius=2,useBondTypes=False)
>>> fp1 = fpgen.GetFingerprint(m1,
                           customAtomInvariants=[1]*m1.GetNumAtoms())
>>> fp3 = fpgen.GetFingerprint(m3,
                           customAtomInvariants=[1]*m3.GetNumAtoms())
>>> fp1==fp3
True
```

MACCS Keys

There is a SMARTS-based implementation of the 166 public MACCS keys. This is not currently supported by the RDKit's fingerprint generators, so you have to use a different interface.

```
>>> from rdkit.Chem import MACCSkeys
>>> ms = [Chem.MolFromSmiles('CCOC'), Chem.MolFromSmiles('CCO'),
>>> Chem.MolFromSmiles('COC')]
>>> fps = [MACCSkeys.GenMACCSKeys(x) for x in ms]
>>> DataStructs.TanimotoSimilarity(fps[0],fps[1])
0.5
>>> DataStructs.TanimotoSimilarity(fps[0],fps[2])
0.538...
>>> DataStructs.TanimotoSimilarity(fps[1],fps[2])
0.214...
```

The MACCS keys were critically evaluated and compared to other MACCS implementations in Q3 2008. In cases where the public keys are fully defined, things looked pretty

good.

Explaining bits from fingerprints

The fingerprint generators can collect information about the atoms/bonds involved in setting bits when a fingerprint is generated. This information is quite useful for understanding which parts of a molecule were involved in each bit. Each fingerprinting method provides different information, but this is all accessed using the additionalOutput argument to the fingerprinting functions.

Morgan Fingerprints

Information is available about the atoms that contribute to particular bits in the Morgan fingerprint via the bit info map. This is a dictionary with one entry per bit set in the fingerprint, the keys are the bit ids, the values are lists of (atom index, radius) tuples.

```
>>> m = Chem.MolFromSmiles('c1cccnc1C')
>>> fpgen = AllChem.GetMorganGenerator(radius=2)
>>> ao = AllChem.AdditionalOutput()
>>> ao.CollectBitInfoMap()
>>> fp = fpgen.GetSparseCountFingerprint(m,additionalOutput=ao)
>>> len(fp.GetNonzeroElements())
16
>>> info = ao.GetBitInfoMap()
>>> len(info)
16
>>> info[98513984]
((1, 1), (2, 1))
>>> info[4048591891]
((5, 2),)
```

Interpreting the above: bit 98513984 is set twice: once by atom 1 and once by atom 2, each at radius 1. Bit 4048591891 is set once by atom 5 at radius 2.

Focusing on bit 4048591891, we can extract the submolecule consisting of all atoms within a radius of 2 of atom 5:

```
>>> env = Chem.FindAtomEnvironmentOfRadiusN(m,2,5)
>>> amap={}
>>> submol=Chem.PathToSubmol(m,env,atomMap=amap)
>>> submol.GetNumAtoms()
6
>>> amap
{0: 0, 1: 1, 3: 2, 4: 3, 5: 4, 6: 5}
```

And then "explain" the bit by generating SMILES for that submolecule:

```
>>> Chem.MolToSmiles(submol)
'ccc(C)nc'
```

This is more useful when the SMILES is rooted at the central atom:

```
>>> Chem.MolToSmiles(submol,rootedAtAtom=amap[5],canonical=False)
'c(cc)(nc)C'
```

An alternate (and faster, particularly for large numbers of molecules) approach to do the same thing, using the function **rdkit.Chem.MolFragmentToSmiles()**:

```
>>> atoms=set()
>>> for bidx in env:
...     atoms.add(m.GetBondWithIdx(bidx).GetBeginAtomIdx())
...     atoms.add(m.GetBondWithIdx(bidx).GetEndAtomIdx())
>>> Chem.MolFragmentToSmiles(m,atomsToUse=list(atoms),bondsToUse=env,rootedAtAtom=5)
'c(C)(cc)nc'
```

RDKit Fingerprints

Information is available about the bond paths that contribute to particular bits in the RDKit fingerprint via the bit info map. This is a dictionary with one entry per bit set in the fingerprint, the keys are the bit ids, the values are tuples of tuples containing bond indices.

```
>>> m = Chem.MolFromSmiles('CCO')
>>> fpgen = AllChem.GetRDKitFPGenerator()
>>> ao = AllChem.AdditionalOutput()
>>> ao.CollectBitPaths()
>>> fp = fpgen.GetSparseCountFingerprint(m,additionalOutput=ao)
>>> len(fp.GetNonzeroElements())
6
>>> paths = ao.GetBitPaths()
>>> len(paths)
6


>>> paths[54413874]
((1,),)
>>> paths[1135572127]
((0, 1),)
>>> paths[1524090560]
((0, 1),)
```

Those last two examples, which each correspond to the path containing bonds 0 and 1, demonstrate that by default each path sets two bits in the RDKit fingerprint. We can, of course, create a fingerprint generator which does not do this:

```
>>> fpgen = AllChem.GetRDKitFPGenerator(numBitsPerFeature=1)
>>> ao = AllChem.AdditionalOutput()
```

```
>>> ao.CollectBitPaths()
>>> fp = fpgen.GetSparseCountFingerprint(m,additionalOutput=ao)
>>> len(fp.GetNonzeroElements())
3
>>> ao.GetBitPaths()
{1524090560: ((0, 1),), 4274652475: ((1,),), 4275705116: ((0,),)}
```

Here we can also use the bond path information to create submolecules:

```
>>> envs = ao.GetBitPaths()[4274652475]
>>> envs
((1,),)
>>> env = envs[0]
>>> atoms=set()
>>> for bidx in env:
...     atoms.add(m.GetBondWithIdx(bidx).GetBeginAtomIdx())
...     atoms.add(m.GetBondWithIdx(bidx).GetEndAtomIdx())
>>> Chem.MolFragmentToSmiles(m,atomsToUse=list(atoms),bondsToUse=env)
'CO'
```

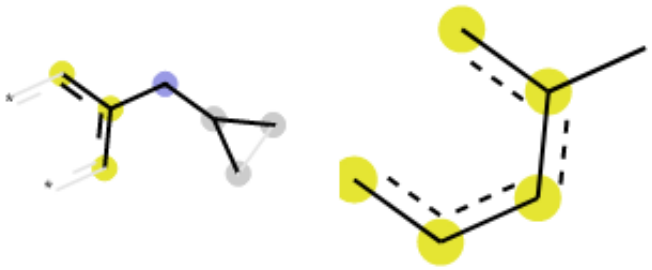
Generating images of fingerprint bits

For the Morgan and RDKit fingerprint types, it's possible to generate images of the atom environment that defines the bit using the functions rdkit.Chem.Draw.DrawMorganBit() and rdkit.Chem.Draw.DrawRDKitBit()

```
>>> from rdkit.Chem import Draw
>>> mol = Chem.MolFromSmiles('c1ccccc1CC1CC1')
>>> fpgen = AllChem.GetMorganGenerator(radius=2)
>>> ao = AllChem.AdditionalOutput()
>>> ao.CollectBitInfoMap()
```

```
>>> fp = fpgen.GetFingerprint(mol,additionalOutput=ao)
>>> bi = ao.GetBitInfoMap()
>>> bi[872]
((6, 2),)
>>> mfp2_svg = Draw.DrawMorganBit(mol, 872, bi, useSVG=True)
>>> fpgen = AllChem.GetRDKitFPGenerator()
>>> ao = AllChem.AdditionalOutput()
>>> ao.CollectBitPaths()
>>> fp = fpgen.GetFingerprint(mol,additionalOutput=ao)
>>> rdkbi = ao.GetBitPaths()
>>> rdkbi[1553]
((0, 1, 9, 5, 4), (2, 3, 4, 9, 5))
>>> rdk_svg = Draw.DrawRDKitBit(mol, 1553, rdkbi, useSVG=True)
```

Producing these images, Morgan bit on the left and RDKit bit on the right:



The default highlight colors for the Morgan bits indicate:

- blue: the central atom in the environment
- yellow: aromatic atoms
- gray: aliphatic ring atoms

The default highlight colors for the RDKit bits indicate:

- yellow: aromatic atoms

Note that in cases where the same bit is set by multiple atoms in the molecule (as for bit 1553 for the RDKit fingerprint in the example above), the drawing functions will display the first example. You can change this by specifying which example to show:

```
>>> rdk_svg = Draw.DrawRDKitBit(mol, 1553, rdkbi, whichExample=1, useSVG=True)
```

Producing this image:

Picking Diverse Molecules Using Fingerprints

A common task is to pick a small subset of diverse molecules from a larger set. The RDKit provides a number of approaches for doing this in the rdkit.SimDivFilters module. The most efficient of these uses the MaxMin algorithm. Here's an example: Start by reading in a set of molecules and generating Morgan fingerprints:

```
>>> from rdkit import Chem
>>> from rdkit.Chem import rdFingerprintGenerator
>>> fpgen = rdFingerprintGenerator.GetMorganGenerator(radius=3)
>>> from rdkit import DataStructs
>>> from rdkit.SimDivFilters.rdSimDivPickers import MaxMinPicker
>>> with Chem.SDMolSupplier('data/actives_5ht3.sdf') as suppl:
...     ms = [x for x in suppl if x is not None]
>>> fps = [fpgen.GetFingerprint(x) for x in ms]
>>> nfps = len(fps)
```

Now create a picker and grab a set of 10 diverse molecules:

```
>>> picker = MaxMinPicker()
>>> pickIndices = picker.LazyBitVectorPick(fps,nfps,10,seed=23)
>>> list(pickIndices)
[93, 137, 135, 109, 18, 150, 142, 12, 6, 160]
```

Note that the picker just returns indices of the fingerprints; we can get the molecules themselves as follows:

```
>>> picks = [ms[x] for x in pickIndices]
```

If we aren't working with bit vector fingerprints, we can also do a diversity pick by providing our own distance matrix to the algorithm. This is less efficient than the above approach, but still works quite quickly:

```
>>> fps = [fpgen.GetSparseCountFingerprint(x) for x in ms]
>>> def distij(i,j,fps=fps):
...     return 1-DataStructs.DiceSimilarity(fps[i],fps[j])
>>> picker = MaxMinPicker()
>>> pickIndices = picker.LazyPick(distij,nfps,10,seed=23)
>>> list(pickIndices)
[93, 109, 154, 6, 95, 135, 151, 61, 137, 139]
```

Generating Similarity Maps Using Fingerprints

Similarity maps are a way to visualize the atomic contributions to the similarity between a molecule and a reference molecule. The methodology is described in Ref. [17] . They are in the rdkit.Chem.Draw.SimilarityMaps module: Start by creating two molecules:

```
>>> from rdkit import Chem
>>> smi = 'COc1cccc2cc(C(=O)NCCCCN3CCN(c4cccc5nccnc54)CC3)oc21'
>>> ref_smi = 'CCCN(CCCCN1CCN(c2ccccc2OC)CC1)Cc1ccc2ccccc2c1'
```

```
>>> mol = Chem.MolFromSmiles(smi)
>>> refmol = Chem.MolFromSmiles(ref_smi)
```

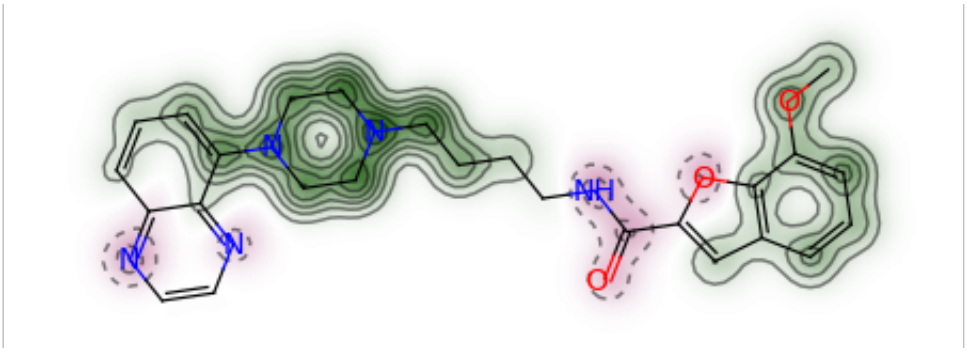
The SimilarityMaps module supports three kind of fingerprints: atom pairs, topological torsions and Morgan fingerprints.

```
>>> from rdkit.Chem import Draw
>>> from rdkit.Chem.Draw import SimilarityMaps
>>> fp = SimilarityMaps.GetAPFingerprint(mol, fpType='normal')
>>> fp = SimilarityMaps.GetTTFingerprint(mol, fpType='normal')
>>> fp = SimilarityMaps.GetMorganFingerprint(mol, fpType='bv')
```

The types of atom pairs and torsions are normal (default), hashed and bit vector (bv). The types of the Morgan fingerprint are bit vector (bv, default) and count vector (count). The function generating a similarity map for two fingerprints requires the specification of the fingerprint function and optionally the similarity metric. The default for the latter is the Dice similarity. Using all the default arguments of the Morgan fingerprint function, the similarity map can be generated like this:

```
>>> fig, maxweight = SimilarityMaps.GetSimilarityMapForFingerprint(
                                refmol, mol, SimilarityMaps.GetMorganFingerprint)
```

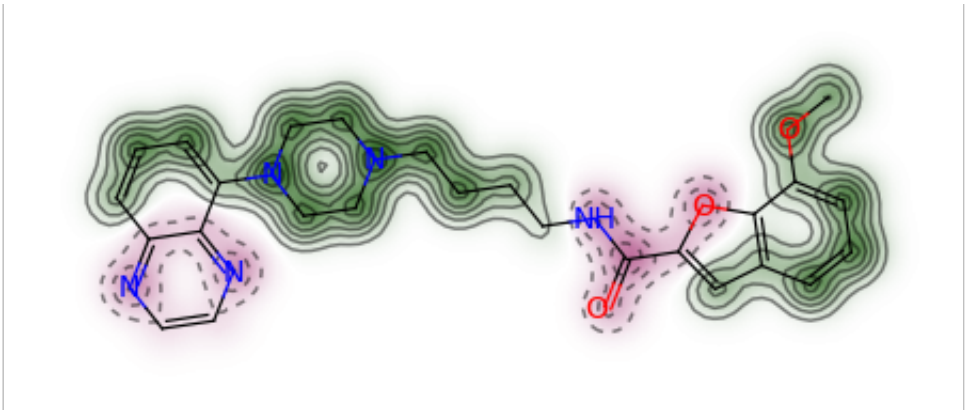
Producing this image:



For a different type of Morgan (e.g. count) and radius = 1 instead of 2, as well as a different similarity metric (e.g. Tanimoto), the call becomes:

```
>>> from rdkit import DataStructs
>>> fig, maxweight = SimilarityMaps.GetSimilarityMapForFingerprint(
            refmol, mol, lambda m,idx: SimilarityMaps.GetMorganFingerprint(
                    m, atomId=idx, radius=1, fpType='count'),
                    metric=DataStructs.TanimotoSimilarity)
```

Producing this image:



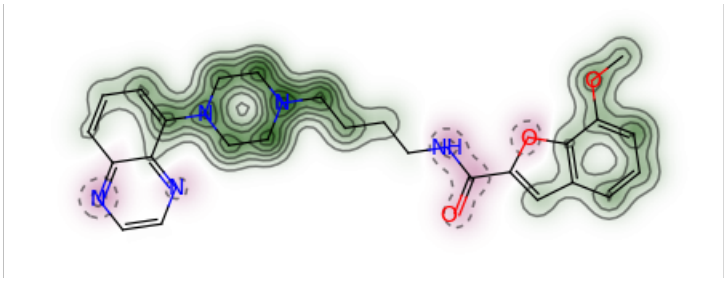
The convenience function GetSimilarityMapForFingerprint involves the normalisation of the atomic weights such that the maximum absolute weight is 1. Therefore, the function outputs the maximum weight that was found when creating the map.

```
>>> print(maxweight)
0.05747...
```

If one does not want the normalisation step, the map can be created like:

```
>>> weights = SimilarityMaps.GetAtomicWeightsForFingerprint(refmol, mol,
                        SimilarityMaps.GetMorganFingerprint)
>>> print(["%.2f " % w for w in weights])
['0.05 ', ...
>>> fig = SimilarityMaps.GetSimilarityMapFromWeights(mol, weights)
```

Producing this image:



Descriptor Calculation

A variety of descriptors are available within the RDKit. The complete list is provided in List of Available Descriptors section below. Most of the descriptors are straightforward to use from Python via the centralized rdkit.Chem.Descriptors module:

```
>>> from rdkit.Chem import Descriptors
>>> m = Chem.MolFromSmiles('c1ccccc1C(=O)O')
>>> Descriptors.TPSA(m)
37.3
>>> Descriptors.MolLogP(m)
1.3848
```

Calculating All Descriptors

The rdkit.Chem.Descriptors module provides a convenience function, *CalcMolDescriptors()*, to calculate all available descriptors for a molecule. *CalcMolDescriptors()* returns a dictionary with descriptor names as the keys and descriptor values as the values:

```
>>> vals = Descriptors.CalcMolDescriptors(m)
>>> vals['TPSA']
37.3
>>> vals['NumHDonors']
1
```

CalcMolDescriptors() makes it easy to generate descriptors for a set of molecules and get the values into a pandas DataFrame:

```
>>> descrs = [Descriptors.CalcMolDescriptors(mol) for mol in mols]
>>> df = pandas.DataFrame(descrs)
>>> df.head()
>>> df.head(3)
```

```
'''
      MaxEStateIndex  ...  fr_thiophene  fr_unbrch_alkane  fr_urea
    0        8.361111 ...             0                 0        0
    1        8.361111 ...             0                 0        0
    2        8.334769 ...             0                 0        0

    [3 rows x 208 columns]
'''
```

Calculating Partial Charges

Partial charges are handled a bit differently:

```
>>> m = Chem.MolFromSmiles('c1ccccc1C(=O)O')
>>> AllChem.ComputeGasteigerCharges(m)
>>> m.GetAtomWithIdx(0).GetDoubleProp('_GasteigerCharge')
-0.047...
```

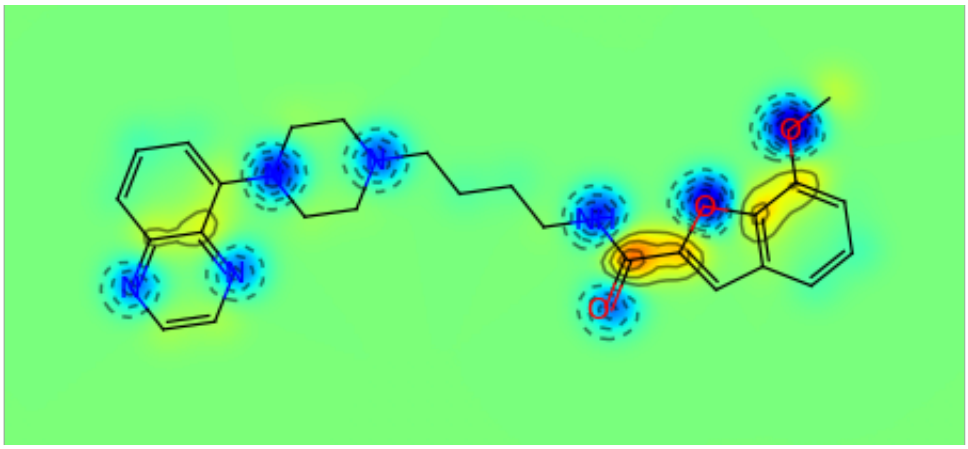
Visualization of Descriptors

Similarity maps can be used to visualize descriptors that can be divided into atomic contributions. The Gasteiger partial charges can be visualized as (using a different color scheme):

```
>>> from rdkit.Chem.Draw import SimilarityMaps
>>> smi = 'COc1cccc2cc(C(=O)NCCCCN3CCN(c4cccc5nccnc54)CC3)oc21'
>>> mol = Chem.MolFromSmiles(smi)
>>> AllChem.ComputeGasteigerCharges(mol)
>>> contribs = [mol.GetAtomWithIdx(i).GetDoubleProp('_GasteigerCharge')
                for i in range(mol.GetNumAtoms())]
>>> fig = SimilarityMaps.GetSimilarityMapFromWeights(
                            mol, contribs,
```

```
                                colorMap='jet',
                                contourLines=10)
```

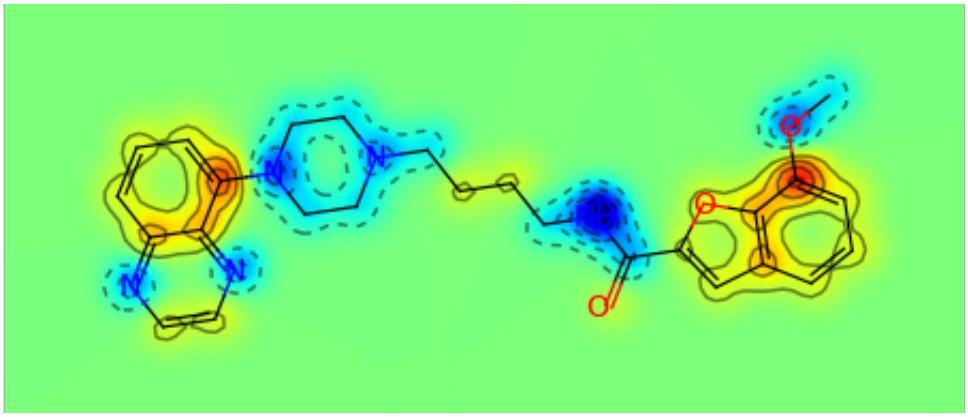
Producing this image:



Or for the Crippen contributions to logP:

```
>>> from rdkit.Chem import rdMolDescriptors
>>> contribs = rdMolDescriptors._CalcCrippenContribs(mol)
>>> fig = SimilarityMaps.GetSimilarityMapFromWeights(mol,
                                [x for x,y in contribs],
                                colorMap='jet', contourLines=10)
```

Producing this image:



Chemical Reactions

The RDKit also supports applying chemical reactions to sets of molecules. One way of constructing chemical reactions is to use a SMARTS-based language similar to Daylight's Reaction SMILES:

```
>>> rxn_smarts = '[C:1](=[O:2])-[OD1].[N!H0:3]>>[C:1](=[O:2])[N:3]'
>>> rxn = AllChem.ReactionFromSmarts(rxn_smarts)
>>> rxn
<rdkit.Chem.rdChemReactions.ChemicalReaction object at 0x...>
>>> rxn.GetNumProductTemplates()
1
>>> ps = rxn.RunReactants((Chem.MolFromSmiles('CC(=O)O'),
                           Chem.MolFromSmiles('NC')))
>>> len(ps) # one entry for each possible set of products
1
>>> len(ps[0]) # each entry contains one molecule for each product
1
>>> Chem.MolToSmiles(ps[0][0])
'CNC(C)=O'
>>> ps = rxn.RunReactants((Chem.MolFromSmiles('C(COC(=O)O)C(=O)O'),
                           Chem.MolFromSmiles('NC')))
>>> len(ps)
2
>>> Chem.MolToSmiles(ps[0][0])
'CNC(=O)OCCC(=O)O'
>>> Chem.MolToSmiles(ps[1][0])
'CNC(=O)CCOC(=O)O'
```

Reactions can also be built from MDL rxn files:

```
>>> rxn = AllChem.ReactionFromRxnFile('data/AmideBond.rxn')
>>> rxn.GetNumReactantTemplates()
2
>>> rxn.GetNumProductTemplates()
1
>>> ps = rxn.RunReactants((Chem.MolFromSmiles('CC(=O)O'),
                           Chem.MolFromSmiles('NC')))
>>> len(ps)
1
>>> Chem.MolToSmiles(ps[0][0])
'CNC(C)=O'
```

It is, of course, possible to do reactions more complex than amide bond formation:

```
>>> rxn_smarts = '[C:1]=[C:2].[C:3]=[*:4][*:5]=[C:6] \
                  >> \
                  [C:1]1[C:2][C:3][*:4]=[*:5][C:6]1'
>>> rxn = AllChem.ReactionFromSmarts(rxn_smarts)
>>> ps = rxn.RunReactants((Chem.MolFromSmiles('OC=C'),
                           Chem.MolFromSmiles('C=CC(N)=C')))
>>> Chem.MolToSmiles(ps[0][0])
'NC1=CCCC(O)C1'
```

Note in this case that there are multiple mappings of the reactants onto the templates, so we have multiple product sets:

```
>>> len(ps)
4
```

You can use canonical smiles and a python dictionary to get the unique products:

```
>>> uniqps = {}
>>> for p in ps:
```

```
...     smi = Chem.MolToSmiles(p[0])
...     uniqps[smi] = p[0]
>>> sorted(uniqps.keys())
['NC1=CCC(O)CC1', 'NC1=CCCC(O)C1']
```

Note that the molecules that are produced by the chemical reaction processing code are not sanitized, as this artificial reaction demonstrates:

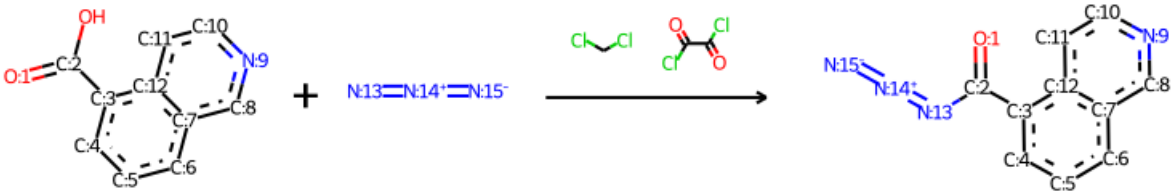
```
>>> rxn_smarts = '[C:1]=[C:2][C:3]=[C:4].[C:5]=[C:6] \
                 >> \
                 [C:1]1=[C:2][C:3]=[C:4][C:5]=[C:6]1'
>>> rxn = AllChem.ReactionFromSmarts(rxm_smarts)
>>> ps = rxn.RunReactants((Chem.MolFromSmiles('C=CC=C'),
                           Chem.MolFromSmiles('C=C')))
>>> Chem.MolToSmiles(ps[0][0])
'C1=CC=CC=C1'
>>> p0 = ps[0][0]
>>> Chem.SanitizeMol(p0)
rdkit.Chem.rdmolops.SanitizeFlags.SANITIZE_NONE
>>> Chem.MolToSmiles(p0)
'c1ccccc1'
```

Drawing Chemical Reactions

The RDKit's MolDraw2D-based rendering can also handle chemical reactions.

```
>>> from rdkit.Chem import Draw
>>> rxn_smarts = '[cH:5]1[cH:6][c:7]2[cH:8][n:9][cH:10][cH:11][c:12]2[c:3] \
                  ([cH:4]1)[C:2](=[O:1])O.[N-:13]=[N+:14]=[N-:15] \
                  >C(Cl)Cl.C(=O)(C(=O)Cl)Cl> \
                  [cH:5]1[cH:6][c:7]2[cH:8][n:9][cH:10][cH:11][c:12]2[c:3] \
                  ([cH:4]1)[C:2](=[O:1])[N:13]=[N+:14]=[N-:15]'
>>> rxn = AllChem.ReactionFromSmarts(rxn_smarts,useSmiles=True)
>>> d2d = Draw.MolDraw2DCairo(800,300)
>>> d2d.DrawReaction(rxn)
>>> png = d2d.GetDrawingText()
>>> open('./images/reaction1.o.png','wb+').write(png)
```

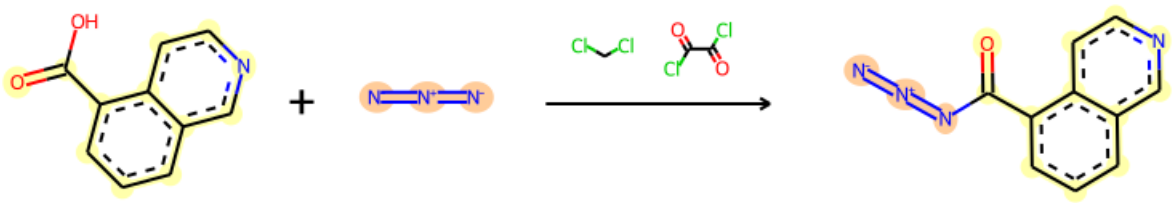
the result looks like this:



There's another drawing mode which leaves out the atom map information but which highlights which of the reactants atoms in the products come from:

```
>>> d2d = Draw.MolDraw2DCairo(800,300)
>>> d2d.DrawReaction(rxn,highlightByReactant=True)
```

```
>>> png = d2d.GetDrawingText()
>>> open('./images/reaction1_highlight.o.png','wb+').write(png)
```



As of the 2020.09 release, PNG images of reactions include metadata allowing the reaction to be reconstructed:

```
>>> newRxn = AllChem.ReactionFromPNGString(png)
>>> AllChem.ReactionToSmarts(newRxn)
```

Advanced Reaction Functionality

Protecting Atoms

Sometimes, particularly when working with rxn files, it is difficult to express a reaction exactly enough to not end up with extraneous products. The RDKit provides a method of "protecting" atoms to disallow them from taking part in reactions. This can be demonstrated re-using the amide-bond formation reaction used above. The query for amines isn't specific enough, so it matches any nitrogen that has at least one H attached. So if we apply the reaction to a molecule that already has an amide bond, the amide N is also treated as a reaction site:

```
>>> rxn = AllChem.ReactionFromRxnFile('data/AmideBond.rxn')
>>> acid = Chem.MolFromSmiles('CC(=O)O')
```

```
>>> base = Chem.MolFromSmiles('CC(=O)NCCN')
>>> ps = rxn.RunReactants((acid,base))
>>> len(ps)
2
>>> Chem.MolToSmiles(ps[0][0])
'CC(=O)N(CCN)C(C)=O'
>>> Chem.MolToSmiles(ps[1][0])
'CC(=O)NCCNC(C)=O'
```

The first product corresponds to the reaction at the amide N. We can prevent this from happening by protecting all amide Ns. Here we do it with a substructure query that matches amides and thioamides and then set the "_protected" property on matching atoms:

```
>>> amidep = Chem.MolFromSmarts('[N;$(NC=[O,S])]')
>>> for match in base.GetSubstructMatches(amidep):
...     base.GetAtomWithIdx(match[0]).SetProp('_protected','1')
```

Now the reaction only generates a single product:

```
>>> ps = rxn.RunReactants((acid,base))
>>> len(ps)
1
>>> Chem.MolToSmiles(ps[0][0])
'CC(=O)NCCNC(C)=O'
```

Recap Implementation

Associated with the chemical reaction functionality is an implementation of the Recap algorithm. Recap uses a set of chemical transformations mimicking common reactions carried out in the lab in order to decompose a molecule into a series of reasonable fragments. The RDKit rdkit.Chem.Recap implementation keeps track of the hierarchy of transformations that were applied:

```
>>> from rdkit import Chem
>>> from rdkit.Chem import Recap
>>> m = Chem.MolFromSmiles('c1ccccc1OCCOC(=O)CC')
>>> hierarch = Recap.RecapDecompose(m)
>>> type(hierarch)
<class 'rdkit.Chem.Recap.RecapHierarchyNode'>
```

The hierarchy is rooted at the original molecule:

```
>>> hierarch.smiles
'CCC(=O)OCCOc1ccccc1'
```

and each node tracks its children using a dictionary keyed by SMILES:

```
>>> ks=hierarch.children.keys()
>>> sorted(ks)
['*C(=O)CC', '*CCOC(=O)CC', '*CCOc1ccccc1', '*OCCOc1ccccc1', '*c1ccccc1']
```

The nodes at the bottom of the hierarchy (the leaf nodes) are easily accessible, also as a dictionary keyed by SMILES:

```
>>> ks=hierarch.GetLeaves().keys()
>>> ks=sorted(ks)
>>> ks
['*C(=O)CC', '*CCO*', '*CCOc1ccccc1', '*c1ccccc1']
```

Notice that dummy atoms are used to mark points where the molecule was fragmented. The nodes themselves have associated molecules:

```
>>> leaf = hierarch.GetLeaves()[ks[0]]
>>> Chem.MolToSmiles(leaf.mol)
'*C(=O)CC'
```

BRICS Implementation

The RDKit also provides an implementation of the BRICS algorithm. BRICS provides another method for fragmenting molecules along synthetically accessible bonds:

```
>>> from rdkit.Chem import BRICS
>>> with Chem.SDMolSupplier('data/cdk2.sdf') as cdk2mols:
...     m1 = cdk2mols[0]
...     m2 = cdk2mols[20]
>>> sorted(BRICS.BRICSDecompose(m1))
['[14*]c1nc(N)nc2[nH]cnc12', '[3*]O[3*]', '[4*]CC(=O)C(C)C']
>>> sorted(BRICS.BRICSDecompose(m2))
['[1*]C(=O)NN(C)C', '[14*]c1[nH]nc2c1C(=O)c1c([16*])cccc1-2',
 '[16*]c1ccc([16*])cc1', '[3*]OC', '[5*]N[5*]']
```

Notice that RDKit BRICS implementation returns the unique fragments generated from a molecule and that the dummy atoms are tagged to indicate which type of reaction applies. It's quite easy to generate the list of all fragments for a group of molecules:

```
>>> allfrags=set()
>>> with Chem.SDMolSupplier('data/cdk2.sdf') as cdk2mols:
...     for m in cdk2mols:
...         if m is None:
...             continue
...         else:
```

```
...             pieces = BRICS.BRICSDecompose(m)
...             allfrags.update(pieces)
>>> len(allfrags)
90
>>> sorted(allfrags)[:5]
['NS(=O)(=O)c1ccc(N/N=C2\\C(=O)Nc3ccc(Br)cc32)cc1',
 '[1*]C(=O)C(C)C',
 '[1*]C(=O)NN(C)C',
 '[1*]C(=O)NN1CC[NH+](C)CC1',
 '[1*]C(C)=O']
```

The BRICS module also provides an option to apply the BRICS rules to a set of fragments to create new molecules:

```
>>> import random
>>> random.seed(127)
>>> fragms = [Chem.MolFromSmiles(x) for x in sorted(allfrags)]
>>> random.seed(0xf00d)
>>> ms = BRICS.BRICSBuild(fragms)
```

The result is a generator object:

```
>>> ms
<generator object BRICSBuild at 0x...>
```

That returns molecules on request:

```
>>> prods = [next(ms) for x in range(10)]
>>> prods[0]
<rdkit.Chem.rdchem.Mol object at 0x...>
```

The molecules have not been sanitized, so it's a good idea to at least update the valences before continuing:

```
>>> for prod in prods:
...     prod.UpdatePropertyCache(strict=False)
>>> Chem.MolToSmiles(prods[0],True)
'[H]/N=C(\\N)NC(=O)C(C)C'
>>> Chem.MolToSmiles(prods[1],True)
'CC(C)C(=O)N/C=C1\\C(=O)Nc2ccc3ncsc3c21'
>>> Chem.MolToSmiles(prods[2],True)
'CC(C)C(=O)N/C=C1\\C(=O)Nc2ccccc21'
>>> Chem.MolToSmiles(prods[3],True)
'CNC(=O)C(C)C'
```

By default those results come back in a random order (technically the example above will always return the same results since we seeded Python's random number generator just before calling BRICSBuild()). If you want the results to be returned in a consistent order use the scrambleReagents argument:

```
>>> ms = BRICS.BRICSBuild(fragms, scrambleReagents=False)
>>> prods = [next(ms) for x in range(10)]
>>> for prod in prods:
...    prod.UpdatePropertyCache(strict=False)
>>> Chem.MolToSmiles(prods[0],True)
'COC(=O)C(C)C'
>>> Chem.MolToSmiles(prods[1],True)
'CNC(=O)C(C)C'
>>> Chem.MolToSmiles(prods[2],True)
'CC(C)C(=O)NC(=N)N'
```

Other fragmentation approaches

In addition to the methods described above, the RDKit provide a very flexible generic function for fragmenting molecules along user-specified bonds. Here's a quick demonstration of

using that to break all bonds between atoms in rings and atoms not in rings. We start by finding all the atom pairs:

```
>>> m = Chem.MolFromSmiles('CC1CC(O)C1CCC1CC1')
>>> bis = m.GetSubstructMatches(Chem.MolFromSmarts('[!R][R]'))
>>> bis
((0, 1), (4, 3), (6, 5), (7, 8))
```

then we get the corresponding bond indices:

```
>>> bs = [m.GetBondBetweenAtoms(x,y).GetIdx() for x,y in bis]
>>> bs
[0, 3, 5, 7]
```

then we use those bond indices as input to the fragmentation function:

```
>>> nm = Chem.FragmentOnBonds(m,bs)
```

the output is a molecule that has dummy atoms marking the places where bonds were broken:

```
>>> Chem.MolToSmiles(nm,True)
'*C1CC([4*])C1[6*].[1*]C.[3*]O.[5*]CC[8*].[7*]C1CC1'
```

By default the attachment points are labelled (using isotopes) with the index of the atom that was removed. We can also provide our own set of atom labels in the form of pairs of unsigned integers. The first value in each pair is used as the label for the dummy that replaces the bond's begin atom, the second value in each pair is for the dummy that replaces the bond's end atom. Here's an example, repeating the analysis above and marking the positions where the non-ring atoms were with the label 10 and marking the positions where the ring atoms were with label 1:

```
>>> bis = m.GetSubstructMatches(Chem.MolFromSmarts('[!R][R]'))
>>> bs = []
```

```
>>> labels=[]
>>> for bi in bis:
...     b = m.GetBondBetweenAtoms(bi[0],bi[1])
...     if b.GetBeginAtomIdx()==bi[0]:
...         labels.append((10,1))
...     else:
...         labels.append((1,10))
...     bs.append(b.GetIdx())
>>> nm = Chem.FragmentOnBonds(m,bs,dummyLabels=labels)
>>> Chem.MolToSmiles(nm,True)
'[1*]C.[1*]CC[1*].[1*]O.[10*]C1CC([10*])C1[10*].[10*]C1CC1'
```

Chemical Features and Pharmacophores

Chemical Features

Chemical features in the RDKit are defined using a SMARTS-based feature definition language (described in detail in the RDKit book). To identify chemical features in molecules, you first must build a feature factory:

```
>>> from rdkit import Chem
>>> from rdkit.Chem import ChemicalFeatures
>>> from rdkit import RDConfig
>>> import os
>>> fdefName = os.path.join(RDConfig.RDDataDir,'BaseFeatures.fdef')
>>> factory = ChemicalFeatures.BuildFeatureFactory(fdefName)
```

and then use the factory to search for features:

```
>>> m = Chem.MolFromSmiles('OCc1ccccc1CN')
>>> feats = factory.GetFeaturesForMol(m)
>>> len(feats)
8
```

The individual features carry information about their family (e.g. donor, acceptor, etc.), type (a more detailed description), and the atom(s) that is/are associated with the feature:

```
>>> feats[0].GetFamily()
'Donor'
>>> feats[0].GetType()
'SingleAtomDonor'
>>> feats[0].GetAtomIds()
(0,)
>>> feats[4].GetFamily()
```

```
'Aromatic'
>>> feats[4].GetAtomIds()
(2, 3, 4, 5, 6, 7)
```

If the molecule has coordinates, then the features will also have reasonable locations:

```
>>> from rdkit.Chem import AllChem
>>> AllChem.Compute2DCoords(m)
0
>>> feats[0].GetPos()
<rdkit.Geometry.rdGeometry.Point3D object at 0x...>
>>> list(feats[0].GetPos())
[2.07..., -2.335..., 0.0]
```

2D Pharmacophore Fingerprints

Combining a set of chemical features with the 2D (topological) distances between them gives a 2D pharmacophore. When the distances are binned, unique integer ids can be assigned to each of these pharmacophores and they can be stored in a fingerprint. Details of the encoding are in the The RDKit Book. Generating pharmacophore fingerprints requires chemical features generated via the usual RDKit feature-typing mechanism:

```
>>> from rdkit import Chem
>>> from rdkit.Chem import ChemicalFeatures
>>> fdefName = 'data/MinimalFeatures.fdef'
>>> featFactory = ChemicalFeatures.BuildFeatureFactory(fdefName)
```

The fingerprints themselves are calculated using a signature (fingerprint) factory, which keeps track of all the parameters required to generate the pharmacophore:

```
>>> from rdkit.Chem.Pharm2D.SigFactory import SigFactory
>>> sigFactory = SigFactory(featFactory,minPointCount=2,maxPointCount=3)
```

```
>>> sigFactory.SetBins([(0,2),(2,5),(5,8)])
>>> sigFactory.Init()
>>> sigFactory.GetSigSize()
885
```

The signature factory is now ready to be used to generate fingerprints, a task which is done using the rdkit.Chem.Pharm2D.Generate module:

```
>>> from rdkit.Chem.Pharm2D import Generate
>>> mol = Chem.MolFromSmiles('OCC(=O)CCCN')
>>> fp = Generate.Gen2DFingerprint(mol,sigFactory)
>>> fp
<rdkit.DataStructs.cDataStructs.SparseBitVect object at 0x...>
>>> len(fp)
885
>>> fp.GetNumOnBits()
57
```

Details about the bits themselves, including the features that are involved and the binned distance matrix between the features, can be obtained from the signature factory:

```
>>> list(fp.GetOnBits())[:5]
[1, 2, 6, 7, 8]
>>> sigFactory.GetBitDescription(1)
'Acceptor Acceptor |0 1|1 0|'
>>> sigFactory.GetBitDescription(2)
'Acceptor Acceptor |0 2|2 0|'
>>> sigFactory.GetBitDescription(8)
'Acceptor Donor |0 2|2 0|'
>>> list(fp.GetOnBits())[-5:]
[704, 706, 707, 708, 714]
```

```
>>> sigFactory.GetBitDescription(707)
'Donor Donor PosIonizable |0 1 2|1 0 1|2 1 0|'
>>> sigFactory.GetBitDescription(714)
'Donor Donor PosIonizable |0 2 2|2 0 0|2 0 0|'
```

For the sake of convenience (to save you from having to edit the fdef file every time) it is possible to disable particular feature types within the SigFactory:

```
>>> sigFactory.skipFeats=['PosIonizable']
>>> sigFactory.Init()
>>> sigFactory.GetSigSize()
510
>>> fp2 = Generate.Gen2DFingerprint(mol,sigFactory)
>>> fp2.GetNumOnBits()
36
```

Another possible set of feature definitions for 2D pharmacophore fingerprints in the RDKit are those published by Gobbi and Poppinger. The module rdkit.Chem.Pharm2D.Gobbi_Pharm2D has a pre-configured signature factory for these fingerprint types. Here's an example of using it:

```
>>> from rdkit import Chem
>>> from rdkit.Chem.Pharm2D import Gobbi_Pharm2D,Generate
>>> m = Chem.MolFromSmiles('OCC=CC(=O)O')
>>> fp = Generate.Gen2DFingerprint(m,Gobbi_Pharm2D.factory)
>>> fp
<rdkit.DataStructs.cDataStructs.SparseBitVect object at 0x...>
>>> fp.GetNumOnBits()
8
>>> list(fp.GetOnBits())
[23, 30, 150, 154, 157, 185, 28878, 30184]
```

```
>>> Gobbi_Pharm2D.factory.GetBitDescription(157)
'HA HD |0 3|3 0|'
>>> Gobbi_Pharm2D.factory.GetBitDescription(30184)
'HA HD HD |0 3 0|3 0 3|0 3 0|'
```

Molecular Fragments

The RDKit contains a collection of tools for fragmenting molecules and working with those fragments. Fragments are defined to be made up of a set of connected atoms that may have associated functional groups. This is more easily demonstrated than explained:

```
>>> fName=os.path.join(RDConfig.RDDataDir,'FunctionalGroups.txt')
>>> from rdkit.Chem import FragmentCatalog
>>> fparams = FragmentCatalog.FragCatParams(1,6,fName)
>>> fparams.GetNumFuncGroups()
39
>>> fcat=FragmentCatalog.FragCatalog(fparams)
>>> fcgen=FragmentCatalog.FragCatGenerator()
>>> m = Chem.MolFromSmiles('OCC=CC(=O)O')
>>> fcgen.AddFragsFromMol(m,fcat)
3
>>> fcat.GetEntryDescription(0)
'C<-O>C'
>>> fcat.GetEntryDescription(1)
'C=C<-C(=O)O>'
>>> fcat.GetEntryDescription(2)
'C<-C(=O)O>=CC<-O>'
```

The fragments are stored as entries in a rdkit.Chem.rdfragcatalog.FragCatalog. Notice that the entry descriptions include pieces in angular brackets (e.g. between '<' and '>'). These describe the functional groups attached to the fragment. For example, in the above example, the catalog entry 0 corresponds to an ethyl fragment with an alcohol attached to one of the carbons and entry 1 is an ethylene with a carboxylic acid on one carbon. Detailed information about the functional groups can be obtained by asking the fragment for the ids of the functional groups it contains and then looking those ids up in the rd-

kit.Chem.rdfragcatalog.FragCatParams object:

```
>>> list(fcat.GetEntryFuncGroupIds(2))
[34, 1]
>>> fparams.GetFuncGroup(1)
<rdkit.Chem.rdchem.Mol object at 0x...>
>>> Chem.MolToSmarts(fparams.GetFuncGroup(1))
'*-C(=O)[O&D1]'
>>> Chem.MolToSmarts(fparams.GetFuncGroup(34))
'*-[O&D1]'
>>> fparams.GetFuncGroup(1).GetProp('_Name')
'-C(=O)O'
>>> fparams.GetFuncGroup(34).GetProp('_Name')
'-O'
```

The catalog is hierarchical: smaller fragments are combined to form larger ones. From a small fragment, one can find the larger fragments to which it contributes using the rdkit.Chem.rdfragcatalog.FragCatalog.GetEntryDownIds() method:

```
>>> fcat=FragmentCatalog.FragCatalog(fparams)
>>> m = Chem.MolFromSmiles('OCC(NC1CC1)CCC')
>>> fcgen.AddFragsFromMol(m,fcat)
15
>>> fcat.GetEntryDescription(0)
'C<-O>C'
>>> fcat.GetEntryDescription(1)
'CN<-cPropyl>'
>>> list(fcat.GetEntryDownIds(0))
[3, 4]
>>> fcat.GetEntryDescription(3)
```

```
'C<-O>CC'
>>> fcat.GetEntryDescription(4)
'C<-O>CN<-cPropyl>'
```

The fragments from multiple molecules can be added to a catalog:

```
>>> with Chem.SmilesMolSupplier('data/bzr.smi') as suppl:
...     ms = [x for x in suppl]
>>> fcat=FragmentCatalog.FragCatalog(fparams)
>>> for m in ms: nAdded=fcgen.AddFragsFromMol(m,fcat)
>>> fcat.GetNumEntries()
1169
>>> fcat.GetEntryDescription(0)
'Cc'
>>> fcat.GetEntryDescription(100)
'cc-nc(C)n'
```

The fragments in a catalog are unique, so adding a molecule a second time doesn't add any new entries:

```
>>> fcgen.AddFragsFromMol(ms[0],fcat)
0
>>> fcat.GetNumEntries()
1169
```

Once a rdkit.Chem.rdfragcatalog.FragCatalog has been generated, it can be used to fingerprint molecules:

```
>>> fpgen = FragmentCatalog.FragFPGenerator()
>>> fp = fpgen.GetFPForMol(ms[8],fcat)
>>> fp
<rdkit.DataStructs.cDataStructs.ExplicitBitVect object at 0x...>
```

```
>>> fp.GetNumOnBits()
189
```

The rest of the machinery associated with fingerprints can now be applied to these fragment fingerprints. For example, it's easy to find the fragments that two molecules have in common by taking the intersection of their fingerprints:

```
>>> fp2 = fpgen.GetFPForMol(ms[7],fcat)
>>> andfp = fp&fp2
>>> obl = list(andfp.GetOnBits())
>>> fcat.GetEntryDescription(obl[-1])
'ccc(cc)NC<=O>'
>>> fcat.GetEntryDescription(obl[-5])
'c<-X>ccc(N)cc'
```

or we can find the fragments that distinguish one molecule from another:

```
>>> combinedFp=fp&(fp^fp2) # can be more efficient than fp&(!fp2)
>>> obl = list(combinedFp.GetOnBits())
>>> fcat.GetEntryDescription(obl[-1])
'cccc(N)cc'
```

Or we can use the bit ranking functionality from the ML.InfoTheory.rdInfoTheory.InfoBitRanker class to identify fragments that distinguish actives from inactives:

```
>>> with Chem.SDMolSupplier('data/bzr.sdf') as suppl:
...     sdms = [x for x in suppl]
>>> fps = [fpgen.GetFPForMol(x,fcat) for x in sdms]
>>> from rdkit.ML.InfoTheory import InfoBitRanker
>>> ranker = InfoBitRanker(len(fps[0]),2)
>>> acts = [x.GetDoubleProp('ACTIVITY') for x in sdms]
>>> for i,fp in enumerate(fps):
```

```
...     act = int(acts[i]>7)
...     ranker.AccumulateVotes(fp,act)
>>> top5 = ranker.GetTopN(5)
>>> for id,gain,n0,n1 in top5:
...     print(int(id),'%.3f'%gain,int(n0),int(n1))


702 0.081 20 17
328 0.073 23 25
341 0.073 30 43
173 0.073 30 43
1034 0.069 5 53
```

The columns above are: bitId, infoGain, nInactive, nActive. Note that this approach isn't particularly effective for this artificial example.

R-Group Decomposition

Let's look at how it works. We'll read in a group of molecules (these were taken ChEMBL), define a core with labelled R groups, and then use the simplest call to do R-group decomposition: rdkit.Chem.rdRGroupDecomposition.RGroupDecompose()

```
>>> from rdkit import Chem
>>> from rdkit.Chem import rdRGroupDecomposition as rdRGD
>>> with Chem.SmilesMolSupplier('data/s1p_chembldoc89753.txt',delimiter=",",
                                smilesColumn=9,nameColumn=10) as suppl:
...     ms = [x for x in suppl if x is not None]
>>> len(ms)
40
>>> core = Chem.MolFromSmarts('[*:1]c1nc([*:2])on1')
>>> res,unmatched = rdRGD.RGroupDecompose([core],ms,asSmiles=True)
>>> unmatched
[]
>>> len(res)
40
>>> res[:2]
[{'Core': 'n1oc([*:2])nc1[*:1]',
  'R1': 'O=C(O)CCCC1NCCOc2c1cccc2[*:1]',
  'R2': 'CC(C)Oc1ccc([*:2])cc1Cl'},
 {'Core': 'n1oc([*:2])nc1[*:1]',
  'R1': 'O=C(O)CCC1NCCOc2c1cccc2[*:1]',
  'R2': 'CC(C)Oc1ccc([*:2])cc1Cl'}]
```

The unmatched return value has the indices of the molecules that did not match a core; in this case there are none. The other result is a list with one dict for each molecule; each dict contains the core that matched the molecule (in this case there was only one) and

the molecule's R groups. As an aside, if you are a Pandas user, it's very easy to get the R-group decomposition results into a DataFrame:

```
>>> import pandas as pd
>>> res,unmatched = rdRGD.RGroupDecompose([core],
                                          ms,
                                          asSmiles=True,
                                          asRows=False)
>>> df= pd.DataFrame(res)
>>> df.head()
```

	Core	R1	R2
0	n1oc([*:2])nc1[*:1]	O=C(O)CCCC1NCCOc2c1cccc2[*:1]	CC(C)Oc1ccc([*:2])cc1Cl
1	n1oc([*:2])nc1[*:1]	O=C(O)CCC1NCCOc2c1cccc2[*:1]	CC(C)Oc1ccc([*:2])cc1Cl
2	n1oc([*:2])nc1[*:1]	O=C(O)CCC1COc2ccc([*:1])cc2CN1	CC(C)Oc1ccc([*:2])cc1Cl
3	n1oc([*:2])nc1[*:1]	O=C(O)CCCC1NCCOc2c1cccc2[*:1]	CC(C)Oc1ncc([*:2])cc1Cl
4	n1oc([*:2])nc1[*:1]	O=C(O)CCCC1NCCOc2c1cccc2[*:1]	CC(C)Oc1ncc([*:2])cc1Cl

It's not necessary to label the attachment points on the core, if you leave them out the code will automatically assign labels:

```
>>> core2 = Chem.MolFromSmarts('c1ncon1')
>>> res,unmatched = rdRGD.RGroupDecompose([core2],ms,asSmiles=True)
>>> res[:2]
[{'Core': 'n1oc([*:1])nc1[*:2]',
  'R1': 'CC(C)Oc1ccc([*:1])cc1Cl',
  'R2': 'O=C(O)CCCC1NCCOc2c1cccc2[*:2]'},
 {'Core': 'n1oc([*:1])nc1[*:2]',
  'R1': 'CC(C)Oc1ccc([*:1])cc1Cl',
  'R2': 'O=C(O)CCC1NCCOc2c1cccc2[*:2]'}]
```

R-group decomposition is actually pretty complex, so there's a lot more there. Hopefully

this is enough to get you started.

Non-Chemical Functionality

Bit vectors

Bit vectors are containers for efficiently storing a set number of binary values, e.g. for fingerprints. The RDKit includes two types of fingerprints differing in how they store the values internally; the two types are easily interconverted but are best used for different purpose:

- SparseBitVects store only the list of bits set in the vector; they are well suited for storing very large, very sparsely occupied vectors like pharmacophore fingerprints. Some operations, such as retrieving the list of on bits, are quite fast. Others, such as negating the vector, are very, very slow.

- ExplicitBitVects keep track of both on and off bits. They are generally faster than SparseBitVects, but require more memory to store.

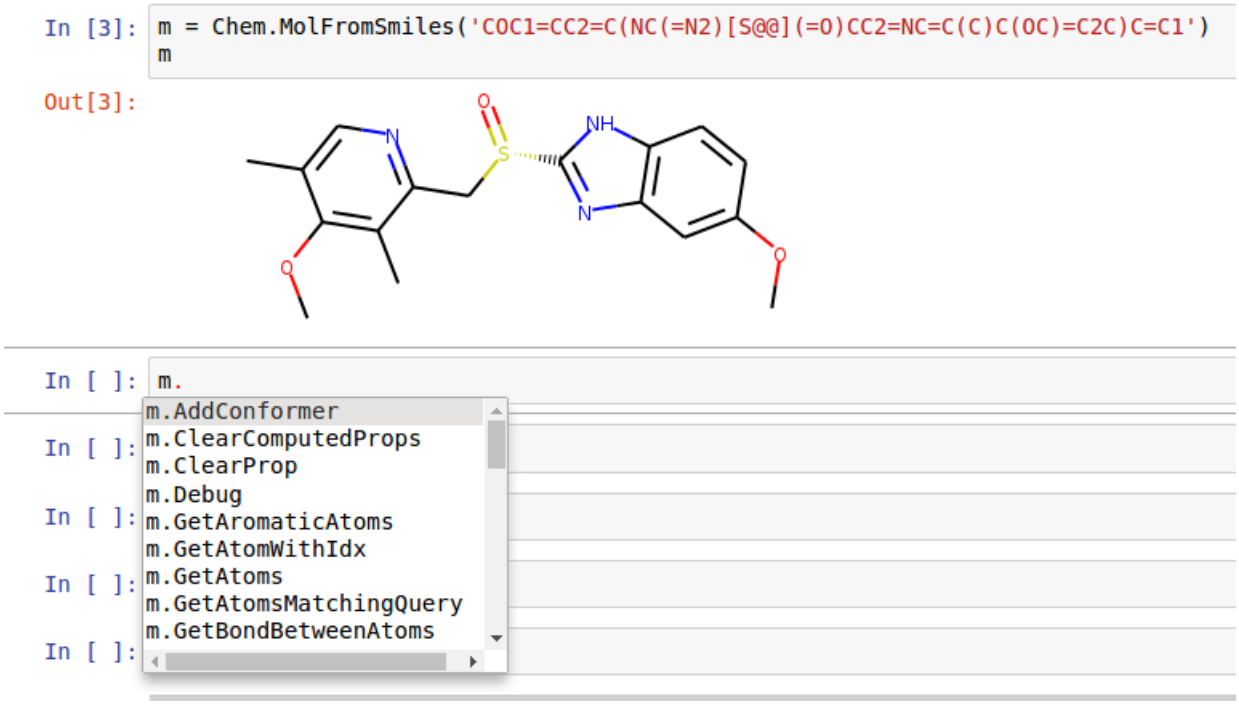
Getting Help

There is a reasonable amount of documentation available within from the RDKit's doc-strings. These are accessible using Python's help command:

```
    >>> m = Chem.MolFromSmiles('Cc1ccccc1')
    >>> m.GetNumAtoms()
    7
    >>> help(m.GetNumAtoms)
'''
Help on method GetNumAtoms:

GetNumAtoms(...) method of rdkit.Chem.rdchem.Mol instance
    GetNumAtoms( (Mol)self [, (int)onlyHeavy=-1 [, (bool)onlyExplicit=True]])
    -> int :
        Returns the number of atoms in the molecule.
          ARGUMENTS:
            - onlyExplicit: (optional) include only explicit atoms
              (atoms in the molecular graph)
                          defaults to 1.
          NOTE: the onlyHeavy argument is deprecated


        C++ signature :
            int GetNumAtoms(...)
'''
    >>> m.GetNumAtoms(onlyExplicit=False)
    15
```

When working in an environment that does command completion or tooltips, one can see

the available methods quite easily. Here's a sample screenshot from within the Jupyter notebook:



Advanced Topics/Warnings

Editing Molecules

Some of the functionality provided allows molecules to be edited "in place":

```
>>> m = Chem.MolFromSmiles('c1ccccc1')
>>> m.GetAtomWithIdx(0).SetAtomicNum(7)
>>> Chem.SanitizeMol(m)
rdkit.Chem.rdmolops.SanitizeFlags.SANITIZE_NONE
>>> Chem.MolToSmiles(m)
'c1ccncc1'
```

Do not forget the sanitization step, without it one can end up with results that look ok (so long as you don't think):

```
>>> m = Chem.MolFromSmiles('c1ccccc1')
>>> m.GetAtomWithIdx(0).SetAtomicNum(8)
>>> Chem.MolToSmiles(m)
'c1ccocc1'
```

but that are, of course, complete nonsense, as sanitization will indicate:

```
>>> Chem.SanitizeMol(m)
Traceback (most recent call last):
  File "/usr/lib/python3.6/doctest.py", line 1253, in __run
    compileflags, 1) in test.globs
  File "<doctest default[0]>", line 1, in <module>
    Chem.SanitizeMol(m)
```

rdkit.Chem.rdchem.KekulizeException: Can't kekulize mol. Unkekulized atoms: 1 2 3 4 5

More complex transformations can be carried out using the rdkit.Chem.rdchem.RWMol class:

```
>>> m = Chem.MolFromSmiles('CC(=O)C=CC=C')
>>> mw = Chem.RWMol(m)
>>> mw.ReplaceAtom(4,Chem.Atom(7))
>>> mw.AddAtom(Chem.Atom(6))
7
>>> mw.AddAtom(Chem.Atom(6))
8
>>> mw.AddBond(6,7,Chem.BondType.SINGLE)
7
>>> mw.AddBond(7,8,Chem.BondType.DOUBLE)
8
>>> mw.AddBond(8,3,Chem.BondType.SINGLE)
9
>>> mw.RemoveAtom(0)
>>> mw.GetNumAtoms()
8
```

The RWMol can be used just like an ROMol:

```
>>> Chem.MolToSmiles(mw)
'O=CC1=NC=CC=C1'
>>> Chem.SanitizeMol(mw)
rdkit.Chem.rdmolops.SanitizeFlags.SANITIZE_NONE
>>> Chem.MolToSmiles(mw)
'O=Cc1ccccn1'
```

The RDKit also has functionality enabling batch edits of molecules which provides a more efficient way to remove multiple atoms or bonds at once.

```
>>> m = Chem.MolFromSmiles('CC(=O)C=CC=C')
>>> mw = Chem.RWMol(m)
>>> mw.BeginBatchEdit()
>>> mw.RemoveAtom(3)
>>> mw.RemoveBond(1,2)  #<- these are the begin and end atoms of the bond
```

None of the changes actually happen until we "commit" them:

```
>>> Chem.MolToSmiles(mw)
'C=CC=CC(C)=O'
>>> mw.CommitBatchEdit()
>>> Chem.MolToSmiles(mw)
'C=CC.CC.O'
```

You can make this more concise using a context manager, which takes care of the commit for you:

```
>>> with Chem.RWMol(m) as mw:
...     mw.RemoveAtom(3)
...     mw.RemoveBond(1,2)
>>> Chem.MolToSmiles(mw)
'C=CC.CC.O'
```

It is even easier to generate nonsense using the RWMol than it is with standard molecules. If you need chemically reasonable results, be certain to sanitize the results.

Miscellaneous Tips and Hints

Chem vs AllChem

The majority of "basic" chemical functionality (e.g. reading/writing molecules, substructure searching, molecular cleanup, etc.) is in the rdkit.Chem module. More advanced, or less frequently used, functionality is in rdkit.Chem.AllChem. The distinction has been made to speed startup and lower import times; there's no sense in loading the 2D to 3D library and force field implementation if one is only interested in reading and writing a couple of molecules. If you find the Chem/AllChem thing annoying or confusing, you can use python's "import ... as ..." syntax to remove the irritation:

```
>>> from rdkit.Chem import AllChem as Chem
>>> m = Chem.MolFromSmiles('CCC')
```

The SSSR Problem

As others have ranted about with more energy and eloquence than I intend to, the definition of a molecule's smallest set of smallest rings is not unique. In some high symmetry molecules, a "true" SSSR will give results that are unappealing. For example, the SSSR for cubane only contains 5 rings, even though there are "obviously" 6. This problem can be fixed by implementing a *small* (instead of *smallest*) set of smallest rings algorithm that returns symmetric results. This is the approach that we took with the RDKit. Because it is sometimes useful to be able to count how many SSSR rings are present in the molecule, there is a rdkit.Chem.rdmolops.GetSSSR() function, but this only returns the SSSR count, not the potentially non-unique set of rings.

List of Available Descriptors

Descriptor/Descriptor Family	Notes	Language
Gasteiger/Marsili Partial Charges	Tetrahedron 36:3219-28 (1980)	C++
BalabanJ	Chem. Phys. Lett. 89:399-404 (1982)	Python
BertzCT	J. Am. Chem. Soc. 103:3599-601 (1981)	Python
Ipc	J. Chem. Phys. 67:4517-33 (1977)	Python
HallKierAlpha	Rev. Comput. Chem. 2:367-422 (1991)	C++
Kappa1 - Kappa3	Rev. Comput. Chem. 2:367-422 (1991)	C++
Phi	New in 2021.03 release Quant. Struct.-Act. Rel. 8:221-224 (1989)	C++
Chi0, Chi1	Rev. Comput. Chem. 2:367-422 (1991)	Python
Chi0n - Chi4n	Rev. Comput. Chem. 2:367-422 (1991)	C++
Chi0v - Chi4v	Rev. Comput. Chem. 2:367-422 (1991)	C++
MolLogP	Wildman and Crippen JCICS 39:868-73 (1999)	C++
MolMR	Wildman and Crippen JCICS 39:868-73 (1999)	C++

Descriptor/Descriptor Family	**Notes**	**Language**
MolWt		C++
ExactMolWt		C++
HeavyAtomCount		C++
HeavyAtomMolWt		C++
NHOHCount		C++
NOCount		C++
NumHAcceptors		C++
NumHDonors		C++
NumHeteroatoms		C++
NumRotatableBonds		C++
NumValenceElectrons		C++
NumAmideBonds		C++
NumAromatic, Saturated, AliphaticRings		C++
NumAromatic, Saturated, AliphaticHetero, Carbo cycles		C++
RingCount		C++
FractionCSP3		C++
NumSpiroAtoms	Number of spiro atoms (atoms shared between rings that share exactly one atom)	C++

Descriptor/Descriptor Family	**Notes**	**Language**
NumBridgeheadAtoms	Number of bridgehead atoms (atoms shared between rings that share at least two bonds)	C++
TPSA	J. Med. Chem. 43:3714-7, (2000) See the section in the RDKit book describing differences to the original publication.	C++
LabuteASA	J. Mol. Graph. Mod. 18:464-77 (2000)	C++
$\mathrm{PEOE}_V SA1 - PEOE_V SA14$	MOE-type descriptors using partial charges and surface area contributions http://www.chemcomp.com/journal/vsadesc.htm	C++
$\mathrm{SMR}_V SA1 - SMR_V SA10$	MOE-type descriptors using MR contributions and surface area contributions http://www.chemcomp.com/journal/vsadesc.htm	C++

Descriptor/Descriptor Family	**Notes**	**Language**
$\text{SlogP}_V SA1 - SlogP_V SA12$	MOE-type descriptors using LogP contributions and surface area contributions http://www.chemcomp.com/journal/vsadesc.htm	C++
$\text{EState}_V SA1 - EState_V SA11$	MOE-type descriptors using EState indices and surface area contributions (developed at RD, not described in the CCG paper)	Python
$\text{VSA}_E State1 - VSA_E State10$	MOE-type descriptors using EState indices and surface area contributions (developed at RD, not described in the CCG paper)	Python
MQNs	Nguyen et al. ChemMedChem 4:1803-5 (2009)	C++
Topliss fragments	implemented using a set of SMARTS definitions in (RD-BASE)/Data/FragmentDescriptors.csv	Python

Descriptor/Descriptor Family	**Notes**	**Language**
Autocorr2D	New in 2017.09 release. Todeschini and Consoni "Descriptors from Molecular Geometry" Handbook of Chemoinformatics https://doi.org/10.1002/9783527618279.ch37	C++
BCUT2D	New in 2020.09 release. Pearlman and Smith in "3D-QSAR and Drug design: Recent Advances" (1997)	C++

List of Available 3D Descriptors

Descriptor/Descriptor Family	**Notes**	**Language**
Plane of best fit (PBF)	Nicholas C. Firth, Nathan Brown, and Julian Blagg, JCIM 52:2516-25	C++
PMI1, PMI2, PMI3	Principal moments of inertia	C++
NPR1, NPR2	Normalized principal moments ratios Sauer and Schwarz JCIM 43:987-1003 (2003) C++	
Radius of gyration	G. A. Arteca "Molecular Shape Descriptors" Reviews in Computational Chemistry vol 9 https://doi.org/10.1002/9780470125861.ch5	C++
Inertial shape factor	Todeschini and Consoni "Descriptors from Molecular Geometry" Handbook of Chemoinformatics https://doi.org/10.1002/9783527618279.ch37	C++
Eccentricity	G. A. Arteca "Molecular Shape Descriptors" Reviews in Computational Chemistry vol 9 https://doi.org/10.1002/9780470125861.ch5	C++

Descriptor/Descriptor Family	**Notes**	**Language**
Asphericity	A. Baumgaertner, "Shapes of flexible vesicles" J. Chem. Phys. 98:7496 (1993) https://doi.org/10.1063/1.464689	C++
Spherocity Index	Todeschini and Consoni "Descriptors from Molecular Geometry" Handbook of Chemoinformatics https://doi.org/10.1002/9783527618279.ch37	C++
Autocorr3D	New in 2017.09 release. Todeschini and Consoni "Descriptors from Molecular Geometry" Handbook of Chemoinformatics https://doi.org/10.1002/9783527618279.ch37	C++
RDF	New in 2017.09 release. Todeschini and Consoni "Descriptors from Molecular Geometry" Handbook of Chemoinformatics https://doi.org/10.1002/9783527618279.ch37	C++

Descriptor/Descriptor Family	**Notes**	**Language**
MORSE	New in 2017.09 release. Todeschini and Consoni "Descriptors from Molecular Geometry" Handbook of Chemoinformatics https://doi.org/10.1002/9783527618279.ch37	C++
WHIM	New in 2017.09 release. Todeschini and Consoni "Descriptors from Molecular Geometry" Handbook of Chemoinformatics https://doi.org/10.1002/9783527618279.ch37 Note insufficient information is available to exactly reproduce values from DRAGON for these descriptors. We believe that this is close.	C++

Descriptor/Descriptor Family	**Notes**	**Language**
GETAWAY	New in 2017.09 release. Todeschini and Consoni "Descriptors from Molecular Geometry" Handbook of Chemoinformatics https://doi.org/10.1002/9783527618279.ch37 **Note** insufficient information is available to exactly reproduce values from DRAGON for these descriptors. We believe that this is close.	C++

RDKit tutorials and video lectures

Introduction

In the evolving landscape of computational chemistry and chemoinformatics, RDKit stands out as a powerful toolkit that empowers researchers and scientists to explore, analyze, and manipulate chemical information. Whether you are a seasoned chemoinformatician or a newcomer to the field, this guide is designed to complement "Introduction to RDKit with Python" with a series of tutorials and exercises. This PDF compiles a collection of tutorials and exercises carefully selected from RDKit Blog webpage to guide you through various aspects of RDKit. Whether you are interested in molecular structure manipulation, substructure searching, or property prediction, these materials aim to provide hands-on experience and practical insights.

Each tutorial includes step-by-step instructions and examples, allowing to follow along and apply RDKit concepts in a practical manner.

The content spans from basic concepts to advanced applications, ensuring that learners of all levels will find valuable insights.

Exercises are designed with real-world scenarios in mind, providing a bridge between theoretical knowledge and practical implementation.

Table of Content

3. **Dealing with multiconformer SD files**
4. **More on constrained embedding**
5. **Clustering conformers**
6. **Generalized substructure search**
7. **Advanced features in SubstrucLibrary**
8. **Searching with generic groups**
9. **Similarity maps with the new drawing code**
10. **3D maximum common substructure**
11. **Using single-molecule reactions**
12. **Highlighting changing atoms and bonds in reactions**
13. **R-Group Decomposition and Highlighting**
14. **RDKit video lectures**

Setting up an RDKit development environment

Setting up an RDKit enviroment is surprisingly straightforward. This tutorial provides a series of step-by-step instructions for getting a local build of the RDKit working on your machine. This is potentially useful if you want to contribute to the RDKit C++ code, make Python contributions (an alternative approach for this is described here) or just try out/work with the pre-release version of the RDKit.

Note: The instructions in this tutorial currently work on Linux and the Mac. This tutorial assumes that you have a local install of conda. That could be miniconda, miniforge, or one of the larger conda installations. If you're just getting started, I would suggest miniforge. You also need to have a working C++17 compiler on your machine. This is not a problem on modern linux distributions where you can install either g++ or clang++ directly from the distribution. On the Mac you need to install the xcode command line tools, the command to do that is:

```
xcode-select --install.
```

Finally, you need a working version of git. If you haven't done this already: there are a number of easy ways to install git, you can decide which makes most sense for you. The simplest is probably to just install it using conda. Now that you've got everything set up, go to the directory where you want to keep the RDKit source code and clone the github repo:

```
git clone https://github.com/rdkit/rdkit.git RDKit_git
```

This will create a directory called RDKit_git with the RDKit source.

Part 1: setup conda environment

Conda makes it very, very easy to install all of the dependencies for building the RDKit, here's the command:

```
mamba create -n py310_rdkit_build -c conda-forge python=3.10 boost-cpp \
boost cairo pandas pillow freetype cmake numpy eigen matplotlib
```

Note that I use *mamba* instead of *conda* for installing things. *mamba* is often significantly faster than *conda*. It's installed by default with miniforge, or you can install it manually with:

```
conda install -c conda-forge mamba
```

Part 2: run cmake and build

Now we're going to actually build the RDKit. These instructions build the code and install it in the RDKit source tree (instead of trying to install it centrally on your computer). From inside the RDKit source directory you checked out above:

```
conda activate py310_rdkit_build
mkdir build_demo
cd build_demo
cmake -DRDK_BUILD_INCHI_SUPPORT=ON -DRDK_BUILD_YAEHMOP_SUPPORT=ON \
-DRDK_BUILD_XYZ2MOL_SUPPORT=ON ..
```

That should run without errors. It'll produce a bunch of warnings, but should end with something like:

```
-- Generating done
-- Build files have been written to: /localhome/glandrum/RDKit_git/build_demo
```

Now actually build it:

```
make -j4 install
```

The -j4 here controls the number of jobs which run simultaneously. If you have a lot of memory on your machine, you can definitely increase the value from 4 to 8 (or whatever). That will take a while and generate a few warnings, but shouldn't have any errors. The last line of the output should be something like this:

```
-- Installing: /localhome/glandrum/RDKit_git/rdkit/RDPaths.py
```

Part 3: setup your environment and then run the tests

At this point you have a completed build of the RDKit that is installed within the source tree. Now we're going to set up your enviroment so that you can actually use that build. Do this from inside the RDKit source directory (not the build directory). On linux:

```
export RDBASE=`pwd`
export PYTHONPATH=$RDBASE
export LD_LIBRARY_PATH=$RDBASE/lib:$CONDA_PREFIX/lib
```

on Mac, that is:

```
export RDBASE=`pwd`
export PYTHONPATH=$RDBASE
export DYLD_LIBRARY_PATH=$RDBASE/lib:$CONDA_PREFIX/lib
```

Now you can run the tests:

```
cd build_demo
ctest -j4 -output-on-failure
```

This will take a bit of time and generate a lot of output, but the last bit should look something like this:

```
100% tests passed, 0 tests failed out of 219


Total Test time (real) =  63.92 sec
```

That's it! You have now built the RDKit and can work with it locally. If you make changes to the C++ code and want to test them out, you just need to run:

```
make -j4 install
```

After making Python changes you don't need to do anything in order to have those changes available. I do, however, strongly suggest running the tests again:

```
ctest -j4 --output-on-failure
```

after making any change. It's good to find out quickly if you've broken anything!

Part 4: Using the environment

In order to use the code you just need to set up your environment by repeating the export commands shown above. I find the following aliases really useful on linux (the Mac form just needs LD_LIBRARY_PATH changed to DYLD_LIBRARY_PATH).

```
alias remote_rdkit='unset RDBASE;export PYTHONPATH="";export LD_LIBRARY_PATH=""'
alias local_rdkit='conda activate py310_rdkit_build;export \
    RDBASE="/localhome/glandrum/RDKit_git";export PYTHONPATH="$RDBASE";\
    export LD_LIBRARY_PATH="$RDBASE/lib:$CONDA_PREFIX/lib"'
alias rdkit_vers='python -c "import rdkit;print(rdkit.__version__,rdkit.__file__)"'
```

local_rdkit sets up the shell to use my local RDKit build. remote_rdkit clears that stuff out so that you can use a conda rdkit install rdkit_vers shows the current rdkit version and where the files are coming from.

Working with 3D Conformers

The RDKit stores atomic coordinates in Conformer objects which are attached to the corresponding molecules. In this tutorial post we're going to look in detail at the way conformers are stored and ways to work with them.

Part 1. Create a Molecule object

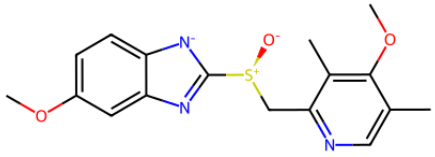
```
>>> from rdkit import Chem
>>> from rdkit.Chem.Draw import IPythonConsole
>>> IPythonConsole.ipython_3d = True
>>> import py3Dmol
>>> from rdkit.Chem import rdDepictor
>>> from rdkit.Chem import rdDistGeom
>>> import rdkit
>>> print(rdkit.__version__)
2022.09.4
```

A molecule constructed from SMILES has no conformer information:

```
>>> smiels = 'COc1ccc2[n-]c([S@@+]([O-])Cc3ncc(C)c(OC)c3C)nc2c1'
>>> esomeprazole = Chem.MolFromSmiles(smiles)
>>> esomeprazole.GetNumConformers()
0
```

An aside: when you display a molecule without conformers in Jupyter, a 2D conformer is generated so that there's something to display:

```
>>> esomeprazole
```



But this conformer is just temporary; the molecule itself is not modified:

```
>>> esomeprazole.GetNumConformers()
0
```

When we ask to have 2D coordinates generated, they are added as a conformer:

```
>>> rdDepictor.Compute2DCoords(esomeprazole)
>>> esomeprazole.GetNumConformers()
1
```

And that conformer is flagged as not being 3D:

```
>>> esomeprazole.GetConformer().Is3D()
False
```

Generating a 3D structure also results in a conformer being added to the molecule (by default any existing conformers are cleared):

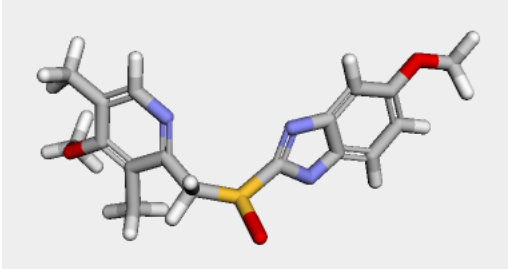
```
>>> esomeprazole = Chem.AddHs(esomeprazole)
>>> rdDistGeom.EmbedMolecule(esomeprazole)
>>> print(esomeprazole.GetNumConformers(),esomeprazole.GetConformer().Is3D())
1 True
[09:59:21] UFFTYPER: Unrecognized charge state for atom: 8
```

Molecules can have more than one conformer attached. This is what happens whenever we ask for multiple conformers to be generated:

```
>>> rdDistGeom.EmbedMultipleConfs(esomeprazole,10, randomSeed=0xf00d)
>>> print(esomeprazole.GetNumConformers(),esomeprazole.GetConformer().Is3D())
[09:59:22] UFFTYPER: Unrecognized charge state for atom: 8
10 True
```

Calling GetConformer() without arguments on a molecule with multiple conformers (as we did above) returns the default conformer. This is also what most RDKit operations that work with conformers do:

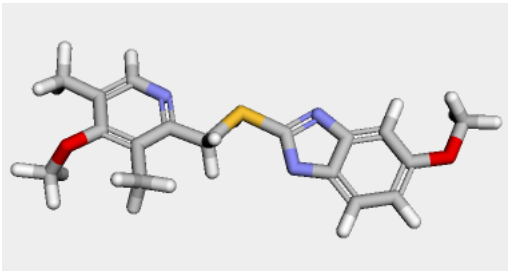
```
>>> IPythonConsole.drawMol3D(esomeprazole)
```



```
>>> print(Chem.MolToV3KMolBlock(esomeprazole)[:200])
     RDKit          3D


  0  0  0  0  0  0  0  0  0  0999 V3000
M  V30 BEGIN CTAB
M  V30 COUNTS 42 44 0 0 0
M  V30 BEGIN ATOM
M  V30 1 C 7.673833 1.902627 0.749932 0
M  V30 2 O 6.455003 1.586360 1.35
```

But functions which use conformer information also generally take an optional confId argument that allows you to change which conformer is used:

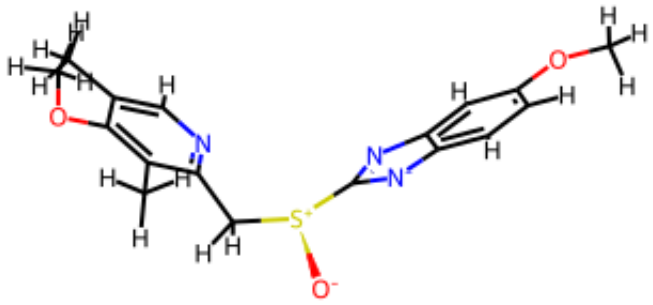
```
>>> IPythonConsole.drawMol3D(esomeprazole,confId=2)
```



```
>>> print(Chem.MolToV3KMolBlock(esomeprazole,confId=2)[:200])
     RDKit           3D


  0  0  0  0  0  0  0  0  0  0999 V3000
M  V30 BEGIN CTAB
M  V30 COUNTS 42 44 0 0 0
M  V30 BEGIN ATOM
M  V30 1 C 8.236088 0.676672 -0.133971 0
M  V30 2 O 7.679673 -0.571643 0.
```

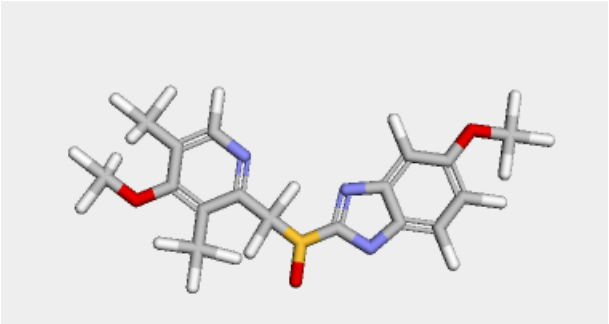
As an aside: the Is3D flag on conformers is used by the RDKit's jupyter integration to decide how to show the molecule. We can fool it by marking a 3D conformer as 2D:

```
>>> tmol = Chem.Mol(esomeprazole)
>>> tmol.GetConformer().Set3D(False)
>>> tmol
```



or a 2D conformer as 3D:

```
>>> rdDepictor.Compute2DCoords(tmol)
>>> tmol.GetConformer().Set3D(True)
>>> tmol
```



Part 2. Conformer IDs

Each conformer has an ID associated with it:

```
>>> print([x.GetId() for x in esomeprazole.GetConformers()])
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

And there's no guarantee that the IDs are consecutive:

```
>>> cp = Chem.Mol(esomeprazole)
>>> cp.RemoveConformer(0)
>>> cp.RemoveConformer(3)
>>> cp.RemoveConformer(7)
>>> print([x.GetId() for x in cp.GetConformers()])
[1, 2, 4, 5, 6, 8, 9]
```

The default conformer is just the first one:

```
>>> esomeprazole.GetConformer().GetId(),cp.GetConformer().GetId()
(0, 1)
```

If we ask for a conformer Id which doesn't exist, we get an error

```
>>> cp.GetConformer(12)
ValueError: Bad Conformer Id
```

So it's usually better to loop over conformers by using *for conf in m.GetConformers()* than to do something like *for confid in range(m.GetNumConformers())*

Part 3. Getting atom positions

We can get the position of an individual atom using the GetAtomPosition() method:

```
>>> conf = esomeprazole.GetConformer()
>>> pos = conf.GetAtomPosition(0)
>>> pos
<rdkit.Geometry.rdGeometry.Point3D at 0x24549d77940>
```

Point3D objects have x, y, an z members:

```
>>> pos.x
7.673833423925606
```

We can also treat them as vectors:

```
>>> pos[0]
7.673833423925606
```

or convert them to a list:

```
>>> list(pos)
[7.673833423925606, 1.9026270201484923, 0.7499323617732389]
```

If you want to work with all of the atom positions at once, you can get the full set of positions as a numeric python array:

```
>>> conf = esomeprazole.GetConformer()
>>> conf.GetPositions()


array([[ 7.67383342,  1.90262702,  0.74993236],
       [ 6.45500303,  1.5863599 ,  1.35650536],
       [ 5.47994258,  0.87148768,  0.70572686],
       [ 5.73975972,  0.49700014, -0.60402982],
       [ 4.78365557, -0.22724183, -1.28080521],
```

```
[ 3.59642905, -0.54463893, -0.59986585],
[ 2.59405829, -1.21382165, -1.03927358],
[ 1.57249064, -1.34923356, -0.0664238 ],
[ 0.23760111, -2.18813362, -0.61069031],
[ 0.81652921, -3.85758446, -0.60953112],
[-1.27391039, -2.26700639,  0.09012225],
[-2.10815098, -1.11798724,  0.39836845],
[-1.98174435, -0.40660263,  1.56568982],
[-2.94381966,  0.4469877 ,  1.9742195 ],
[-4.10302205,  0.68692145,  1.27699325],
[-5.19127878,  1.57733968,  1.7705111 ],
[-4.26093798, -0.01014656,  0.07490913],
[-5.42572563,  0.23147474, -0.6552053 ],
[-5.42087157,  1.29218749, -1.61825447],
[-3.29351791, -0.86279333, -0.31409244],
[-3.3729589 , -1.58163605, -1.64292335],
[ 2.09334884, -0.69551121,  0.96089041],
[ 3.32320448, -0.18671147,  0.68892685],
[ 4.28153998,  0.53232818,  1.34410496],
[ 8.09389757,  0.95596456,  0.26748966],
[ 7.65061035,  2.62967184, -0.06019159],
[ 8.38623492,  2.21563152,  1.56819868],
[ 6.66853902,  0.74454419, -1.13424145],
[ 4.89232116, -0.56528193, -2.30134486],
[-1.97542789, -3.01240938, -0.43608787],
[-1.27389641, -2.83764033,  1.13153208],
[-2.83784116,  0.99189563,  2.93071709],
[-5.86663733,  0.96082859,  2.44934068],
```

```
       [-5.84018865,  1.83545526,  0.9297767 ],
       [-4.86235377,  2.50117065,  2.2287445 ],
       [-4.78779089,  1.06132465, -2.50433141],
       [-4.99817085,  2.23486583, -1.20407518],
       [-6.43595476,  1.42487573, -2.00400931],
       [-4.24971742, -1.21643589, -2.21758627],
       [-2.46593717, -1.22685901, -2.26727455],
       [-3.47585534, -2.65007473, -1.58068986],
       [ 4.1067109 ,  0.83680778,  2.34637208]])
```

Part 4. Adding/removing conformers

I already showed how to remove conformer in the section on conformer IDs, but you can also add conformers to a molecule. Let's start by getting a conformer which is a canonical orientation: centered at the origin and with its principle axes aligned with the cartesian axes:

```
>>> from rdkit.Chem import rdMolTransforms
# GetConformer() returns a reference to the existing conformer,
  we want a copy:
>>> conf = Chem.Conformer(esomeprazole.GetConformer(0))
>>> rdMolTransforms.CanonicalizeConformer(conf)
```

Make a copy of the molecule and add a 2D conformer, by default Compute2DCoords() clears the existing conformers

```
>>> cp = Chem.Mol(esomeprazole)
>>> rdDepictor.Compute2DCoords(cp)
0
```

Now add the that canonical conformer and set its ID so that it is unique. The return value is the new conformer ID:

```
>>> cp.AddConformer(conf,assignId=True)
1
>>> cp.GetConformer().GetPositions()[:3]
array([[8.41075857, 1.90814121, 0.        ],
       [6.92408705, 2.10766008, 0.        ],
       [6.00796287, 0.91992422, 0.        ]])

>>> cp.GetConformer(1).GetPositions()[:3]
array([[ 7.39395019,  1.81765991, -0.52124964],
       [ 6.17231368,  1.90827963,  0.15172549],
       [ 5.15244456,  1.01460864, -0.06395248]])
```

Part 5. Getting a new molecule with just one conformer

There are often times you want to copy a multi-conformer molecule but just keep one of the conformers. The standard way of copying RDKit molecules will copy all conformers:

```
>>> cp = Chem.Mol(esomeprazole)
>>> cp.GetNumConformers()
10
```

But we can tell it to just copy one:

```
>>> cp = Chem.Mol(esomeprazole,confId=5)
>>> print(cp.GetNumConformers(), cp.GetConformer().GetId())
1 5
```

Dealing with multiconformer SD files

A recurring question is how to save and share multi-conformer molecules. The easiest (and fastest) way to do this in the RDKit is to just pickle the molecules. It's not significantly more difficult to use rdMolInterchange.MolToJSON() to serialize the molecules as JSON. Neither of these methods work if you want to work with other tools, so we're frequently stuck with using something like SD files.

This tutorial is about how to create and work with multi-conformer SD files as well as multi-molecule, multi-conformer SD files with the RDKit.

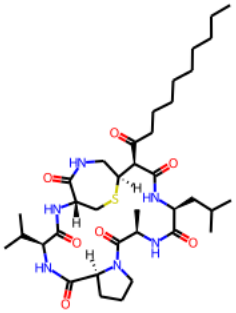
```
>>> from rdkit import Chem
>>> from rdkit.Chem import Draw
>>> from rdkit.Chem import rdDepictor
>>> rdDepictor.SetPreferCoordGen(True)
>>> import rdkit
>>> print(rdkit.__version__)
2022.03.5
```

Part 1. Generate conformers

Let's start by generating conformers for doravirine, lenalidomide, and mutanobactin:

```
>>> dorav = Chem.MolFromSmiles('Cn1c(n[nH]c1=O)Cn2ccc(c(c2=O) \
                               Oc3cc(cc(c3)Cl)C#N)C(F)(F)F')
>>> lenal = Chem.MolFromSmiles('O=C1NC(=O)CCC1N3C(=O)c2cccc(c2C3)N')
>>> mutanob = Chem.MolFromSmiles('CCCCCCCCCC(=O)[C@@H]1[C@@H]2CNC(=O)[C@H](CS2) \
                                 NC(=O)[C@@H](NC(=O)[C@@H]3CCCN3C(=O)[C@H] \
                                 (NC(=O)[C@@H](NC1=O)CC(C)C)C)C(C)C')
>>> Draw.MolsToGridImage([dorav,lenal,mutanob],subImgSize=(300,250))
```





```
>>> from rdkit.Chem import rdDistGeom
>>> ps = rdDistGeom.ETKDGv3()
>>> ps.randomSeed = 0xf00d
>>> ps.numThreads = 4
>>> ps.pruneRmsThresh = 0.5
>>> dorav = Chem.AddHs(dorav)
>>> lenal = Chem.AddHs(lenal)
>>> mutanob = Chem.AddHs(mutanob)
>>> d_cids = rdDistGeom.EmbedMultipleConfs(dorav,200,ps)
>>> l_cids = rdDistGeom.EmbedMultipleConfs(lenal,200,ps)
>>> m_cids = rdDistGeom.EmbedMultipleConfs(mutanob,200,ps)
>>> print(dorav.GetNumConformers(),
          lenal.GetNumConformers(),
          mutanob.GetNumConformers())
58 7 200
```

Now let's start with a digression and quickly demo the two easy ways of storing and retrieving this multi-conformer molecule: pickling and using JSON:

```
>>> import pickle
>>> pkl = pickle.dumps(dorav)
>>> nmol = pickle.loads(pkl)
>>> nmol.GetNumConformers()
```

```
58
>>> from rdkit.Chem import rdMolInterchange
>>> mjs = rdMolInterchange.MolToJSON(dorav)
>>> nmol = rdMolInterchange.JSONToMols(mjs)[0]
>>> nmol.GetNumConformers()
58
```

Just out of curiosity let's see how long those take:

```
>>> %timeit pickle.dumps(dorav)
162 µs ± 1.33 µs per loop (mean ± std. dev. of 7 runs, 10,000 loops each)
>>> %timeit pickle.loads(pkl)
175 µs ± 1.44 µs per loop (mean ± std. dev. of 7 runs, 10,000 loops each)
>>> %timeit rdMolInterchange.MolToJSON(dorav)
646 µs ± 12.7 µs per loop (mean ± std. dev. of 7 runs, 1,000 loops each)
>>> %timeit rdMolInterchange.JSONToMols(mjs)[0]
252 µs ± 3.5 µs per loop (mean ± std. dev. of 7 runs, 1,000 loops each)
```

Part 2. Write to an SDF "file"

```
>>> from io import StringIO
>>> sio = StringIO()
>>> w = Chem.SDWriter(sio)
>>> for cid in d_cids:
...     w.write(dorav,confId=cid)
>>> w.flush()
>>> sdf = sio.getvalue()
>>> sdf[:100]
'\n     RDKit          3D\n\n 40 42  0  0  0  0  0  0  0  0999 V2000\n \
 -3.5730   -0.0689    0.9965 C   '
```

```
>>> from io import BytesIO
>>> bio = BytesIO(sdf.encode())
>>> suppl = Chem.ForwardSDMolSupplier(bio)
>>> ref = next(suppl)
>>> for mol in suppl:
...     ref.AddConformer(mol.GetConformer(),assignId=True)
>>> print(ref.GetNumConformers())
58
```

Again: how long does this take?

```
>>> %timeit sio=StringIO();w=Chem.SDWriter(sio);
        [w.write(dorav,confId=cid) for cid in d_cids];
         w.flush();sdf=sio.getvalue()
9.85 ms ± 213 µs per loop (mean ± std. dev. of 7 runs, 100 loops each)
>>> %timeit bio=BytesIO(sdf.encode());
        suppl=Chem.ForwardSDMolSupplier(bio);ref=next(suppl);
        [ref.AddConformer(m.GetConformer(),assignId=True) for m in suppl]
13.1 ms ± 250 µs per loop (mean ± std. dev. of 7 runs, 100 loops each)
```

Part 3. Handling SDFs which contain multiple conformers of more than one molecule

```
>>> from io import StringIO
>>> sio = StringIO()
>>> w = Chem.SDWriter(sio)
>>> mol = Chem.Mol(dorav)
>>> mol.SetProp('_Name','doravirine')
>>> for cid in d_cids:
...     w.write(mol,confId=cid)
```

```
>>> mol = Chem.Mol(lenal)
>>> mol.SetProp('_Name','lenalidomide')
>>> for cid in l_cids:
...     w.write(mol,confId=cid)
>>> mol = Chem.Mol(mutanob)
>>> mol.SetProp('_Name','mutanobactin')
>>> for cid in m_cids:
...    w.write(mol,confId=cid)
>>> w.flush()
>>> multimol_sdf = sio.getvalue()
```

And here's a function to return the molecules from a multi-molecule, multi-conformer supplier:

```
def mols_from_multimol_multiconf_supplier(supplier,propertyName='_Name'):
    mol = None
    for itm in supplier:
        if itm is None:
            continue
        if mol is None:
            mol = itm
            refVal = mol.GetProp(propertyName)
            continue
        pVal = itm.GetProp(propertyName)
        if pVal == refVal:
            mol.AddConformer(itm.GetConformer(),assignId=True)
        else:
# we're done with the last molecule, so let's restart the next one
            res = mol
            mol = itm
```

```
            refVal = pVal
            yield res

    yield mol
```

Now try that out:

```
>>> from io import BytesIO
>>> bio = BytesIO(multimol_sdf.encode())
>>> suppl = Chem.ForwardSDMolSupplier(bio)
>>> ms = [x for x in mols_from_multimol_multiconf_supplier(suppl)]
>>> print([m.GetNumConformers() for m in ms])
[58, 7, 200]
```

By default the function uses the molecule names, but we can use other property names if we want:

```
>>> from io import StringIO
>>> sio = StringIO()
>>> w = Chem.SDWriter(sio)
>>> mol = Chem.Mol(dorav)
>>> mol.SetProp('molecule_id','doravirine')
>>> for cid in d_cids:
...     w.write(mol,confId=cid)
>>> mol = Chem.Mol(lenal)
>>> mol.SetProp('molecule_id','lenalidomide')
>>> for cid in l_cids:
...     w.write(mol,confId=cid)
>>> mol = Chem.Mol(mutanob)
>>> mol.SetProp('molecule_id','mutanobactin')
>>> for cid in m_cids:
```

```
...     w.write(mol,confId=cid)
>>> w.flush()
>>> multimol_sdf = sio.getvalue()
>>> bio = BytesIO(multimol_sdf.encode())
>>> suppl = Chem.ForwardSDMolSupplier(bio)
>>> ms = [x for x in mols_from_multimol_multiconf_supplier(
                    suppl,propertyName='molecule_id')]
>>> print([m.GetNumConformers() for m in ms])
[58, 7, 200]
```

More on Constrained embedding

This tutorial focus on generating conformers where some atomic coordinates are constrained.

The Python function AllChem.ConstrainedEmbed() can be used to generate conformers where the positions of a set of atoms are constrained to match the coordinates of a template molecule.

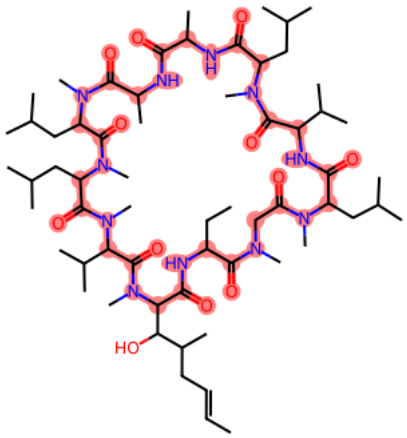
```
>>> from rdkit import Chem
>>> from rdkit.Chem.Draw import IPythonConsole
>>> from rdkit.Chem.Draw import rdDepictor
>>> rdDepictor.SetPreferCoordGen(True)
>>> IPythonConsole.molSize = (350,350)
>>> from rdkit.Chem import AllChem
>>> import math
>>> import rdkit
>>> print(rdkit.__version__)
2022.09.4
```

We're going to work with cyclosporine here, since it's delightfully complicated. In this case we're going to ignore atomic stereochemistry in order to speed the conformer generation up (the RDKit tends to take a while to generate conformers for molecules with a large number of stereocenters).

We'll also define the macrocylce as the core; this is what we're going to use to provide constraints.

```
>>> m = Chem.MolFromSmiles('C/C=C/CC(C)C(O)C1C(=O)NC(CC)C(=O)N(C)CC(=O)N \
                           (C)C(CC(C)C)C(=O)NC(C(C)C)C(=O)N(C)C(CC(C)C) \
                           C(=O)NC(C)C(=O)NC(C)C(=O)N(C)C(CC(C)C)C(=O)N \
                           (C)C(CC(C)C)C(=O)N(C)C(C(C)C)C(=O)N1C')
>>> rdDepictor.Compute2DCoords(m)
```

```
>>> core = Chem.MolFromSmiles('C1C(=O)NCC(=O)NCC(=O)NCC(=O)NCC(=O)NCC(=O) \
                              NCC(=O)NCC(=O)NCC(=O)NCC(=O)NCC(=O)N1')
>>> m.GetSubstructMatch(core)
>>> m
```

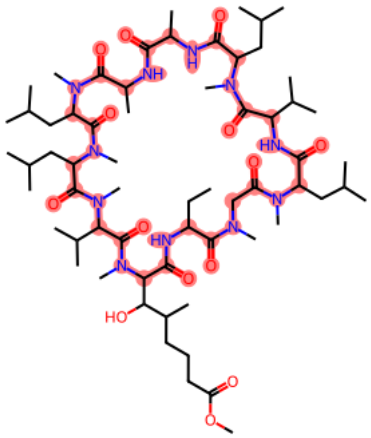


Start by generating a conformer for cyclosporine itself.

```
>>> mh = Chem.AddHs(m)
>>> AllChem.EmbedMolecule(mh,randomSeed=0xf00d)
0
```

For the other molecule, we'll take a molecule I pulled from ChEMBL that includes the core (stereochemistry has been removed here as well).

```
>>> smi = 'CCC1NC(=O)C(C(O)C(C)CCCC(=O)OC)N(C)C(=O)C(C(C)C)NC)C(=O)C(CC \
           (C)C)N(C)C(=O)C(CC(C)C)N(C)C(=O)C(C)NC(=O)C(C)NC(=O)C(CC(C)C) \
           N(C)C(=O)C(C(C)C)NC(=O)C(CC(C)C)N(C)C(=O)CN(C)C1=O'
>>> newm = Chem.MolFromSmiles(smi)
>>> rdDepictor.Compute2DCoords(newm)
>>> newm.GetSubstructMatch(core)
>>> newm
```



Generate an unconstrained conformer for that:

```
>>> newmh = Chem.AddHs(newm)
>>> AllChem.EmbedMolecule(newmh,randomSeed=0xf00d)
0
```

Now calculate the RMSD between the core atoms in cylosporine and in our test molecule.

```
>>> m_match = mh.GetSubstructMatch(core)
>>> newm_match = newmh.GetSubstructMatch(core)
>>> delta2 = 0.0
>>> for mi,newmi in zip(m_match,newm_match):
...     d = (mh.GetConformer().GetAtomPosition(mi) -
             newmh.GetConformer().GetAtomPosition(newmi)).LengthSq()
...     delta2 += d
>>> print('core RMSD:',math.sqrt(delta2/len(m_match)))
core RMSD: 7.101400868778492
```

There's no reason to expect this RMSD to be anything other than huge: we're using different conformer of a flexible core and haven't aligned them to each other.

We can go ahead and do an alignmeent and see how that affects the RMSD:

```
>>> AllChem.AlignMol(newmh,mh,atomMap = list(zip(newm_match,m_match)))
>>> delta2 = 0.0
```

```
>>> for mi,newmi in zip(m_match,newm_match):
...     d = (mh.GetConformer().GetAtomPosition(mi) -
          newmh.GetConformer().GetAtomPosition(newmi)).LengthSq()
...     delta2 += d
>>> print('aligned core RMSD:',math.sqrt(delta2/len(m_match)))
aligned core RMSD: 2.067717832557964
```

Again, it's not surprising that this is a large RMSD: the core is quite flexible and we haven't constrained it at all.

Adding constraints is the point of this blog post though, so let's move onto that. AllChem.EmbedMolecule() can be given constraints for the positions of certain atoms using the coordMap argument, which expects a dictionary that provides a Point3D for each atom that should have a fixed position.

```
>>> cmap = {newm_match[i]:mh.GetConformer().GetAtomPosition(m_match[i])
           for i in range(len(m_match))}
>>> AllChem.EmbedMolecule(newmh,randomSeed=0xf00d,coordMap=cmap)
>>> delta2 = 0.0
>>> for mi,newmi in zip(m_match,newm_match):
...     d = (mh.GetConformer().GetAtomPosition(mi) -
          newmh.GetConformer().GetAtomPosition(newmi)).LengthSq()
...     delta2 += d
>>> print('core RMSD:',math.sqrt(delta2/len(m_match)))
core RMSD: 2.5231352176127078
```

Wait... what happened here? Shouldn't this number be smaller since we introduced constraints?

The function AllChem.EmbedMolecule() uses the coordinates provided in coordMap to set elements of the distance bounds matrix that is used to generate conformers (details about the RDKit's distance-geometry-based conformer generator are in the documentation).

This results in conformers where the distances between the atoms in the conformer closely match the corresponding distances in the coordMap.

Note that, because the coordinates are being constrained using the distances between them, you should expect rigid shifts of the core atoms relative to the constraints. This is solveable by aligning the core of the test molecule to the core of the reference:

```
>>> AllChem.AlignMol(newmh,mh,atomMap = list(zip(newm_match,m_match)))
>>> delta2 = 0.0
>>> for mi,newmi in zip(m_match,newm_match):
...     d = (mh.GetConformer().GetAtomPosition(mi) -
            newmh.GetConformer().GetAtomPosition(newmi)).LengthSq()
...     delta2 += d
>>> print('aligned core RMSD:',math.sqrt(delta2/len(m_match)))
aligned core RMSD: 0.9007025663628238
```

Note that the output coordinates don't match the constraint coordinates exactly. This will almost always be the case; they should be close, but some differences are, unforunately, expected due to the nature of the algorithm.

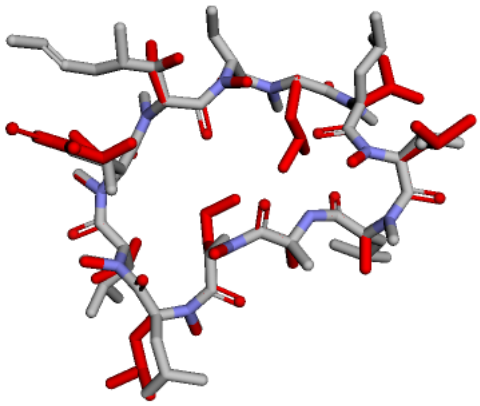
An alternative is to use random coordinate embedding instead of the usual distance-bounds embedding to generate the initial coordinates in the conformer generation. When we do this it's possible to get exact coordinate matches of the core and no alignment is necessary. Random-coordinate embedding is not the default because the current RDKit implementation tends to be slower than the other approach.

```
>>> AllChem.EmbedMolecule(newmh,randomSeed=0xf00d,
                          coordMap=cmap,useRandomCoords=True)
>>> delta2 = 0.0
>>> for mi,newmi in zip(m_match,newm_match):
...     d = (mh.GetConformer().GetAtomPosition(mi) -
             newmh.GetConformer().GetAtomPosition(newmi)).LengthSq()
```

```
...     delta2 += d
>>> print('core RMSD:',math.sqrt(delta2/len(m_match)))
core RMSD: 0.0
```

It's worth taking a look at the conformers of our molecules. We'll do that using py3Dmol. This code snippet uses a bit of convenience functionality that the RDKit's IPythonConsole provides, but it also demonstrates how to draw molecules with different colors.

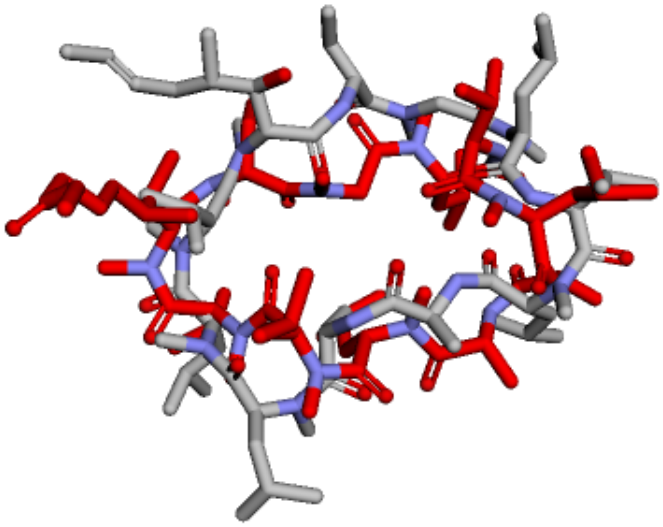
```
>>> import py3Dmol
def drawit(ms, p=None, confId=-1, removeHs=True,
           colors=('cyanCarbon','redCarbon','blueCarbon')):

    if p is None:
        p = py3Dmol.view(width=400, height=400)
    p.removeAllModels()
    for i,m in enumerate(ms):
        if removeHs:
            m = Chem.RemoveHs(m)
        IPythonConsole.addMolToView(m,p,confId=confId)
        p.setStyle({'model':-1,},
                           {'stick':{'colorscheme':colors[i%len(colors)]}})
    p.zoomTo()
    return p.show()

drawit((mh,newmh))
```



Here one molecule is drawn with cyan bonds and the other with red bonds.

Extra: minimize with constraints

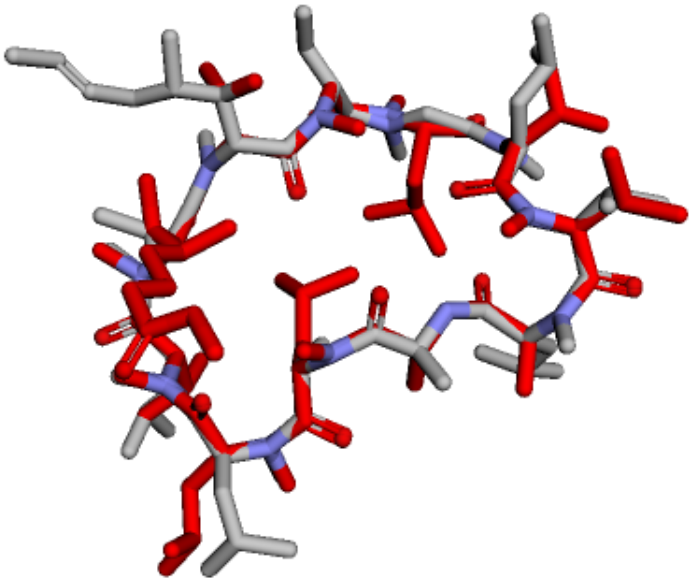
```
>>> from rdkit.Chem import rdForceFieldHelpers
>>> mcp = Chem.Mol(newmh)
>>> mmffps = rdForceFieldHelpers.MMFFGetMoleculeProperties(mcp)
>>> ff = rdForceFieldHelpers.MMFFGetMoleculeForceField(mcp,mmffps)
>>> maxIters = 10
>>> while ff.Minimize(maxIts=1000) and maxIters>0:
...     maxIters -= 1


>>> delta2 = 0.0
>>> for mi,newmi in zip(m_match,newm_match):
...     d = (mh.GetConformer().GetAtomPosition(mi) -
             mcp.GetConformer().GetAtomPosition(newmi)).LengthSq()
...     delta2 += d
>>> print('core RMSD:',math.sqrt(delta2/len(m_match)))
core RMSD: 2.313775204899967
>>> drawit((mh,mcp))
```



That's messed up the core coordinates... we can fix that by adding some position constraints to the force field:

```
>>> mcp = Chem.Mol(newmh)
>>> mmffps = rdForceFieldHelpers.MMFFGetMoleculeProperties(mcp)
>>> ff = rdForceFieldHelpers.MMFFGetMoleculeForceField(mcp,mmffps)
>>> for atidx in newm_match:
...     ff.MMFFAddPositionConstraint(atidx,0.05,200)
>>> maxIters = 10
>>> while ff.Minimize(maxIts=1000) and maxIters>0:
...    maxIters -= 1


>>> delta2 = 0.0
>>> for mi,newmi in zip(m_match,newm_match):
...     d = (mh.GetConformer().GetAtomPosition(mi) -
            mcp.GetConformer().GetAtomPosition(newmi)).LengthSq()
...     delta2 += d
>>> print('core RMSD:',math.sqrt(delta2/len(m_match)))
core RMSD: 0.12659186987312163
```

```
drawit((mh,mcp))
```



Clustering Conformers

With this tutorial we will go through how to cluster conformers. One common workflow in conformational analysis is to generate a bunch of conformers for a molecule and then find a representative subset by clustering them.

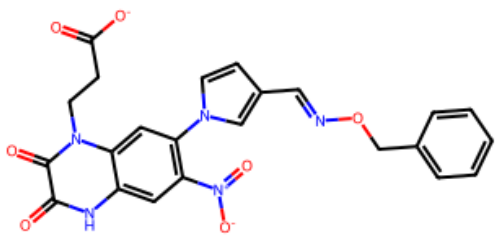
The RDKit has everything required to do this, but there's not all that much info out there showing how to do it.

```
>>> from rdkit.Chem import Draw
>>> from rdkit.Chem.Draw import IPythonConsole
>>> from rdkit import Chem
>>> from rdkit.Chem import rdDistGeom
>>> from rdkit.Chem import rdMolAlign
>>> import rdkit
>>> print(rdkit.__version__)
2022.09.4
```

Part 1. Generate conformers

Start by constructing a molecule and then generating a set of 300 conformers for it using ETKDGv3.

```
>>> m = Chem.MolFromSmiles('O=C([O-])CCn1c(=O)c(=O)[nH]c2cc([N+] \
                           (=O)[O-])c(-n3ccc(C=NOCc4ccccc4)c3)cc21')
>>> m
```



```
>>> mh = Chem.AddHs(m)
>>> ps = rdDistGeom.ETKDGv3()
>>> ps.randomSeed = 0xd06f00d
>>> ps.numThreads = 10
>>> cids = rdDistGeom.EmbedMultipleConfs(mh,300,ps)
>>> len(cids)
300
```

Remove Hs from the molecule at this point because they aren't particularly informative for the rest of the analysis and they just make things more difficult

```
>>> m3d = Chem.RemoveHs(mh)
```

Part 2. Direct alignment vs best alignment

Find a pair of conformers with a decent size mismatch between the direct alignment (which does not take symmetry into account) and the best alignment (which does):

```
>>> maxd = -100
>>> for j in range(0,5):
...     for i in range(j,len(cids)):
...         d1 = rdMolAlign.AlignMol(m3d,m3d,prbCid=cids[i],refCid=cids[j])
...         d2 = rdMolAlign.GetBestRMS(m3d,m3d,prbId=cids[i],refId=cids[j])
...         delt = d1-d2
...         if delt<-1e-5:
...             print(f'ooops, {i}, {delt}')
...         if delt>maxd:
...             maxd = delt
...             maxi = i
...             maxj = j
>>> d1 = rdMolAlign.AlignMol(m3d,m3d,prbCid=cids[maxi],refCid=cids[maxj])
```

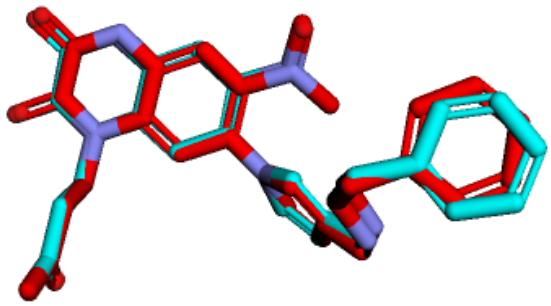
```
>>> d2 = rdMolAlign.GetBestRMS(m3d,m3d,prbId=cids[maxi],refId=cids[maxj])
>>> print(maxi,maxd,d1,d2)
282 0.703433090406554 1.0038182177331938 0.3003851273604343
```

Now show those two conformers:

```
>>> import py3Dmol
def drawit(m, cids=[-1], p=None, removeHs=True,
           colors=('cyanCarbon','redCarbon',
                   'blueCarbon','magentaCarbon',
                   'whiteCarbon','purpleCarbon')):
    if removeHs:
        m = Chem.RemoveHs(m)
    if p is None:
        p = py3Dmol.view(width=400, height=400)
    p.removeAllModels()
    for i,cid in enumerate(cids):
        IPythonConsole.addMolToView(m,p,confId=cid)
    for i,cid in enumerate(cids):
        p.setStyle({'model':i,},
                   {'stick':{'colorscheme':colors[i%len(colors)]}})
    p.zoomTo()
    return p.show()
```

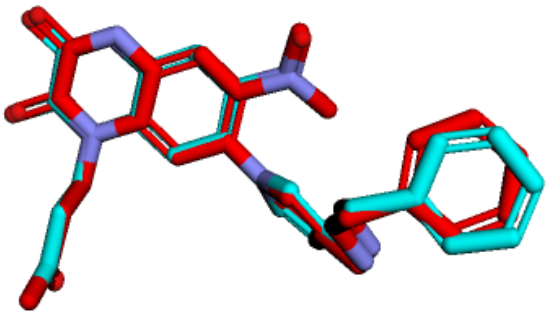
Here are the two conformers aligned using the atom indices:

```
>>> d1 = rdMolAlign.AlignMol(m3d,m3d,prbCid=cids[maxi],refCid=cids[maxj])
>>> drawit(m3d,[cids[maxj],cids[maxi]])
```



And then aligned with GetBestRMS:

```
>>> d2 = rdMolAlign.GetBestRMS(m3d,m3d,prbId=cids[maxi],refId=cids[maxj])
>>> drawit(m3d,[cids[maxj],cids[maxi]])
```



The alignments don't actually look all that different. The difference in RMSD is due to the atoms in the nitro group and the terminal phenyl ring: GetBestRMS() recognizes that the symmetry while AlignMol() just matches atoms by their indices.

Part 3. Cluster them

Now generate the RMSD distance matrix using GetBestRMS(). Note that as of the 2022.09 release cycle the convenience function AllChem.GetConformerRMSMatrix() does not take symmetry into account, so we need to build the symmetric matrix (we just store the lower triangle) manually:

```
>>> dists = []
>>> for i in range(len(cids)):
```

```
...     for j in range(i):
...         dists.append(rdMolAlign.GetBestRMS(m3d,m3d,i,j))
```

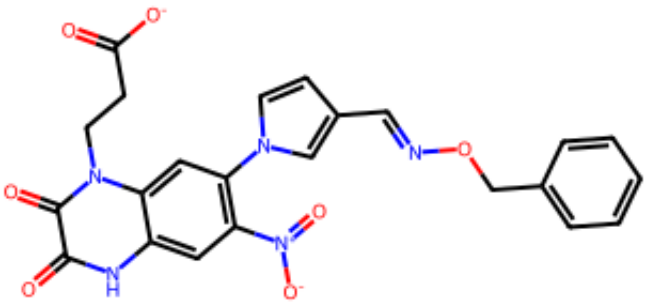
Now we can do Butina clustering. We use a distance threshold of 1.5 Å:

```
>>> from rdkit.ML.Cluster import Butina
>>> clusts = Butina.ClusterData(dists, len(cids), 1.5,
                                isDistData=True, reordering=True)
>>> len(clusts)
10
```

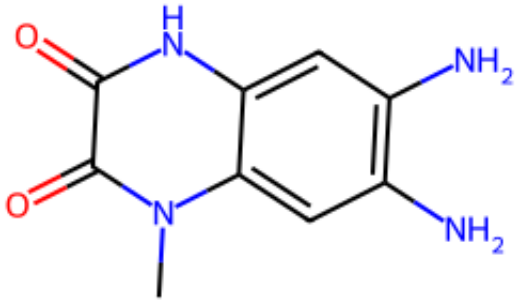
That's it. The 300 conformers form 10 clusters. Let's visualize the centroids (the first conformer in each cluster).

To make visualization of the clusters easier to interpret, align all the conformers to the rigid 6-6 core:

```
>>> m
```



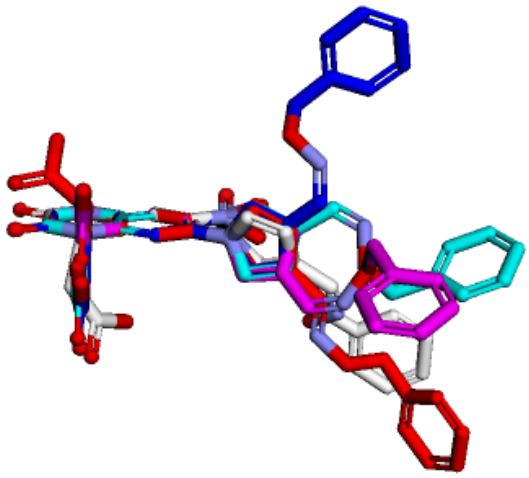
```
>>> core = Chem.MolFromSmiles('O=C1C(=O)N(C)c2cc(N)c(N)cc2N1')
>>> core
```



```
>>> rdMolAlign.AlignMolConformers(m3d,atomIds = m3d.GetSubstructMatch(core))
```

Now look at the first five cluster centroids;

```
>>> centroids = [x[0] for x in clusts]
>>> drawit(m3d,centroids[:5])
```



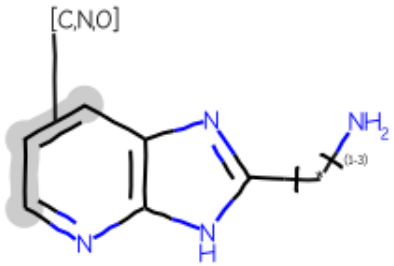
It's also possible to cluster molecules using torsion fingerprint differences (TFDs), but that's a topic for another post.

Generalized Substructure Search

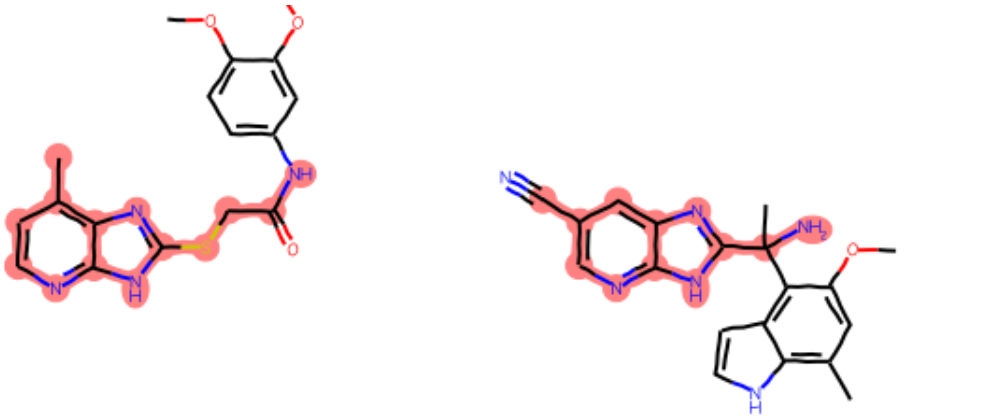
Over the last couple of releases a number of RDKit features have been implemented to allow the usage of more advanced substructure query features and more control over the results returned by substructure searches. These include:

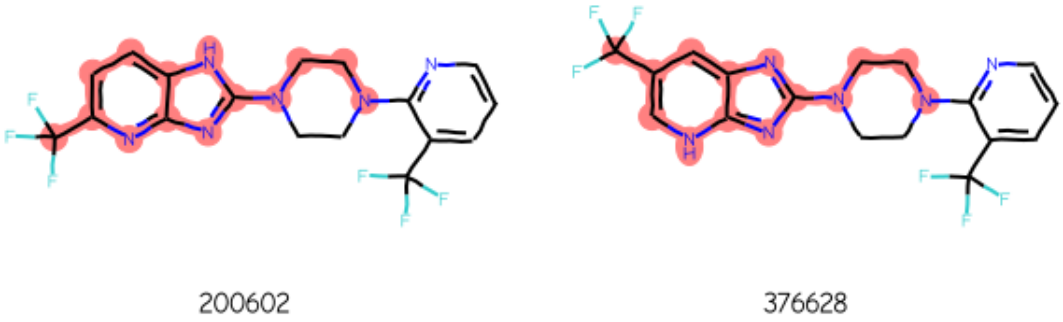
- **Chem.AdjustQueryProperties()**: to tune the results returned by a substructure query
- **rdMolEnumerator.Enumerate()**: to enumerate some V3000 mol block query features (as of the 2021.03 release the supported features are variable attachment points and link nodes)
- **rdTautomerQuery.TautomerQuery()**: to allow tautomer-insensitive substructures search

This tutorial focuses on how to use these functionalities together to do "generalized substructures searching" with the RDKit. Towards the bottom there are also a couple of Python functions which can be used in other scripts to make this process easier.
As an example, here's a query:



and here are four ChEMBL molecules returned using that query:






```
>>> from rdkit import Chem
>>> from rdkit.Chem import rdMolEnumerator
>>> from rdkit.Chem import rdTautomerQuery
>>> from rdkit.Chem import Draw
>>> from rdkit.Chem.Draw import IPythonConsole
>>> IPythonConsole.drawOptions.minFontSize = 10
>>> Draw.SetComicMode(IPythonConsole.drawOptions)
>>> from rdkit.Chem import rdDepictor
>>> rdDepictor.SetPreferCoordGen(True)
>>> import rdkit
>>> print(rdkit.__version__)
>>> import time
```

```
>>> print(time.asctime())
2021.09.3
Sun Dec 19 06:14:00 2021
```

Part 1. Substructure library creation

A SubstructLibrary can be created using ChEMBL 29. The chembl_downloader Python package is used to abstract away downloading and parsing the ChEMBL SDF data. More information can be found at this link. The code used to construct this is:

```
>>> from rdkit import RDLogger
>>> from rdkit import Chem
>>> from rdkit.Chem import rdSubstructLibrary
>>> import pickle, time
>>> import gzip
>>> import chembl_downloader

>>> RDLogger.DisableLog("rdApp.warning")
>>> with chembl_downloader.supplier(version="29") as suppl:
...     t1=time.time()
...     data = []
...     for i,mol in enumerate(suppl):
...         if not ((i+1)%50000):
...             print(f"Processed {i+1} molecules in
                        {(time.time()-t1):.1f} seconds")
...         if mol is None or mol.GetNumAtoms()>50:
...             continue
...         fp = Chem.PatternFingerprint(mol,fpSize=1024,
                                         tautomerFingerprints=True)
...         smi = Chem.MolToSmiles(mol)
```

```
...         data.append((smi,fp))
>>> t2=time.time()
>>> pickle.dump(data,open('../data/chembl29_sssdata.pkl','wb+'))
>>> t1=time.time()
>>> mols = rdSubstructLibrary.CachedTrustedSmilesMolHolder()
>>> fps = rdSubstructLibrary.TautomerPatternHolder(1024)
>>> for smi,fp in data:
...     mols.AddSmiles(smi)
...     fps.AddFingerprint(fp)
>>> library = rdSubstructLibrary.SubstructLibrary(mols,fps)
>>> t2=time.time()
>>> print(f"That took {t2-t1:.2f} seconds. \
            The library has {len(library)} molecules.")
>>> pickle.dump(library,open('../data/chembl29_ssslib.pkl','wb+'))
```

The library is then saved as binary for further uses. To load it the following code can be used:

```
>>> import pickle
>>> with open('./results/chembl29_ssslib.pkl','rb') as inf:
...     sslib = pickle.load(inf)
>>> print(f'SubstructLibrary loaded with {len(sslib)} molecules')
SubstructLibrary loaded with 2049078 molecules
```

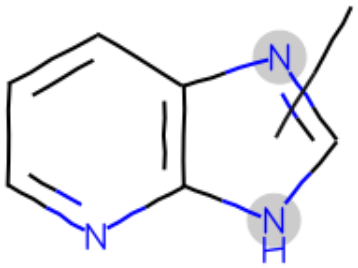
Part 2. Enumeration

Start with a query including a variable attachment point:

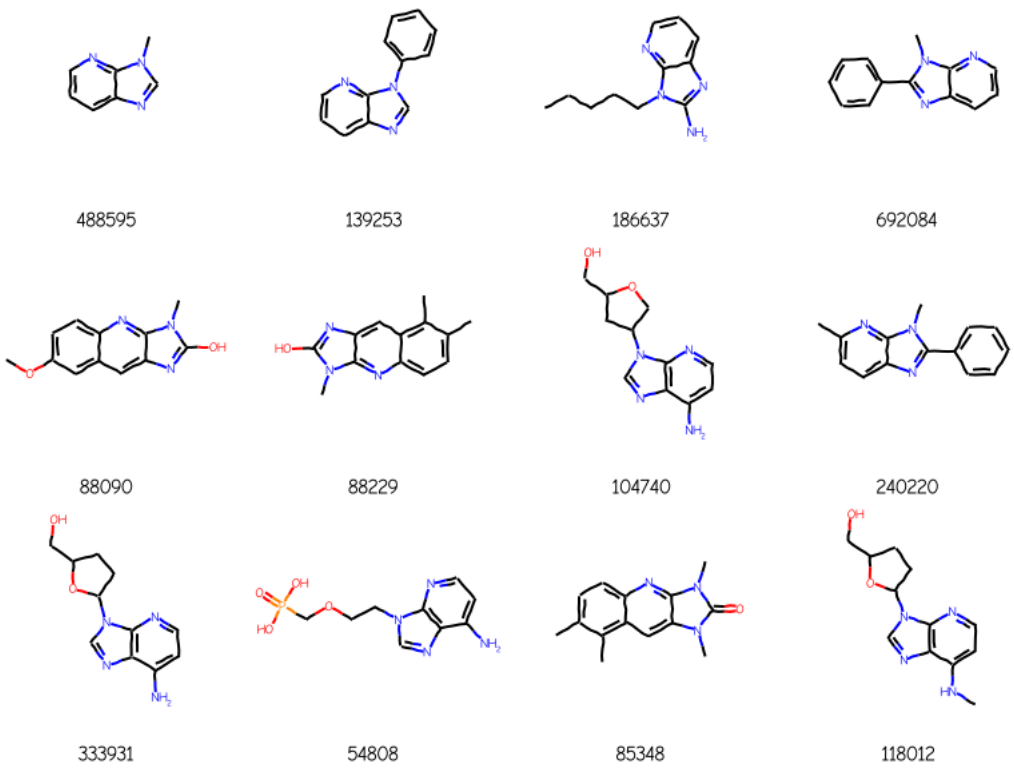
```
>>> qry = Chem.MolFromMolBlock('''
     Mrv2108 08012107372D
```

```
  0  0  0     0  0            999 V3000
M  V30 BEGIN CTAB
M  V30 COUNTS 11 11 0 0 0
M  V30 BEGIN ATOM
M  V30 1 C -2.4167 7.8734 0 0
M  V30 2 C -3.7503 7.1034 0 0
M  V30 3 C -3.7503 5.5633 0 0
M  V30 4 N -2.4167 4.7933 0 0
M  V30 5 C -1.083 5.5633 0 0
M  V30 6 C -1.083 7.1034 0 0
M  V30 7 N 0.3973 7.5279 0 0
M  V30 8 N 0.3104 5.0377 0 0
M  V30 9 C 1.2585 6.2511 0 0
M  V30 10 * 0.3539 6.2828 0 0
M  V30 11 C 1.5089 8.2833 0 0
M  V30 END ATOM
M  V30 BEGIN BOND
M  V30 1 1 1 2
M  V30 2 2 2 3
M  V30 3 1 3 4
M  V30 4 2 4 5
M  V30 5 1 5 6
M  V30 6 2 1 6
M  V30 7 1 7 6
M  V30 8 1 5 8
M  V30 9 1 8 9
M  V30 10 2 7 9
M  V30 11 1 10 11 ENDPTS=(2 8 7) ATTACH=ANY
```

```
    M  V30 END BOND
    M  V30 END CTAB
    M  END
    ''')
>>> qry
```



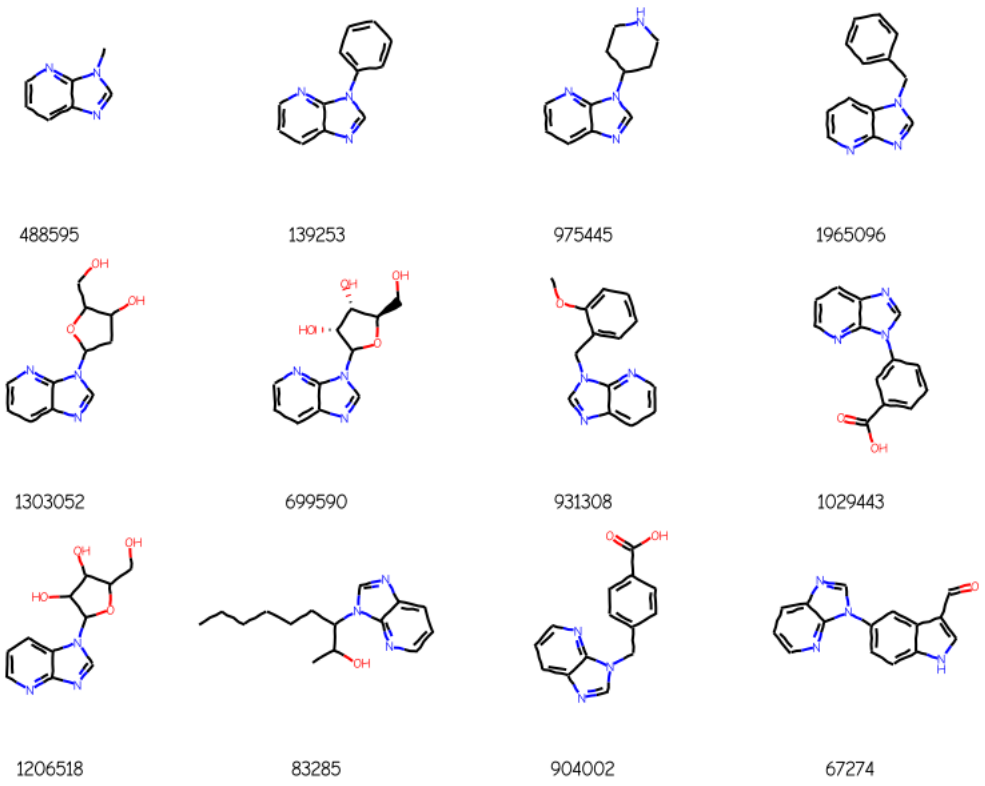

```
>>> bndl = rdMolEnumerator.Enumerate(qry)
>>> matches = sslib.GetMatches(bndl)
>>> print(f'{len(matches)} matches')
>>> mols = [sslib.GetMol(x) for x in matches]
    # sort the molecules by number of atoms and preserve the match ID
>>> sorted_res = sorted(zip(mols,matches),key=lambda x:x[0].GetNumAtoms())
>>> sorted_mols,sorted_matches = zip(*sorted_res)
>>> Draw.MolsToGridImage(sorted_mols[:12],
                         legends=[str(x) for x in sorted_matches],
                         molsPerRow=4)
1000 matches
```



Those include some addditional rings attached to the core in molecules like 1476, 10083, and 10853. We can prevent that by calling AdjustQueryProperties():

```
>>> bndl = Chem.MolBundle()
>>> for m in rdMolEnumerator.Enumerate(qry):
...     bndl.AddMol(Chem.AdjustQueryProperties(m))
>>> matches = sslib.GetMatches(bndl)
>>> print(f'{len(matches)} matches')
>>> mols = [sslib.GetMol(x) for x in matches]
    # sort the molecules by number of atoms and preserve the match ID
>>> sorted_res = sorted(zip(mols,matches),key=lambda x:x[0].GetNumAtoms())
>>> sorted_mols,sorted_matches = zip(*sorted_res)
>>> Draw.MolsToGridImage(sorted_mols[:12],
                         legends=[str(x) for x in sorted_matches],
                         molsPerRow=4)
```

```
148 matches
```



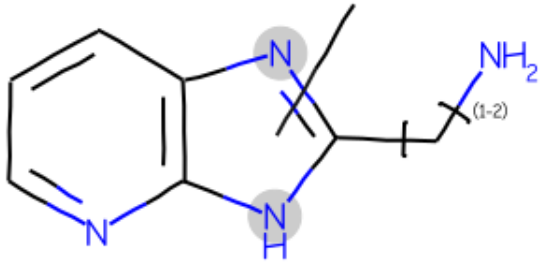
An aside: this would be more convenient if AdjustQueryProperties directly supported passing MolBundle objects. That's something for a future version.

Now let's make the query more complex by adding a link node in addition to the variable attachment point:

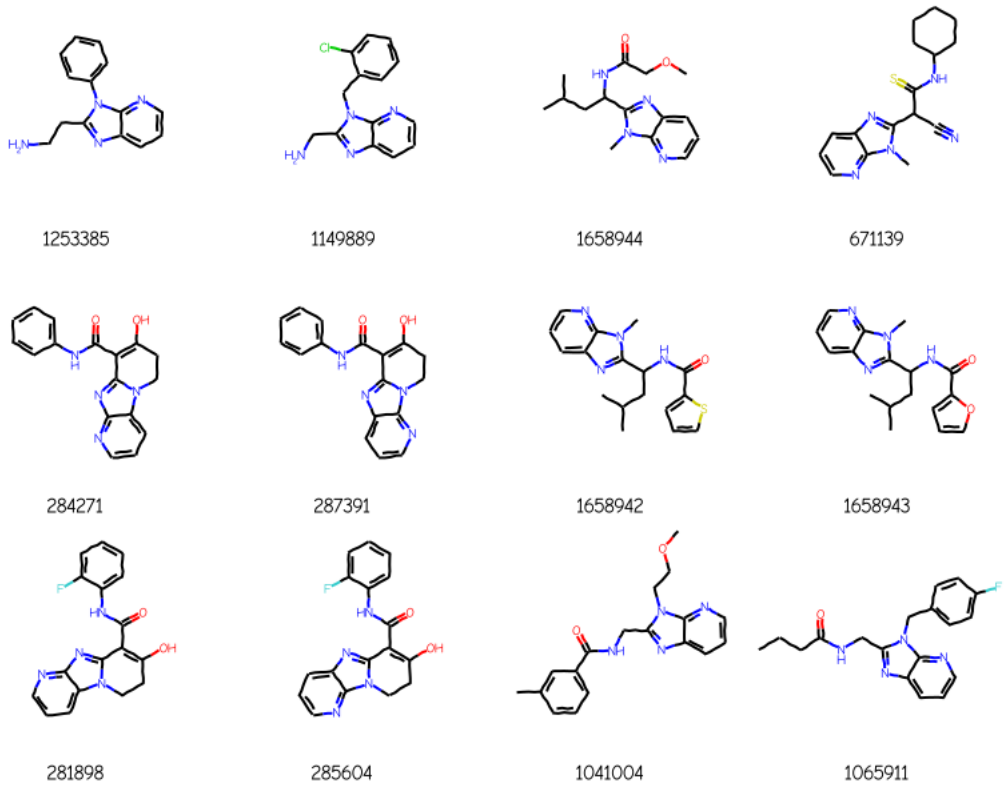
```
>>> qry = Chem.MolFromMolBlock('''
     Mrv2108 08012108062D

     0  0  0     0  0            999 V3000
M  V30 BEGIN CTAB
M  V30 COUNTS 13 13 0 0 0
M  V30 BEGIN ATOM
M  V30 1 C -2.4167 7.8734 0 0
M  V30 2 C -3.7503 7.1034 0 0
```

```
M  V30 3 C -3.7503 5.5633 0 0
M  V30 4 N -2.4167 4.7933 0 0
M  V30 5 C -1.083 5.5633 0 0
M  V30 6 C -1.083 7.1034 0 0
M  V30 7 N 0.3973 7.5279 0 0
M  V30 8 N 0.3104 5.0377 0 0
M  V30 9 C 1.2585 6.2511 0 0
M  V30 10 * 0.3539 6.2828 0 0
M  V30 11 C 1.5089 8.2833 0 0
M  V30 12 C 2.7975 6.1974 0 0
M  V30 13 N 3.6136 7.5033 0 0
M  V30 END ATOM
M  V30 BEGIN BOND
M  V30 1 1 1 2
M  V30 2 2 2 3
M  V30 3 1 3 4
M  V30 4 2 4 5
M  V30 5 1 5 6
M  V30 6 2 1 6
M  V30 7 1 7 6
M  V30 8 1 5 8
M  V30 9 1 8 9
M  V30 10 2 7 9
M  V30 11 1 10 11 ENDPTS=(2 8 7) ATTACH=ANY
M  V30 12 1 9 12
M  V30 13 1 12 13
M  V30 END BOND
M  V30 LINKNODE 1 2 2 12 9 12 13
```

```
M  V30 END CTAB
M  END
''')
>>> qry
```




```
>>> bndl = rdMolEnumerator.Enumerate(qry)
>>> matches = sslib.GetMatches(bndl)
>>> print(f'{len(matches)} matches')
>>> mols = [sslib.GetMol(x) for x in matches]
    # sort the molecules by number of atoms and preserve the match ID
>>> sorted_res = sorted(zip(mols,matches),key=lambda x:x[0].GetNumAtoms())
>>> sorted_mols,sorted_matches = zip(*sorted_res)
>>> Draw.MolsToGridImage(sorted_mols[:12],
                         legends=[str(x) for x in sorted_matches],
                         molsPerRow=4)
193 matches
```



Part 3. Enumeration and tautomer-insensitive queries

Here we will use the RDKit's TautomerQuery class to do tautomer-insensitive substructure queries. We start by enumerating the molecules, as above, but then convert each of the results into a TautomerQuery.

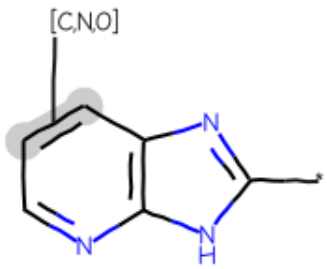
To see what's going on here it helps to have the result molecules all aligned the same way. In order to do that we also need to generate query molecules with aligned coordinates.

```
>>> from rdkit.Chem import rdFMCS
def getAlignedQueries(qry):
    # generate a conformer for the query if we don't have one already
    if not qry.GetNumConformers():
        rdDepictor.Compute2DCoords(qry)


    bndl = rdMolEnumerator.Enumerate(qry)
```

```
    # find the MCS of the enumerated molecules:
    mcs = rdFMCS.FindMCS(bndl)
    qmcs = Chem.MolFromSmarts(mcs.smartsString)

    # now adjust the properties, generate coordinates,
    # and create the TautomerQuery
    queries = []
    for q in bndl:
        q = Chem.AdjustQueryProperties(q)
        rdDepictor.GenerateDepictionMatching2DStructure(q,qry,refPatt=qmcs)
        queries.append(rdTautomerQuery.TautomerQuery(q))
    return queries

def drawAlignedMols(mols,qry,legends=None,molsPerRow=4):
    queries = getAlignedQueries(qry)
    for i,m in enumerate(mols):
        for q in queries:
            if q.IsSubstructOf(m):
                rdDepictor.GenerateDepictionMatching2DStructure(m,
                        q.GetTemplateMolecule())
    return Draw.MolsToGridImage(mols,legends=legends,molsPerRow=molsPerRow)

>>> qry = Chem.MolFromMolBlock('''
     Mrv2108 08012108222D

     0  0  0     0  0            999 V3000
    M  V30 BEGIN CTAB
    M  V30 COUNTS 12 12 0 0 0
```

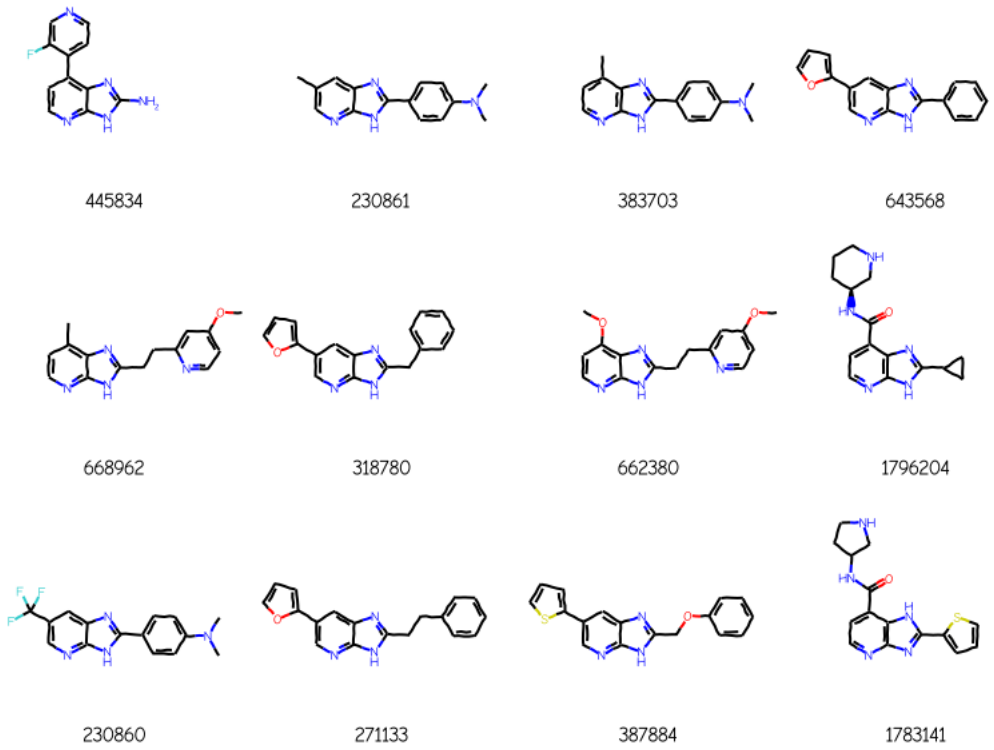
```
M  V30 BEGIN ATOM
M  V30 1 C -2.4167 7.8734 0 0
M  V30 2 C -3.7503 7.1034 0 0
M  V30 3 C -3.7503 5.5633 0 0
M  V30 4 N -2.4167 4.7933 0 0
M  V30 5 C -1.083 5.5633 0 0
M  V30 6 C -1.083 7.1034 0 0
M  V30 7 N 0.3973 7.5279 0 0
M  V30 8 N 0.3104 5.0377 0 0
M  V30 9 C 1.2585 6.2511 0 0
M  V30 10 * -3.0835 7.4884 0 0
M  V30 11 [C,N,O] -3.0835 9.7984 0 0
M  V30 12 * 2.7975 6.1974 0 0
M  V30 END ATOM
M  V30 BEGIN BOND
M  V30 1 1 1 2
M  V30 2 2 2 3
M  V30 3 1 3 4
M  V30 4 2 4 5
M  V30 5 1 5 6
M  V30 6 2 1 6
M  V30 7 1 7 6
M  V30 8 1 5 8
M  V30 9 1 8 9
M  V30 10 2 7 9
M  V30 11 1 10 11 ENDPTS=(2 2 1) ATTACH=ANY
M  V30 12 1 9 12
M  V30 END BOND
```

```
M  V30 END CTAB
M  END
''')
>>> qry
```



Start by doing a tautomer-sensitive query to see how many results we get:

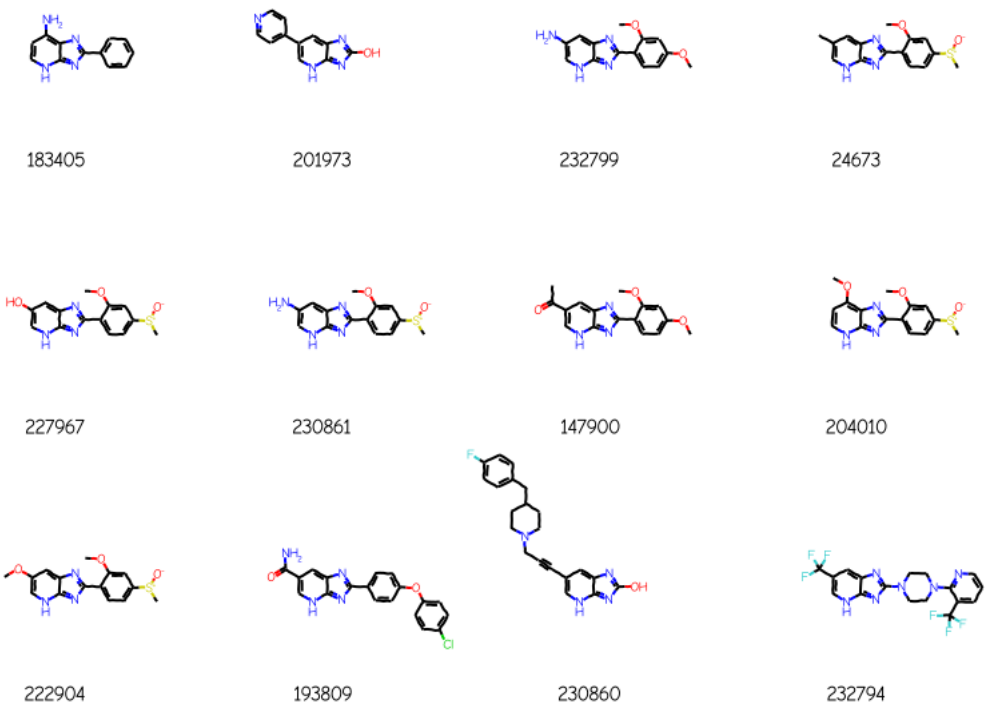
```
>>> bndl = rdMolEnumerator.Enumerate(qry)
>>> matches = []
>>> for m in bndl:
...     m = Chem.AdjustQueryProperties(m)
...     matches.extend(sslib.GetMatches(m))
>>> print(f'{len(matches)} matches')
>>> mols = [sslib.GetMol(x) for x in matches]
    # sort the molecules by number of atoms and preserve the match ID
>>> sorted_res = sorted(zip(mols,matches),key=lambda x:x[0].GetNumAtoms())
>>> sorted_mols,sorted_matches = zip(*sorted_res)
>>> drawAlignedMols(sorted_mols[:12],qry,[str(x) for x in sorted_matches])
276 matches
```



Now do the tautomer-insensitive version of that and show just the new molecules

```
>>> bndl = rdMolEnumerator.Enumerate(qry)
>>> matches2 = []
>>> for m in bndl:
...     m = Chem.AdjustQueryProperties(m)
...     tqry = rdTautomerQuery.TautomerQuery(m)
...     matches2.extend(sslib.GetMatches(tqry))
>>> extras = set(matches2).difference(matches)
>>> print(f'{len(matches2)} matches, {len(extras)} are non-overlapping')
>>> mols = [sslib.GetMol(x) for x in extras]
    # sort the molecules by number of atoms and preserve the match ID
>>> sorted_res = sorted(zip(mols,matches),key=lambda x:x[0].GetNumAtoms())
>>> sorted_mols,sorted_matches = zip(*sorted_res)
>>> drawAlignedMols(sorted_mols[:12],qry,[str(x) for x in sorted_matches])
```

```
288 matches, 12 are non-overlapping
```



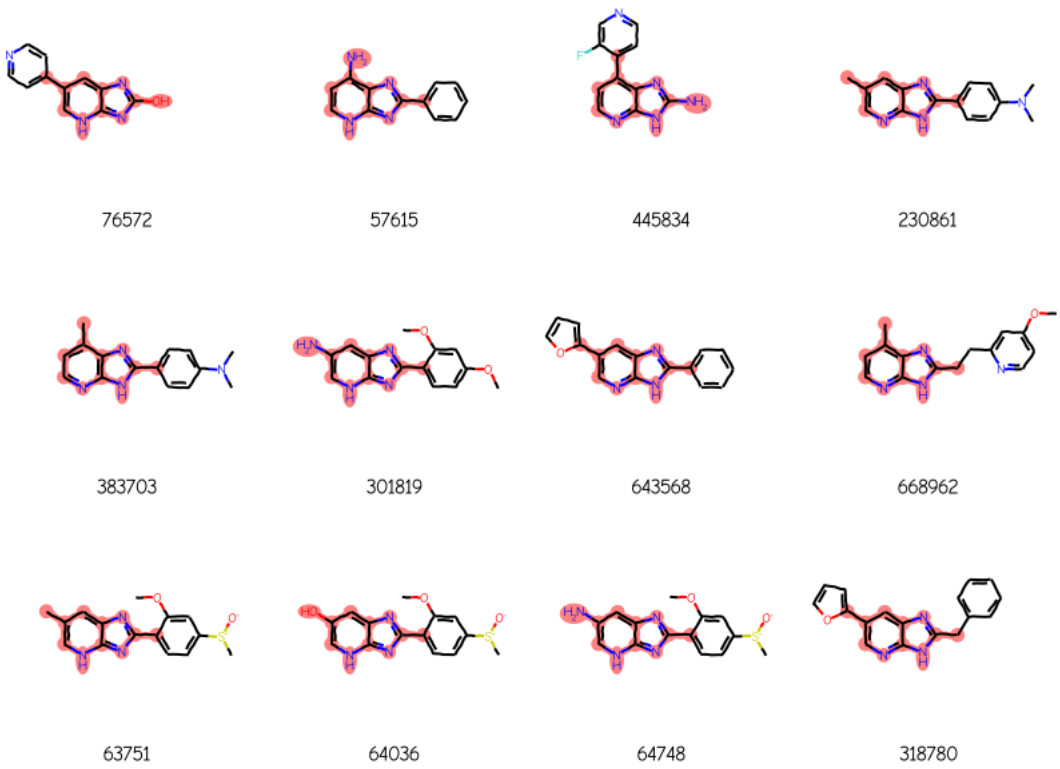
Part 4. Bring it all together

Now let's put that all together in one function and include the information required to do atom highlighting in the structure drawings.

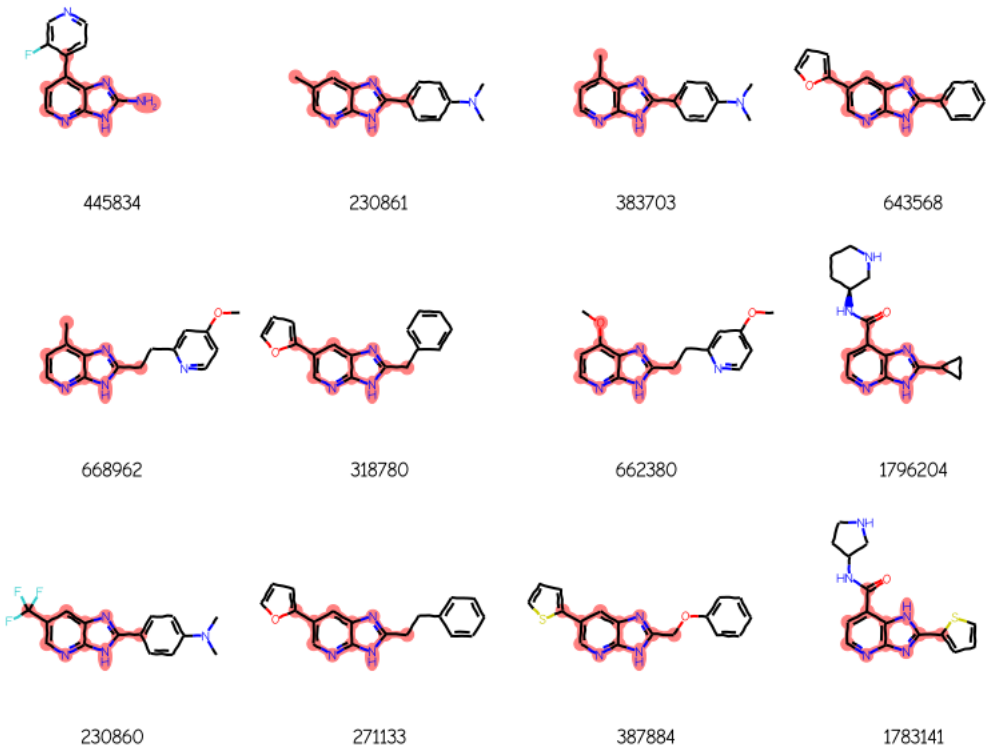
```
>>> from rdkit.Chem import rdFMCS
>>> from rdkit.Chem import rdTautomerQuery

# this function does the enumeration of the queries
def getAlignedQueries(qry,tautomerInsensitive=True):
    if not qry.GetNumConformers():
        rdDepictor.Compute2DCoords(qry)
    bndl = rdMolEnumerator.Enumerate(qry)
```

```
    # find the MCS of the enumerated molecules:
    mcs = rdFMCS.FindMCS(bndl)
    qmcs = Chem.MolFromSmarts(mcs.smartsString)

    # now adjust the properties, generate coordinates,
    # and create the TautomerQuery
    queries = []
    for q in bndl:
        q = Chem.AdjustQueryProperties(q)
        rdDepictor.GenerateDepictionMatching2DStructure(q,qry,refPatt=qmcs)
        if tautomerInsensitive:
            q = rdTautomerQuery.TautomerQuery(q)
        queries.append(q)
    return queries

def generalizedSubstructureSearch(query,sslib,tautomerInsensitive=True,
                                  alignResults=True, maxResults=1000):
    queries = getAlignedQueries(query,tautomerInsensitive=tautomerInsensitive)
    matches = []
    for q in queries:
        matches.extend(sslib.GetMatches(q,maxResults=maxResults))
    tmols = [(x,sslib.GetMol(x)) for x in matches]
    mols = []
    for idx,mol in sorted(tmols,key=lambda x:x[1].GetNumAtoms()):
        match = None
        if(alignResults):
            for q in queries:
                if tautomerInsensitive:
```

```
                    match = q.GetSubstructMatch(mol)
                    if match:
                        rdDepictor.GenerateDepictionMatching2DStructure(mol,
                                q.GetTemplateMolecule())
                        break
                else:
                    match = mol.GetSubstructMatch(q)
                    if match:
                        rdDepictor.GenerateDepictionMatching2DStructure(mol,q)
                        break

        mols.append((idx,mol,match))
        if len(mols)>=maxResults:
            break
    return mols

>>> res = generalizedSubstructureSearch(qry,sslib)
>>> ids,mols,matchAtoms = zip(*res)
>>> print(f'{len(mols)} results')
>>> Draw.MolsToGridImage(mols[:12],legends=[str(x) for x in ids],
                         highlightAtomLists=matchAtoms, molsPerRow=4)
288 results
```



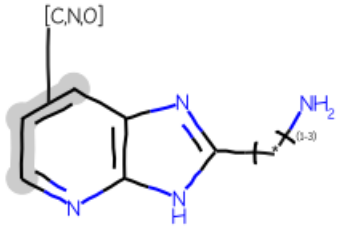

```
>>> res = generalizedSubstructureSearch(qry,sslib,tautomerInsensitive=False)
>>> ids,mols,matchAtoms = zip(*res)
>>> print(f'{len(mols)} results')
>>> Draw.MolsToGridImage(mols[:12],legends=[str(x) for x in ids],
                         highlightAtomLists=matchAtoms,
                         molsPerRow=4)
276 results
```



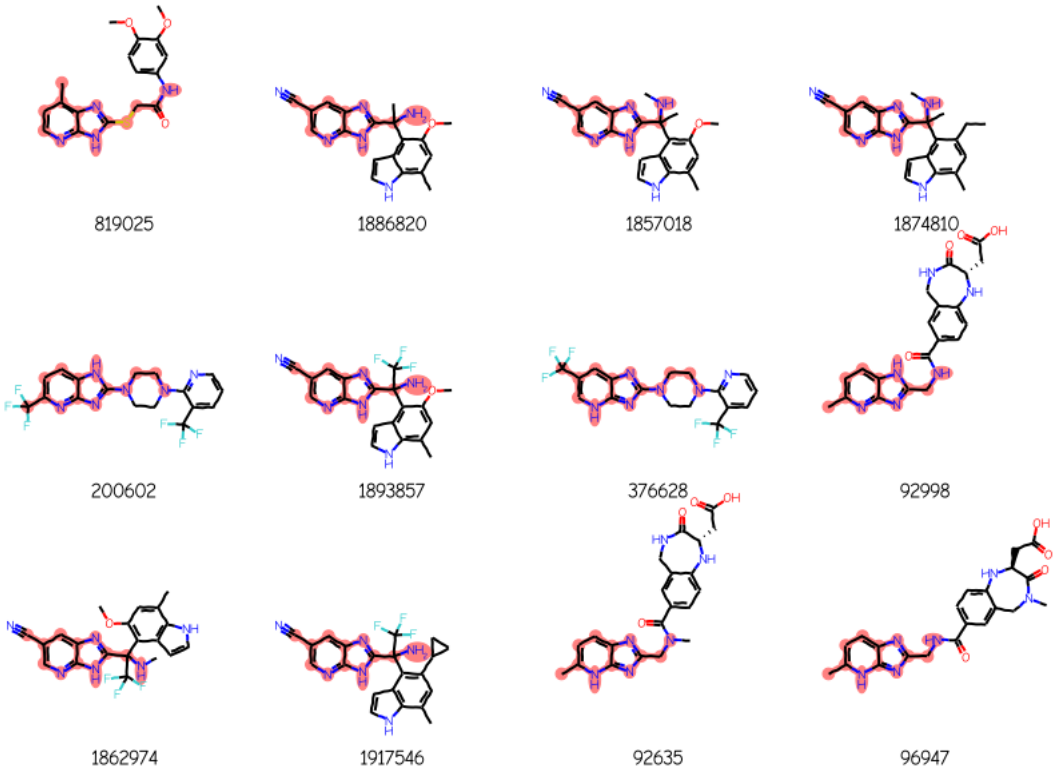
Last example, link nodes + variable attachment + tautomer enumeration

```
>>> qry = Chem.MolFromMolBlock('''
     Mrv2108 08032106392D

  0  0  0     0  0            999 V3000
M  V30 BEGIN CTAB
M  V30 COUNTS 13 13 0 0 0
M  V30 BEGIN ATOM
M  V30 1 C -2.4167 7.8734 0 0
M  V30 2 C -3.7503 7.1034 0 0
M  V30 3 C -3.7503 5.5633 0 0
M  V30 4 N -2.4167 4.7933 0 0
M  V30 5 C -1.083 5.5633 0 0
M  V30 6 C -1.083 7.1034 0 0
```

```
    M  V30 7 N 0.3973 7.5279 0 0
    M  V30 8 N 0.3104 5.0377 0 0
    M  V30 9 C 1.2585 6.2511 0 0
    M  V30 10 * -3.0835 7.4884 0 0
    M  V30 11 [C,N,O] -3.0835 9.7984 0 0
    M  V30 12 * 2.7975 6.1974 0 0
    M  V30 13 N 3.6136 7.5033 0 0
    M  V30 END ATOM
    M  V30 BEGIN BOND
    M  V30 1 1 1 2
    M  V30 2 2 2 3
    M  V30 3 1 3 4
    M  V30 4 2 4 5
    M  V30 5 1 5 6
    M  V30 6 2 1 6
    M  V30 7 1 7 6
    M  V30 8 1 5 8
    M  V30 9 1 8 9
    M  V30 10 2 7 9
    M  V30 11 1 10 11 ENDPTS=(3 3 2 1) ATTACH=ANY
    M  V30 12 1 9 12
    M  V30 13 1 12 13
    M  V30 END BOND
    M  V30 LINKNODE 1 3 2 12 9 12 13
    M  V30 END CTAB
    M  END
    ''')
>>> qry
```

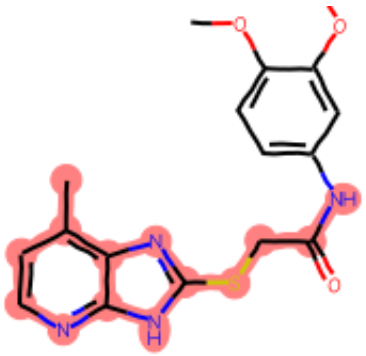



```
>>> res = generalizedSubstructureSearch(qry,sslib)
>>> ids,mols,matchAtoms = zip(*res)
>>> print(f'{len(mols)} results')
>>> Draw.MolsToGridImage(mols[:12],legends=[str(x) for x in ids],
                         highlightAtomLists=matchAtoms,
                         molsPerRow=4)
24 results
```

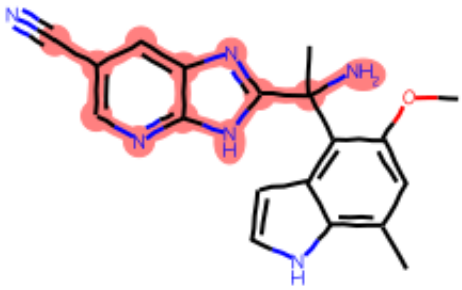


Summarizing:

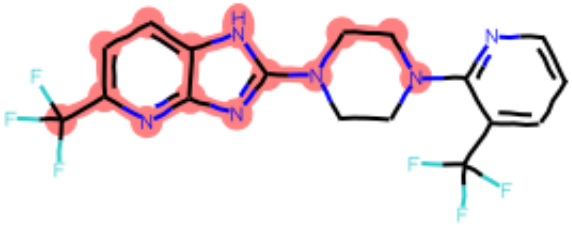
```
>>> keep = [0,1,4,6]
>>> ids,mols,matchAtoms = zip(*[res[x] for x in keep])
>>> Draw.MolsToGridImage(mols[:12],legends=[str(x) for x in ids],
                         highlightAtomLists=matchAtoms,
                         molsPerRow=2,subImgSize=(300,250))
```



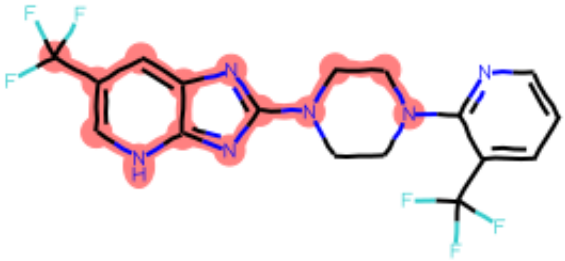
819025



1886820



200602



376628

Advanced features in SubstructLibrary

In the previous tutorial "Generalized substructure search" we saw how to use some query features like link nodes, variable attachment points, and tautomer insensitivity to search through the compounds from ChEMBL 29 with the RDKit's SubstructLibrary. Using the same ChEMBL 29 SubstructLibrary, we'll look into some more advanced new features which were added in the 2021.09 release of the RDKit:

1. Changing the search order
2. Specifying which compounds are actually searched
3. Saving a molecule key (or name) together with the molecules in the SubstructLibrary

```
>>> from rdkit import Chem
>>> from rdkit.Chem.Draw import IPythonConsole
>>> from rdkit.Chem import Draw
>>> from rdkit.Chem import rdSubstructLibrary
>>> from rdkit import RDLogger
>>> from rdkit import rdBase
>>> import pickle
>>> import time
>>> import gzip
>>> print(rdBase.rdkitVersion)
>>> print(time.asctime())
2021.09.3
Mon Dec 20 05:14:01 2021
```

Here's the code to build the SubstructLibrary from the sdf file distributed by the ChEMBL team. This uses a feature added in RDKit v2021.09 to allow a molecule key (or name) to be stored with the molecules in a SubstructLibrary.
Executing this takes about 45 minutes on my machine.

```
>>> RDLogger.DisableLog('rdApp.warning')
>>> molholder = rdSubstructLibrary.CachedTrustedSmilesMolHolder()
>>> patts = rdSubstructLibrary.TautomerPatternHolder()
# this will automatically grab the '_Name' property for each molecule
# in the ChEMBL SD file this contains the ChEMBL ID for the molecules.
>>> keys = rdSubstructLibrary.KeyFromPropHolder()
>>> slib = rdSubstructLibrary.SubstructLibrary(molholder,patts,keys)
>>> t1 = time.time()
>>> with gzip.GzipFile('/home/glandrum/Downloads/chembl_29.sdf.gz') as gz and
            Chem.ForwardSDMolSupplier(gz) as suppl:
...     nDone = 0
...     for m in suppl:
...         if m is None: continue
        # skip huge molecules
            if m.GetNumHeavyAtoms()>75: continue
            slib.AddMol(m) nDone += 1
            if not nDone%50000:
...             print(f' did {nDone} in {time.time()-t1:.2f}s')
>>>     with open('./results/chembl29_ssslib.pkl','wb+') as outf:
...         pickle.dump(slib,outf)
>>>     print(f'That took {time.time()-t1:.2f}s in total.')
```

We're going to use the number of heavy atoms to determine the search order. Since that takes a while, go ahead and pre-calculate those values and store them in the same pickle file as the SubstructLibrary:

```
>>> holder = slib.GetMolHolder()
>>> nats = sorted([(holder.GetMol(x).GetNumHeavyAtoms(),x)
                   for x in range(len(slib))])
>>> order = [y for x,y in nats]
```

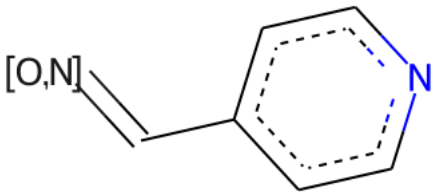
```
# append that to the pickle file with the substruct lib
>>> with open('/results/chembl29_ssslib.pkl','ab') as outf:
...     pickle.dump(order,outf)
```

Read in the saved data:

```
>>> with open('./results/chembl29_ssslib.pkl','rb') as inf:
...     slib = pickle.load(inf)
...     nat_order = pickle.load(inf)
```

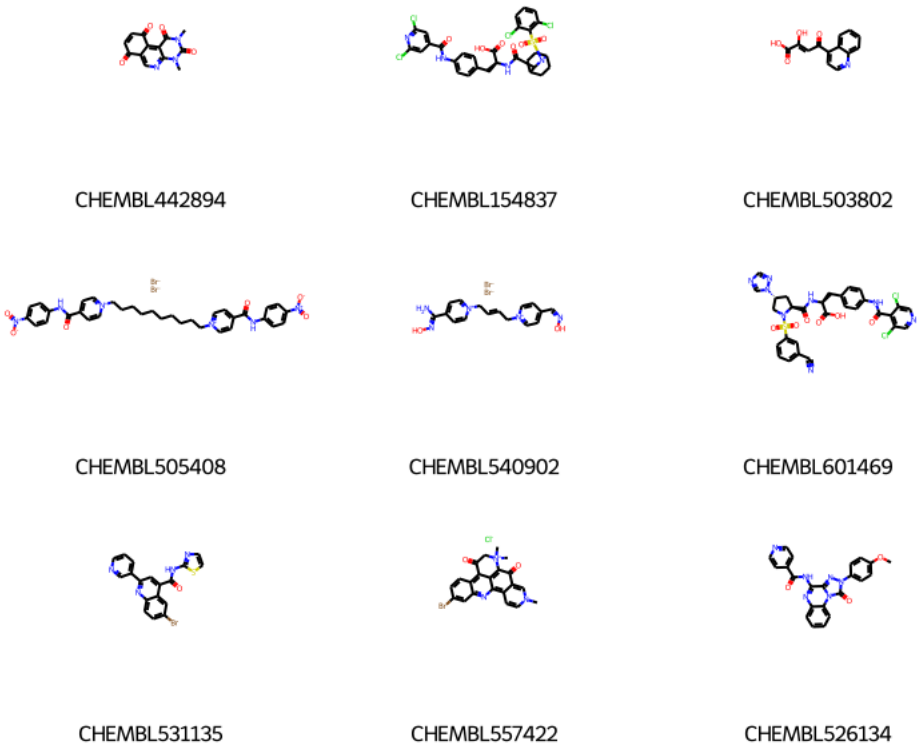
Here's the query we'll use for this post:

```
>>> qry = Chem.MolFromSmarts('[O,N]=C-c:1:c:c:n:c:c:1')
>>> qry
```



Let's look at doing a search. We also take advantage of the SubstructLibrary's KeyHolder (a new feature in v2021.09) to include the compound ChEMBL IDs in the results:

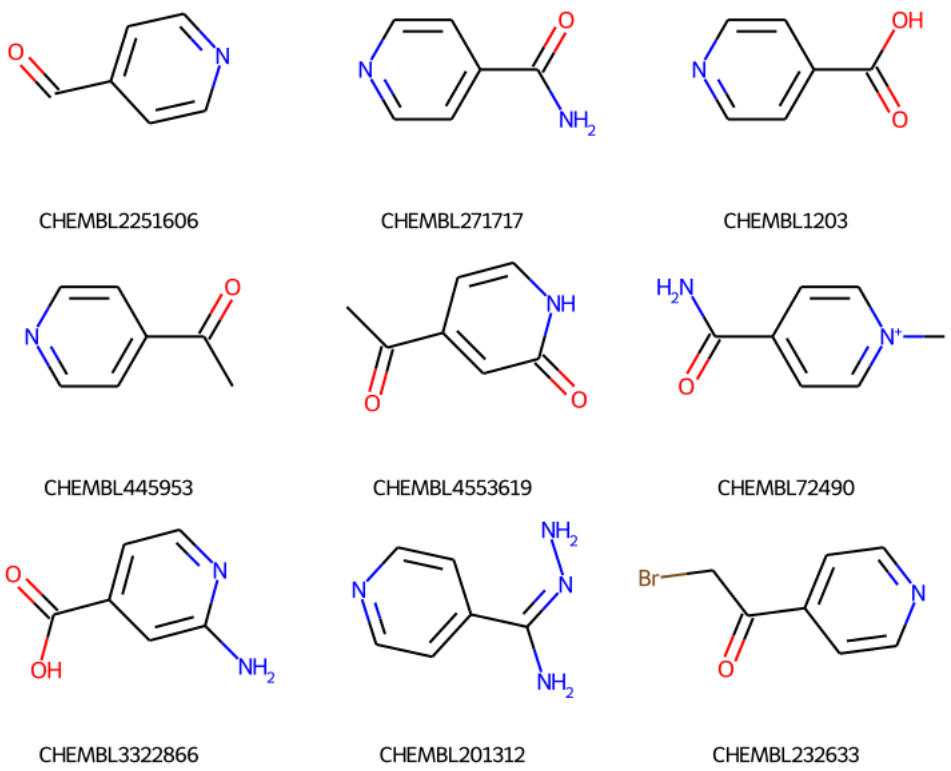
```
>>> mids = slib.GetMatches(qry)
>>> print(f'{len(mids)} results')
>>> ms = [slib.GetMolHolder().GetMol(x) for x in mids[:9]]
>>> legends = [slib.GetKeyHolder().GetKey(x) for x in mids[:9]]
>>> Draw.MolsToGridImage(ms,legends=legends,subImgSize=(250,200))
1000 results
```



One of the new features is that we can change the search order; this allows us to get the smallest molecules first (always a good idea with a substructure search).

Here we're using the number of heavy atoms to set the search order:

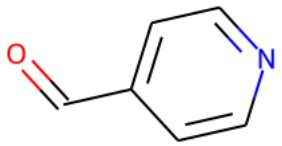
```
>>> slib.SetSearchOrder(nat_order)
>>> mids = slib.GetMatches(qry)
>>> print(f'{len(mids)} results')
>>> ms = [slib.GetMolHolder().GetMol(x) for x in mids[:9]]
>>> legends = [slib.GetKeyHolder().GetKey(x) for x in mids[:9]]
>>> Draw.MolsToGridImage(ms,legends=legends,subImgSize=(250,200))
1000 results
```



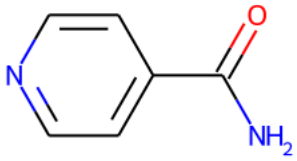
It's important to note that we are not just sorting the results from the search here: we're changing the order in which the search is done. So even though we're only getting 1000 results (the default max number of results from the SubstructLibrary), we know that they are the 1000 smallest results.

So if we change the maximum number of results to three, we'll get the same first three results:

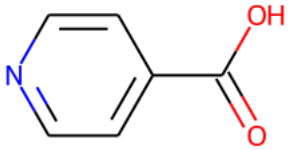
```
>>> slib.SetSearchOrder(nat_order)
>>> mids2 = slib.GetMatches(qry,maxResults=3)
>>> print(f'{len(mids2)} results')
>>> ms = [slib.GetMolHolder().GetMol(x) for x in mids2[:9]]
>>> legends = [slib.GetKeyHolder().GetKey(x) for x in mids2[:9]]
>>> Draw.MolsToGridImage(ms,legends=legends,subImgSize=(250,200))
3 results
```



CHEMBL2251606



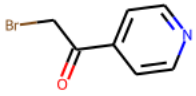
CHEMBL271717



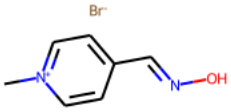
CHEMBL1203

We can also use the search order to limit the compounds we search. In this case I'm going to refine the results of the previous search and identify compounds which also contain Br:

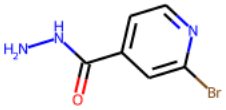
```
>>> slib.SetSearchOrder(mids)
>>> mids_new = slib.GetMatches(Chem.MolFromSmarts('[Br]'))
>>> print(f'{len(mids_new)} sub-results')
>>> ms = [slib.GetMolHolder().GetMol(x) for x in mids_new[:9]]
>>> legends = [slib.GetKeyHolder().GetKey(x) for x in mids_new[:9]]
>>> Draw.MolsToGridImage(ms,legends=legends,subImgSize=(250,200))
66 sub-results
```



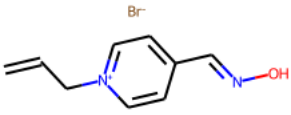
CHEMBL232633



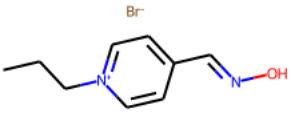
CHEMBL3335075



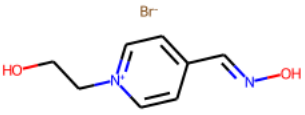
CHEMBL341358



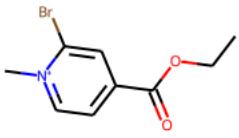
CHEMBL453349



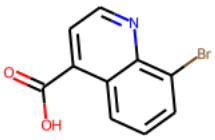
CHEMBL507002



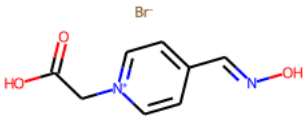
CHEMBL565771



CHEMBL1178821



CHEMBL3446210



CHEMBL4061312

Notice that the results are still coming back sorted by the number of heavy atoms. That's because the IDs of the molecules being used for the search search is sorted.

We almost certainly ran up against the default limit on the number of results (1000 compounds) when doing the first search. Let's loosen that to 50K. This will take longer since the first query ends up having to run through the entire database.

```
>>> slib.SetSearchOrder(nat_order)
>>> mids = slib.GetMatches(qry,maxResults=50000)
>>> print(f'{len(mids)} results')


>>> slib.SetSearchOrder(mids)
>>> mids_new = slib.GetMatches(Chem.MolFromSmarts('[Br]'))
>>> print(f'{len(mids_new)} sub-results')
17764 results
919 sub-results
```

It's never a bad idea to check the performance of these queries.

Here's the run time for the default value of maxResults:

```
>>> slib.SetSearchOrder(nat_order)
>>> %timeit mids = slib.GetMatches(qry)


71.7 ms + 710 µs per loop (mean ± std. dev. of 7 runs, 10 loops each)
```

And here's the search time for going through the entire database (we only get 17K results here, so maxResults=50K corresponds to searching through the entire database):

```
>>> slib.SetSearchOrder(nat_order)
>>> %timeit mids = slib.GetMatches(qry,maxResults=50000)


5.09 s ± 79.4 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
```

Searching with generic groups

One of the features added for the v2022.03 RDKit release is support for "Reaxys/Beilstein" generic groups, for example atoms with labels like "ARY" or "ACY" which can be used to make substructure searches more specific. This tutorial provides a quick overview of that functionality.

```
>>> from rdkit import Chem
>>> from rdkit.Chem import rdMolEnumerator
>>> from rdkit.Chem import rdTautomerQuery
>>> from rdkit.Chem import Draw
>>> from rdkit.Chem.Draw import IPythonConsole
>>> IPythonConsole.drawOptions.minFontSize = 10
>>> IPythonConsole.molSize = 350,300


>>> Draw.SetComicMode(IPythonConsole.drawOptions)
>>> from rdkit.Chem import rdDepictor
>>> rdDepictor.SetPreferCoordGen(True)
>>> import rdkit
>>> print(rdkit.__version__)
>>> import time
>>> print(time.asctime())
2022.03.1
Tue Apr  5 06:10:45 2022
```

Part 1. Create ChEMBL30 substructure library

Create a SubstructLibrary using ChEMBL_30 using the code below. The SubstructLibrary has been configured to return search results sorted by the number of heavy atoms so that it returns smaller, more specific results first.

```
>>> from rdkit import RDLogger
>>> from rdkit import Chem
>>> from rdkit.Chem import rdSubstructLibrary
>>> import pickle, time
>>> import gzip
>>> import numpy as np

>>> gz = gzip.GzipFile('/home/glandrum/Downloads/chembl_30.sdf.gz')
>>> suppl = Chem.ForwardSDMolSupplier(gz)
>>> RDLogger.DisableLog("rdApp.warning")
>>> t1=time.time()
>>> data = []
>>> for i,mol in enumerate(suppl):
...     if not ((i+1)%50000):
...         print(f"Processed {i+1} molecules \
                    in {(time.time()-t1):.1f} seconds")
...     if mol is None or mol.GetNumHeavyAtoms()>50:
...         continue
...     fp = Chem.PatternFingerprint(mol,fpSize=1024,tautomerFingerprints=True)
...     smi = Chem.MolToSmiles(mol)
...     data.append((smi,fp,mol.GetNumHeavyAtoms()))
>>> t2=time.time()
>>> pickle.dump(data,open('./results/chembl30_sssdata.pkl','wb+'))
>>> t1=time.time()
>>> mols = rdSubstructLibrary.CachedTrustedSmilesMolHolder()
>>> fps = rdSubstructLibrary.TautomerPatternHolder(1024)
>>> natoms = []
>>> for smi,fp,nats in data:
```

```
...     mols.AddSmiles(smi)
...     fps.AddFingerprint(fp)
...     natoms.append(nats)
>>> library = rdSubstructLibrary.SubstructLibrary(mols,fps)
>>> library.SetSearchOrder([int(x) for x in np.argsort(natoms)])


>>> t2=time.time()
>>> print(f"That took {t2-t1:.2f} seconds.  \
            The library has {len(library)} molecules.")
>>> pickle.dump(library,open('./results/chembl30_ssslib.pkl','wb+'))
```

Now check if everything went ok and the library can be loaded.

```
>>> import pickle
>>> with open('./results/chembl30_ssslib.pkl','rb') as inf:
...     sslib = pickle.load(inf)
>>> print(f'SubstructLibrary loaded with {len(sslib)} molecules')
SubstructLibrary loaded with 2035013 molecules
```

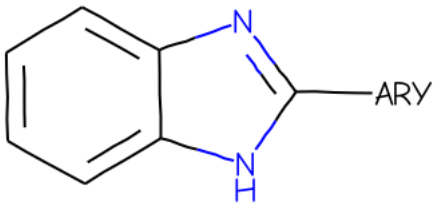
Part 2. Beilstein generic queries

Start with a simple query molecule which includes a generic atom in an SGroup:

```
>>> mb = '''
     Mrv2108 04052206092D
     0  0  0     0  0            999 V3000
   M  V30 BEGIN CTAB
   M  V30 COUNTS 10 11 1 0 0
   M  V30 BEGIN ATOM
   M  V30 1 C -11.4118 0.5479 0 0
   M  V30 2 C -11.3849 -0.9919 0 0
```

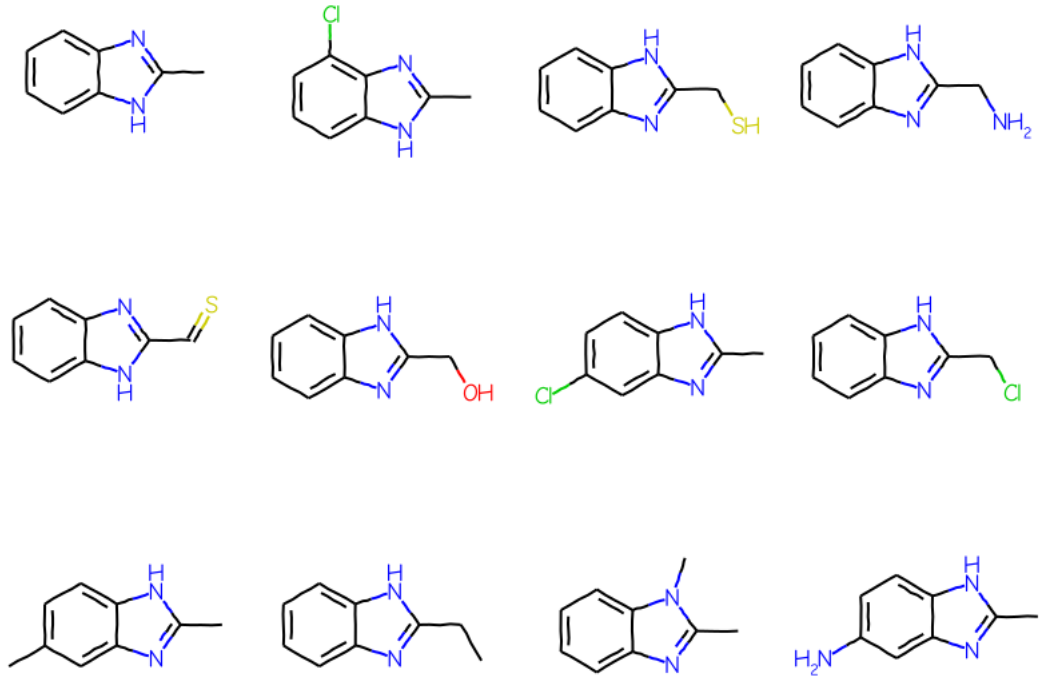
```
M  V30 3 C -10.0379 -1.7385 0 0
M  V30 4 C -8.7179 -0.9454 0 0
M  V30 5 C -8.7448 0.5943 0 0
M  V30 6 C -10.0918 1.341 0 0
M  V30 7 N -7.2453 -1.3956 0 0
M  V30 8 C -6.362 -0.1342 0 0
M  V30 9 N -7.2888 1.0958 0 0
M  V30 10 C -4.8222 -0.161 0 0
M  V30 END ATOM
M  V30 BEGIN BOND
M  V30 1 1 1 2
M  V30 2 2 2 3
M  V30 3 1 3 4
M  V30 4 2 4 5
M  V30 5 1 5 6
M  V30 6 2 1 6
M  V30 7 1 7 8
M  V30 8 1 5 9
M  V30 9 1 4 7
M  V30 10 2 8 9
M  V30 11 1 8 10
M  V30 END BOND
M  V30 BEGIN SGROUP
M  V30 1 SUP 0 -
M  V30 ATOMS=(1 10) -
M  V30 LABEL="ARY"
M  V30 END SGROUP
M  V30 END CTAB
```

```
    M  END
    '''
>>> bqry = Chem.MolFromMolBlock(mb)
    # show labels for the Sgroups:
>>> for sgs in Chem.GetMolSubstanceGroups(bqry):
...     if sgs.GetProp('TYPE') == 'SUP':
...         bqry.GetAtomWithIdx(sgs.GetAtoms()[0]).SetProp("atomLabel",
                                                           sgs.GetProp("LABEL"))
>>> bqry
```



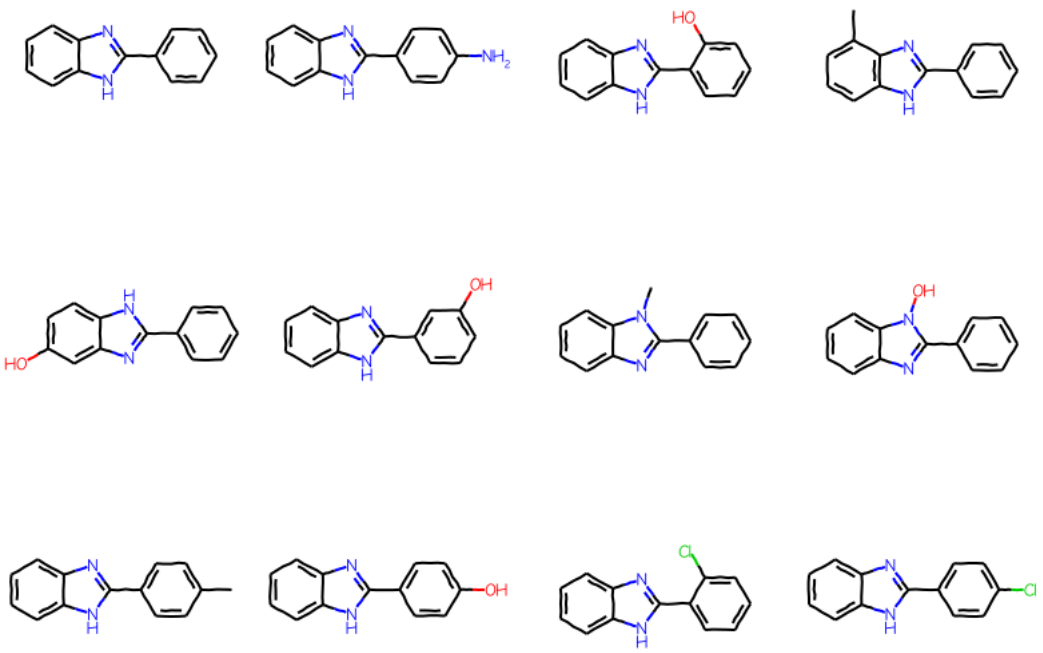
Initially the generic ARY query is not used in the substructure search:

```
>>> matches = sslib.GetMatches(bqry)
>>> mols = [sslib.GetMol(x) for x in matches]
>>> print(f'There are {len(mols)} matches')
>>> Draw.MolsToGridImage(mols[:12],molsPerRow=4)
There are 1000 matches
```



But we can limit the results to aryl substituents by expanding the generic query and telling the SubstructLib to use it:

```
>>> ary_qry = Chem.Mol(bqry)
    # expand the query:
>>> Chem.SetGenericQueriesFromProperties(ary_qry)
    # do the search using generic query matchers:
>>> params = Chem.SubstructMatchParameters()
>>> params.useGenericMatchers = True
>>> matches = sslib.GetMatches(ary_qry,params)

>>> mols = [sslib.GetMol(x) for x in matches]
>>> print(f'There are {len(mols)} matches')
>>> Draw.MolsToGridImage(mols[:12],molsPerRow=4)
There are 1000 matches
```



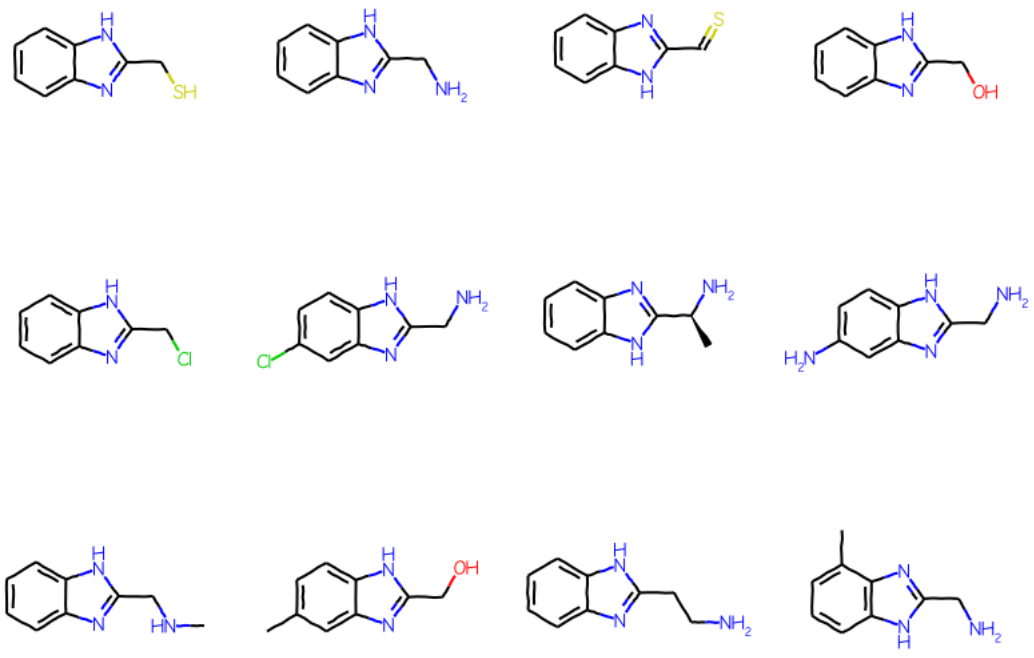
The full list of recognized generic atoms is here. Here are a couple of more examples:

1. AHC: heteroacyclic groups

```
>>> mb = '''
     Mrv2108 03242208122D

     0  0  0     0  0            999 V3000
M  V30 BEGIN CTAB
M  V30 COUNTS 10 11 1 0 0
M  V30 BEGIN ATOM
M  V30 1 C -11.4118 0.5479 0 0
M  V30 2 C -11.3849 -0.9919 0 0
M  V30 3 C -10.0379 -1.7385 0 0
M  V30 4 C -8.7179 -0.9454 0 0
M  V30 5 C -8.7448 0.5943 0 0
M  V30 6 C -10.0918 1.341 0 0
```

```
    M  V30 7 N -7.2453 -1.3956 0 0
    M  V30 8 C -6.362 -0.1342 0 0
    M  V30 9 N -7.2888 1.0958 0 0
    M  V30 10 C -4.8222 -0.161 0 0
    M  V30 END ATOM
    M  V30 BEGIN BOND
    M  V30 1 1 1 2
    M  V30 2 2 2 3
    M  V30 3 1 3 4
    M  V30 4 2 4 5
    M  V30 5 1 5 6
    M  V30 6 2 1 6
    M  V30 7 1 7 8
    M  V30 8 1 5 9
    M  V30 9 1 4 7
    M  V30 10 2 8 9
    M  V30 11 1 8 10
    M  V30 END BOND
    M  V30 BEGIN SGROUP
    M  V30 1 SUP 0 ATOMS=(1 10) SAP=(3 10 8 1) XBONDS=(1 11) LABEL=AHC
    M  V30 END SGROUP
    M  V30 END CTAB
    M  END
    '''
>>> bqry = Chem.MolFromMolBlock(mb)
    # show labels for the Sgroups:
>>> for sgs in Chem.GetMolSubstanceGroups(bqry):
...     if sgs.GetProp('TYPE') == 'SUP':
```

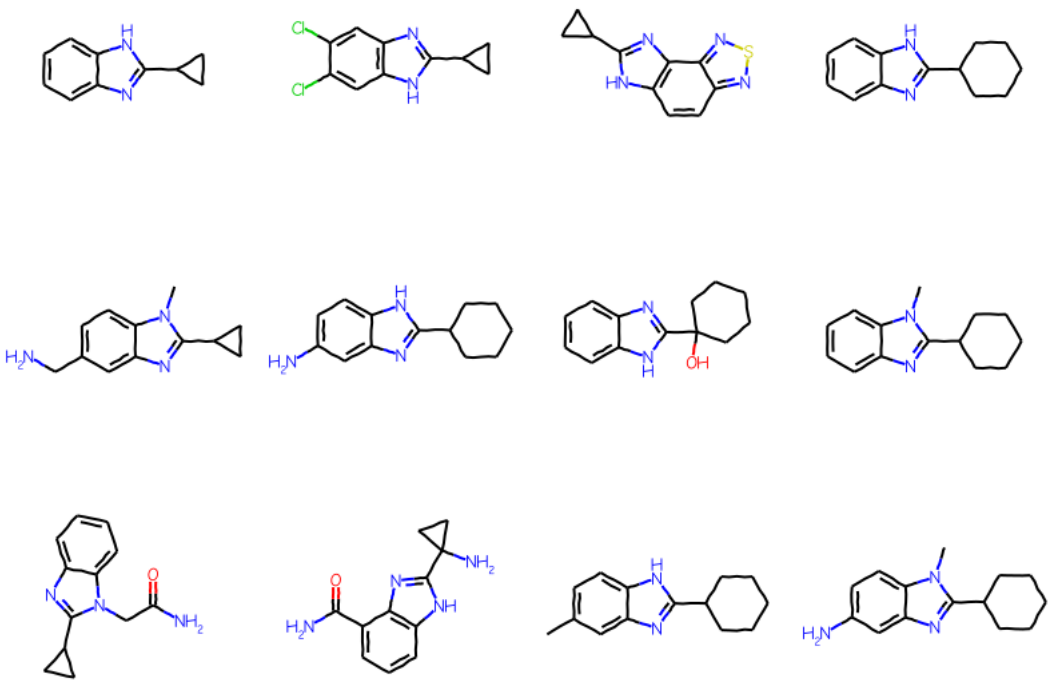
```
...         bqry.GetAtomWithIdx(sgs.GetAtoms()[0]).SetProp("atomLabel",
                                                    sgs.GetProp("LABEL"))
>>> ahc_qry = Chem.Mol(bqry)
    # expand the query:
>>> Chem.SetGenericQueriesFromProperties(ahc_qry)
    # do the search using generic query matchers:
>>> params = Chem.SubstructMatchParameters()
>>> params.useGenericMatchers = True
>>> matches = sslib.GetMatches(ahc_qry,params)
>>> mols = [sslib.GetMol(x) for x in matches]
>>> print(f'There are {len(mols)} matches')
>>> Draw.MolsToGridImage(mols[:12],molsPerRow=4)
There are 1000 matches
```



2. CAL: carbocyclic alkyl

```
>>> mb = '''
     Mrv2108 03242208122D

     0  0  0     0  0            999 V3000
   M  V30 BEGIN CTAB
   M  V30 COUNTS 10 11 1 0 0
   M  V30 BEGIN ATOM
   M  V30 1 C -11.4118 0.5479 0 0
   M  V30 2 C -11.3849 -0.9919 0 0
   M  V30 3 C -10.0379 -1.7385 0 0
   M  V30 4 C -8.7179 -0.9454 0 0
   M  V30 5 C -8.7448 0.5943 0 0
   M  V30 6 C -10.0918 1.341 0 0
   M  V30 7 N -7.2453 -1.3956 0 0
   M  V30 8 C -6.362 -0.1342 0 0
   M  V30 9 N -7.2888 1.0958 0 0
   M  V30 10 C -4.8222 -0.161 0 0
   M  V30 END ATOM
   M  V30 BEGIN BOND
   M  V30 1 1 1 2
   M  V30 2 2 2 3
   M  V30 3 1 3 4
   M  V30 4 2 4 5
   M  V30 5 1 5 6
   M  V30 6 2 1 6
   M  V30 7 1 7 8
   M  V30 8 1 5 9
   M  V30 9 1 4 7
```

```
    M  V30 10 2 8 9
    M  V30 11 1 8 10
    M  V30 END BOND
    M  V30 BEGIN SGROUP
    M  V30 1 SUP 0 ATOMS=(1 10) SAP=(3 10 8 1) XBONDS=(1 11) LABEL=CAL
    M  V30 END SGROUP
    M  V30 END CTAB
    M  END
    '''
>>> bqry = Chem.MolFromMolBlock(mb)
>>> cal_qry = Chem.Mol(bqry)
    # expand the query:
>>> Chem.SetGenericQueriesFromProperties(cal_qry)
    # do the search using generic query matchers:
>>> params = Chem.SubstructMatchParameters()
>>> params.useGenericMatchers = True
>>> matches = sslib.GetMatches(cal_qry,params)

>>> mols = [sslib.GetMol(x) for x in matches]
>>> print(f'There are {len(mols)} matches')
>>> Draw.MolsToGridImage(mols[:12],molsPerRow=4)
There are 784 matches
```

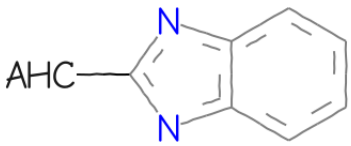


Part 3. Providing the queries in CXSMILES

It is also possible to use CXSMILES/CXSMARTS to provide these queries:

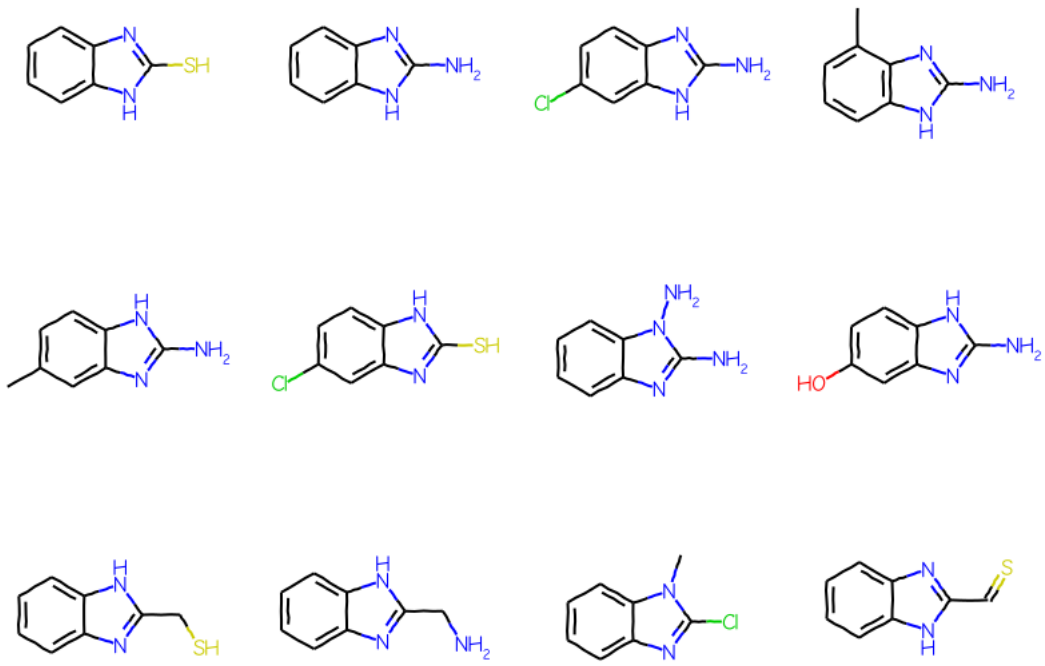
AHC: heteroacyclic

```
>>> sqry = Chem.MolFromSmarts('*-c1nc2c(n1)cccc2 |$AHC;;;;;;;;;$|')
>>> sqry
```




```
>>> ahc_qry = Chem.Mol(sqry)
    # expand the query:
```

```
>>> Chem.SetGenericQueriesFromProperties(ahc_qry)
    # do the search using generic query matchers:
>>> params = Chem.SubstructMatchParameters()
>>> params.useGenericMatchers = True
>>> matches = sslib.GetMatches(ahc_qry,params)
>>> mols = [sslib.GetMol(x) for x in matches]
>>> print(f'There are {len(mols)} matches')
>>> Draw.MolsToGridImage(mols[:12],molsPerRow=4)
There are 1000 matches
```



Though note that if you use CXSMILES it's important to make sure the dummy atom is replaced with a query before doing the search. If you don't do so, you'll get no results since the dummy atom only matches other dummy atoms:

```
>>> sqry = Chem.MolFromSmiles('*C1=NC2=C(N1)C=CC=C2 |$AHC;;;;;;;;;$|')
>>> ahc_qry = Chem.Mol(sqry)
```

```
   # expand the query:
>>> Chem.SetGenericQueriesFromProperties(ahc_qry)
>>> params = Chem.SubstructMatchParameters()
>>> params.useGenericMatchers = True
>>> matches = sslib.GetMatches(ahc_qry,params)
>>> print(f'There are {len(matches)} matches')
There are 0 matches
```

Here's the fix:

```
>>> qps = Chem.AdjustQueryParameters.NoAdjustments()
>>> qps.makeDummiesQueries = True
>>> sqry = Chem.AdjustQueryProperties(sqry)
>>> ahc_qry = Chem.Mol(sqry)
>>> Chem.SetGenericQueriesFromProperties(ahc_qry)
>>> params = Chem.SubstructMatchParameters()
>>> params.useGenericMatchers = True
>>> matches = sslib.GetMatches(ahc_qry,params)
>>> print(f'There are {len(matches)} matches')
There are 186 matches
```

Part 4. Search performance

Get a baseline for how long a search takes by ignoring the generic matchers:

```
>>> params = Chem.SubstructMatchParameters()
>>> params.useGenericMatchers = False
>>> %timeit matches = sslib.GetMatches(cal_qry,params,maxResults=5000)
329 ms ± 6.11 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
```

We'd expect the searches using generic groups to take at least a bit longer since they will results in more molecules being scanned until we find our 5000 results. The big question is

whether or not merely including the generic queries results in a significant slow down. Let's test how much longer it takes using two of the different generic groups:

```
    # Start with "CAL"
>>> params = Chem.SubstructMatchParameters()
>>> params.useGenericMatchers = True
>>> %timeit matches = sslib.GetMatches(cal_qry,params,maxResults=5000)
415 ms ± 18.3 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
```

That really doesn't make much of a difference at all.

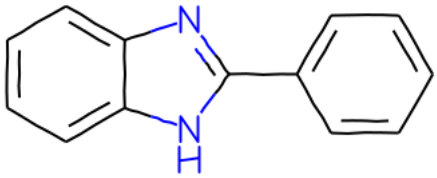
```
    # "ARY"
>>> params = Chem.SubstructMatchParameters()
>>> params.useGenericMatchers = True
>>> %timeit matches = sslib.GetMatches(ary_qry,params,maxResults=5000)
1.24 s ± 139 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
```

So there is definitely some impact, and it depends on which generic query is being used.

Part 5. Aside

Since the ARY query mainly returns molecules which have a phenyl group attached at the ARY position, let's compare search performance with a query where we explicitly include the phenyl. Here the extra atoms/bonds will slow the substructure search down but they will also make the fingerprint screenout more effective. The question is which effect is dominant.

```
>>> fullqry = Chem.MolFromSmiles('c1ccccc1-c1nc2ccccc2[nH]1')
>>> fullqry
```

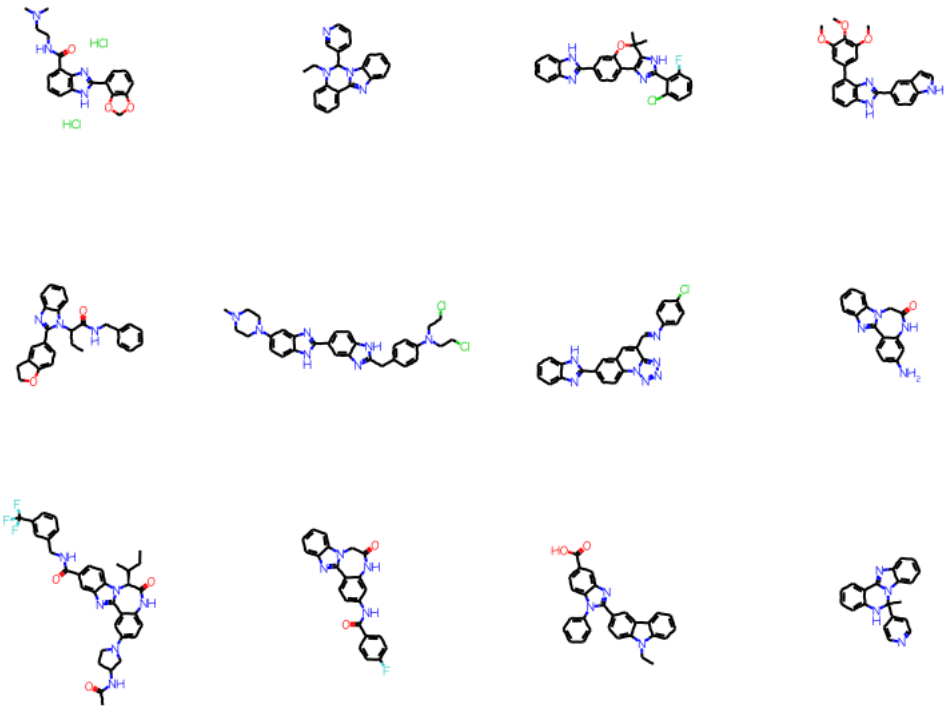


```
>>> %timeit matches = sslib.GetMatches(fullqry,maxResults=5000)
1.05 s ± 54.3 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
```

In this case the larger query is a bit faster.

It's worth mentioning that the query with the phenyl group returns different search results than the version with the ARY:

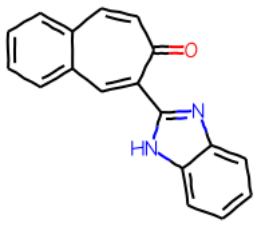
```
>>> full_matches = sslib.GetMatches(fullqry,params,maxResults=-1)
>>> ary_matches = sslib.GetMatches(ary_qry,params,maxResults=-1)
>>> len(full_matches),len(ary_matches)
(6866, 6488)
>>> full_not_ary = set(full_matches).difference(ary_matches)
>>> mols = [sslib.GetMol(x) for x in full_not_ary]
>>> print(f'There are {len(mols)} matches')
>>> Draw.MolsToGridImage(mols[:12],molsPerRow=4)
There are 379 matches
```



As you'd expect, there are a bunch of molecules with heterocyclic aromatic systems here. The ARY query only matches carboaryl systems.

There is only one molecule returned by the ARY query but not the full query; this one has an unusual aromatic ring system:

```
>>> ary_not_full = set(ary_matches).difference(full_matches)
>>> mols = [sslib.GetMol(x) for x in ary_not_full]
>>> print(f'There are {len(mols)} matches')
>>> Draw.MolsToGridImage(mols[:12],molsPerRow=4)
There are 1 matches
```



That's a good one to ignite a round of "but that's not aromatic!" arguments, but before you start down that road, please read the section on RDKit webpage and aromaticity in the Daylight Theory manual.

Similarity maps with the new drawing code

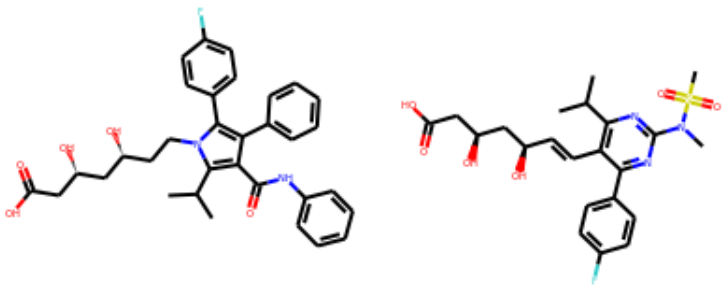
This tutorial is about a new and more detailes way to generate similarity maps.

```
>>> from rdkit import Chem
>>> from rdkit.Chem import Draw
>>> from rdkit.Chem.Draw import SimilarityMaps
>>> from IPython.display import SVG
>>> import io
>>> from PIL import Image
>>> import numpy as np
>>> import rdkit
>>> print(rdkit.__version__)
2019.09.2
```

Let' start by using the "classic" similarity map functionality to show why atorvastatin (Lipitor) and rosuvastatin (Crestor) are similar to each other when using the Morgan fingerprint.

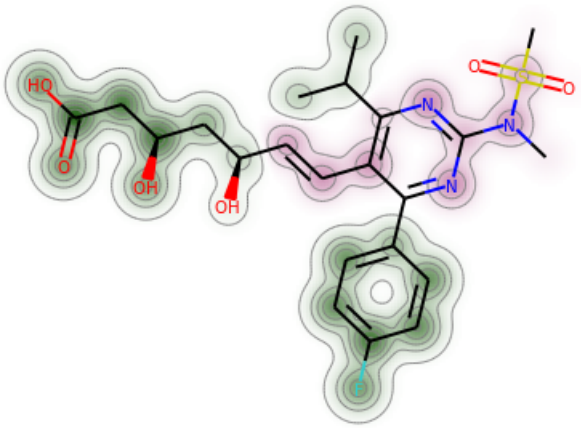
Here are the two molecules:

```
>>> atorvastatin = Chem.MolFromSmiles('O=C(O)C[C@H](O)C[C@H](O)CCn2c(c(c(c2c1 \
                                      ccc(F)cc1)c3ccccc3)C(=O)Nc4ccccc4)C(C)C')
>>> rosuvastatin = Chem.MolFromSmiles('OC(=O)C[C@H](O)C[C@H](O)\C=C\c1c \
                                   (C(C)C)nc(N(C)S(=O)(=O)C)nc1c2ccc(F)cc2')
>>> Draw.MolsToGridImage((atorvastatin,rosuvastatin))
```



To use the new drawing code, we create a Draw2D object and pass that to SimilarityMaps.GetSimilarityMapForFingerprint:

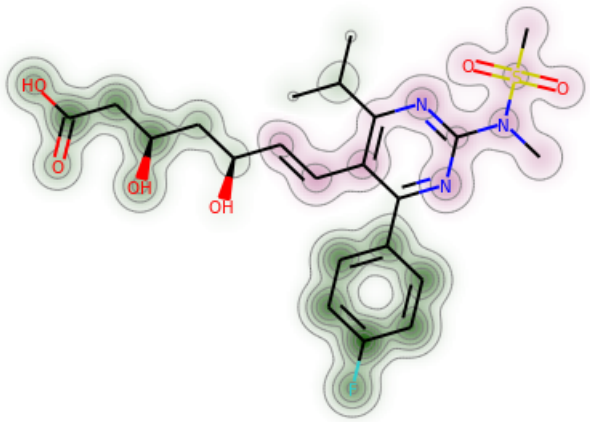
```
def show_png(data):
    bio = io.BytesIO(data)
    img = Image.open(bio)
    return img


>>> d = Draw.MolDraw2DCairo(400, 400)
>>> _, maxWeight = SimilarityMaps.GetSimilarityMapForFingerprint(atorvastatin,
                        rosuvastatin, lambda m, i:
                        SimilarityMaps.GetMorganFingerprint(m,
                        i, radius=2, fpType='bv'), draw2d=d)
>>> d.FinishDrawing()
>>> show_png(d.GetDrawingText())
```



We can do the same thing with count-based fingerprints:

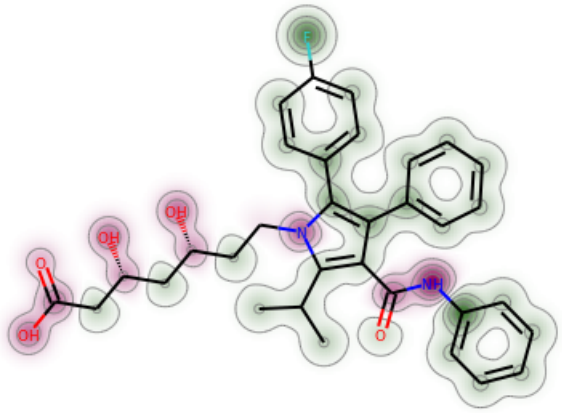
```
>>> d = Draw.MolDraw2DCairo(400, 400)
>>> _, maxWeight = SimilarityMaps.GetSimilarityMapForFingerprint(atorvastatin,
```

```
                                rosuvastatin, lambda m, i:
                                SimilarityMaps.GetMorganFingerprint(m, i,
                                radius=2, fpType='count'), draw2d=d)
>>> d.FinishDrawing()
>>> show_png(d.GetDrawingText())
```



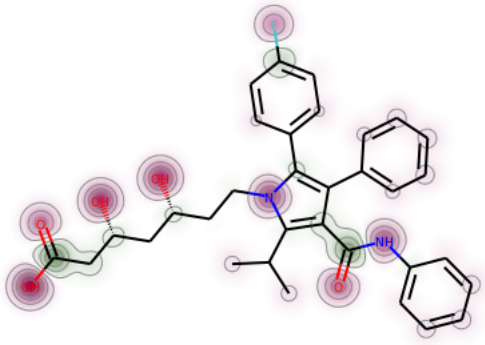
The other GetSimilarityMapFrom... functions also accept the optional draw2d argument. Here's a visualization of the contributions made by the atoms in atorvastatin to its calculatied logp value:

```
>>> from rdkit.Chem import rdMolDescriptors
>>> ator_contribs = rdMolDescriptors._CalcCrippenContribs(atorvastatin)
>>> d = Draw.MolDraw2DCairo(400, 400)
>>> SimilarityMaps.GetSimilarityMapFromWeights(atorvastatin,
                        [x[0] for x in ator_contribs],draw2d=d)
>>> d.FinishDrawing()
>>> show_png(d.GetDrawingText())
```



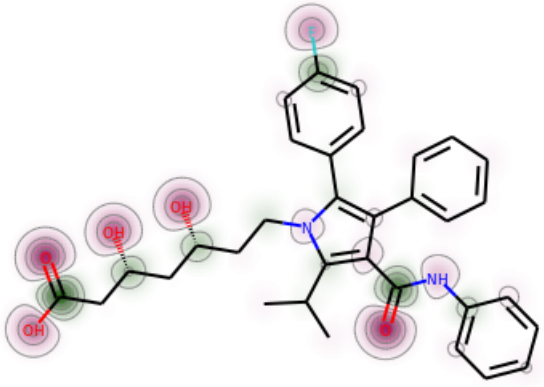
And a couple more visualizations of various partial charge schemes. Starting with Gasteiger-Marsilli charges:

```
>>> from rdkit.Chem import rdPartialCharges
>>> rdPartialCharges.ComputeGasteigerCharges(atorvastatin)
>>> chgs = [x.GetDoubleProp("_GasteigerCharge") for x in atorvastatin.GetAtoms()]
>>> d = Draw.MolDraw2DCairo(400, 400)
>>> SimilarityMaps.GetSimilarityMapFromWeights(atorvastatin,chgs,draw2d=d)
>>> d.FinishDrawing()
>>> show_png(d.GetDrawingText())
```



And also the partial charges calculated with extended Hueckel theory (eHT) using Mulliken analysis:

```
>>> from rdkit.Chem import rdEHTTools
>>> from rdkit.Chem import rdDistGeom
>>> mh = Chem.AddHs(atorvastatin)
>>> rdDistGeom.EmbedMolecule(mh)
>>> _,res = rdEHTTools.RunMol(mh)
>>> static_chgs = res.GetAtomicCharges()[:atorvastatin.GetNumAtoms()]
>>> d = Draw.MolDraw2DCairo(400, 400)
>>> SimilarityMaps.GetSimilarityMapFromWeights(atorvastatin,
                              list(static_chgs),draw2d=d)
>>> d.FinishDrawing()
>>> show_png(d.GetDrawingText())
```

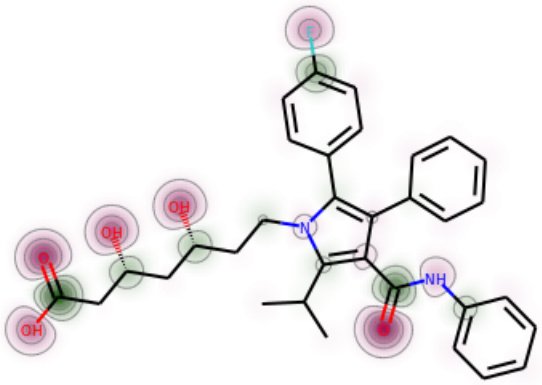


As one final demo, I'll use the method to visualize the variability of the eHT charges with conformation for atorvastatin.

Start by generating 10 diverse conformers, calculating the charges for each, and plotting the average:

```
>>> mh = Chem.AddHs(atorvastatin)
>>> ps = rdDistGeom.ETKDGv2()
>>> ps.pruneRmsThresh = 0.5
>>> ps.randomSeed = 0xf00d
>>> rdDistGeom.EmbedMultipleConfs(mh,10,ps)
>>> print(f'Found {mh.GetNumConformers()} conformers')
>>> chgs = []
>>> for conf in mh.GetConformers():
...     _,res = rdEHTTools.RunMol(mh,confId=conf.GetId())
...     chgs.append(res.GetAtomicCharges()[:atorvastatin.GetNumAtoms()])
>>> chgs = np.array(chgs)
>>> mean_chgs = np.mean(chgs,axis=0)
>>> std_chgs = np.std(chgs,axis=0)
```

```
>>> d = Draw.MolDraw2DCairo(400, 400)
>>> SimilarityMaps.GetSimilarityMapFromWeights(atorvastatin,
                               list(mean_chgs),draw2d=d)
>>> d.FinishDrawing()
>>> show_png(d.GetDrawingText())
Found 10 conformers
```

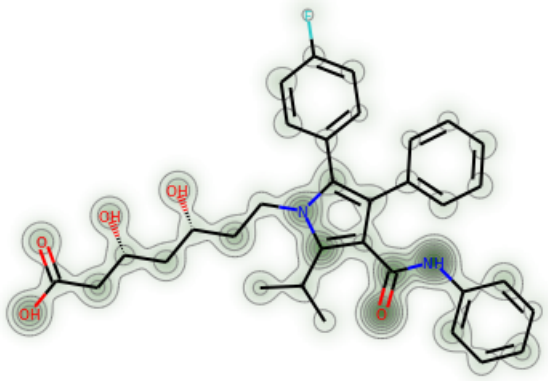


That doesn't look hugely different from what we saw above.

To show the variability, plot the standard deviation of the charges across the 10 conformers:

```
>>> print(std_chgs)
>>> print(max(std_chgs),min(std_chgs))
>>> d = Draw.MolDraw2DCairo(400, 400)
>>> SimilarityMaps.GetSimilarityMapFromWeights(atorvastatin,
                               list(std_chgs),draw2d=d)
>>> d.FinishDrawing()
>>> show_png(d.GetDrawingText())
[0.01292592 0.00743163 0.01971312 0.01433223 0.01063085 0.01283745
 0.01219511 0.00748435 0.01234194 0.01492494 0.00640842 0.02264999
```

```
 0.02481744 0.00987842 0.00843151 0.01289956 0.00560632 0.00498617
 0.00702613 0.00634237 0.00699789 0.00539868 0.00521868 0.02412709
 0.03131741 0.03709349 0.00657276 0.01175903 0.00674661 0.01012909
 0.0050995  0.01139418 0.00831795 0.00581207 0.00960073]
0.03709348867462464 0.00452067345998171
```



The deviations aren't huge (the printed array shows that), but the largest value is clearly the amide N. There's definitely a ToDo here to improve the way the negative contours are drawn so that the fact that they are being drawn with dashed lines is visible.

3D maximum common substructure

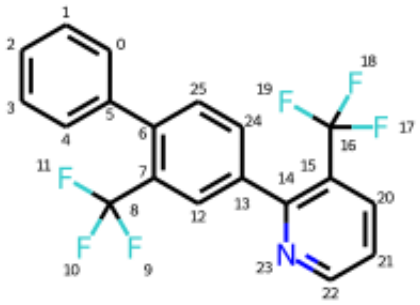
One of the "underdocumented", and perhaps lesser known, features of the RDKit MCS code is the ability to take atomic coordinates into account when generating the MCS. The idea here is to find the MCS between a set of 3D molecules where the distance between potential matching atoms is taken into account.

This tutorial shows how to do it.

```
>>> from rdkit import Chem
>>> from rdkit.Chem import rdDistGeom
>>> from rdkit.Chem import rdFMCS
>>> from rdkit.Chem import Draw
>>> from rdkit.Chem.Draw import IPythonConsole
>>> from rdkit.Chem import rdDepictor
>>> rdDepictor.SetPreferCoordGen(True)
>>> IPythonConsole.ipython_3d = True
>>> import rdkit
>>> print(rdkit.__version__)
2022.03.3
```

Let's start with an artifical example as a demo:

```
>>> m1 = Chem.MolFromSmiles('c1ccccc1-c1c(C(F)(F)F)cc(-c2c(C(F)(F)F)cccn2)cc1')
>>> IPythonConsole.drawOptions.addAtomIndices = True
>>> m1
```



Generate a conformer:

```
m1 = Chem.AddHs(m1)
rdDistGeom.EmbedMolecule(m1,randomSeed=0xf00d)
m1 = Chem.RemoveHs(m1)
```

Clip out the central ring and change one of the atoms to an N

```
>>> m2 = Chem.RWMol(m1)
>>> keep = [6,7,12,13,24,25]
>>> remove = set(range(m2.GetNumAtoms())).difference(keep)
>>> m2.BeginBatchEdit()
>>> for aidx in remove:
...     m2.RemoveAtom(aidx)
>>> m2.CommitBatchEdit()
>>> m2.GetAtomWithIdx(0).SetAtomicNum(7)
```

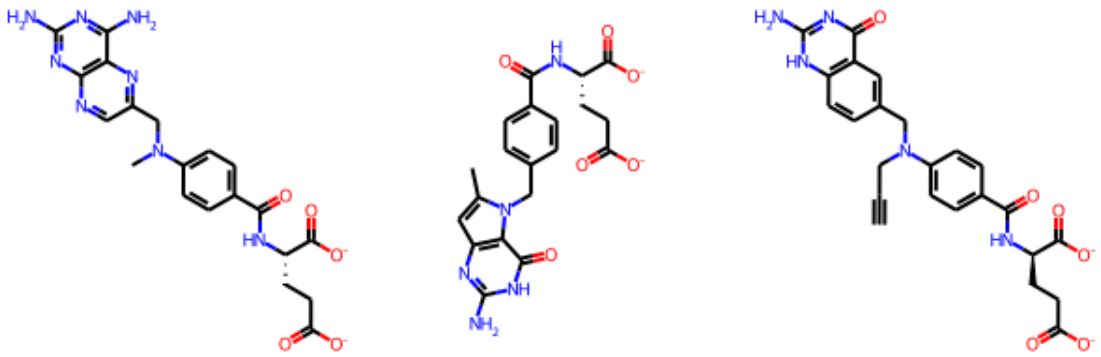
A normal MCS will, of course, match this to the N-containing ring:

```
>>> ps = rdFMCS.MCSParameters()
>>> res = rdFMCS.FindMCS([m1,m2],ps)
>>> print(res.smartsString)
[#7]1:[#6]:[#6]:[#6]:[#6]:[#6]:1
>>> ps = rdFMCS.MCSParameters()
>>> ps.AtomCompareParameters.MaxDistance = 0.5
>>> res = rdFMCS.FindMCS([m1,m2],ps)
>>> print(res.smartsString)
[#6]:[#6]:[#6]:[#6]:[#6]
>>> ps = rdFMCS.MCSParameters()
>>> ps.AtomCompareParameters.MaxDistance = 0.5
>>> ps.AtomTyper = rdFMCS.AtomCompare.CompareAny
>>> res = rdFMCS.FindMCS([m1,m2],ps)
```

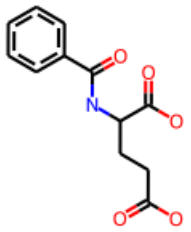
```
>>> print(res.smartsString)
[#7,#6]1:[#6]:[#6]:[#6]:[#6]:[#6]:1
```

A real example

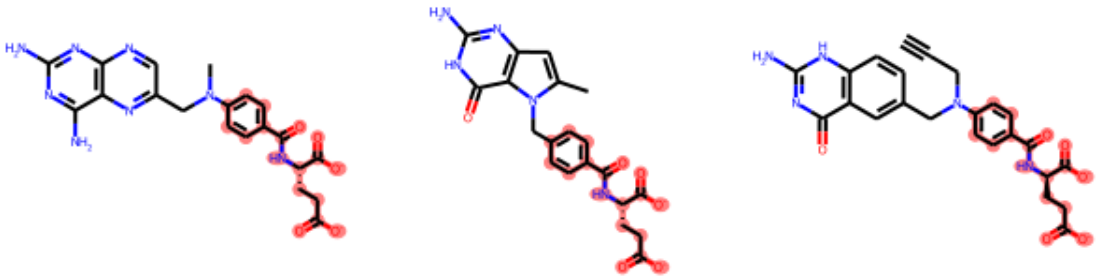
```
# exported from the binding db and converted to SDF in pymol
>>> s = [x for x in Chem.ForwardSDMolSupplier('../data/1TDU-results.sdf')]
>>> ms2d = [Chem.Mol(x) for x in ms]
>>> for m in ms2d:
...     rdDepictor.Compute2DCoords(m)
>>> IPythonConsole.drawOptions.addAtomIndices = False
>>> Draw.MolsToGridImage(ms2d)
```



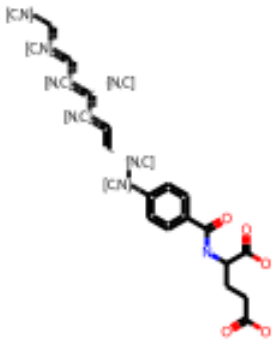
```
>>> import py3Dmol
>>> viewer = py3Dmol.view(width=350, height=350)
>>> IPythonConsole.addMolToView(ms[0],viewer)
>>> IPythonConsole.addMolToView(ms[1],viewer)
>>> IPythonConsole.addMolToView(ms[2],viewer)
>>> ps = rdFMCS.MCSParameters()
>>> res = rdFMCS.FindMCS(ms,ps)
>>> qry = Chem.MolFromSmarts(res.smartsString)
>>> qry
```



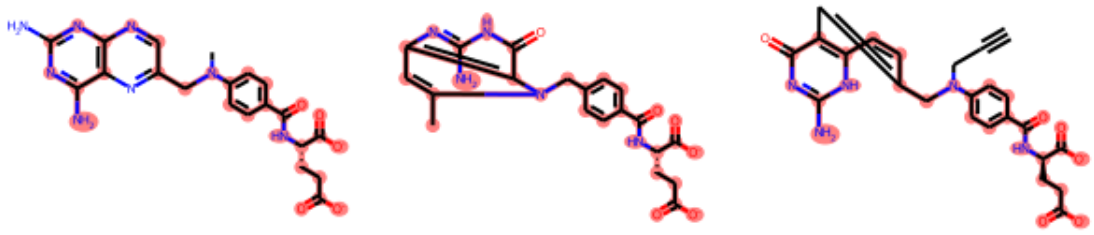
```
>>> matches = [x.GetSubstructMatch(qry) for x in ms2d]
>>> conf = Chem.Conformer(qry.GetNumAtoms())
>>> for i,mi in enumerate(matches[0]):
...     conf.SetAtomPosition(i,ms2d[0].GetConformer().GetAtomPosition(mi))
>>> qry.AddConformer(conf)
>>> rdDepictor.SetPreferCoordGen(False)
>>> for m in ms2d:
...     rdDepictor.GenerateDepictionMatching2DStructure(m,qry)
>>> rdDepictor.SetPreferCoordGen(True)
>>> Draw.MolsToGridImage(ms2d,highlightAtomLists=matches)
```



```
>>> ps = rdFMCS.MCSParameters()
>>> ps.AtomTyper = rdFMCS.AtomCompare.CompareAny
>>> res = rdFMCS.FindMCS(ms,ps)
>>> qry = Chem.MolFromSmarts(res.smartsString)
>>> qry
```



```
>>> matches = [x.GetSubstructMatch(qry) for x in ms2d]
>>> conf = Chem.Conformer(qry.GetNumAtoms())
>>> for i,mi in enumerate(matches[0]):
...     conf.SetAtomPosition(i,ms2d[0].GetConformer().GetAtomPosition(mi))
>>> qry.AddConformer(conf)
>>> rdDepictor.SetPreferCoordGen(False)
>>> for m in ms2d:
...     rdDepictor.GenerateDepictionMatching2DStructure(m,qry)
>>> rdDepictor.SetPreferCoordGen(True)
>>> Draw.MolsToGridImage(ms2d,highlightAtomLists=matches)
```

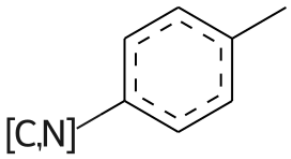


This is an example where the constrained coordinates, which only match part of a ring system, cause problems.

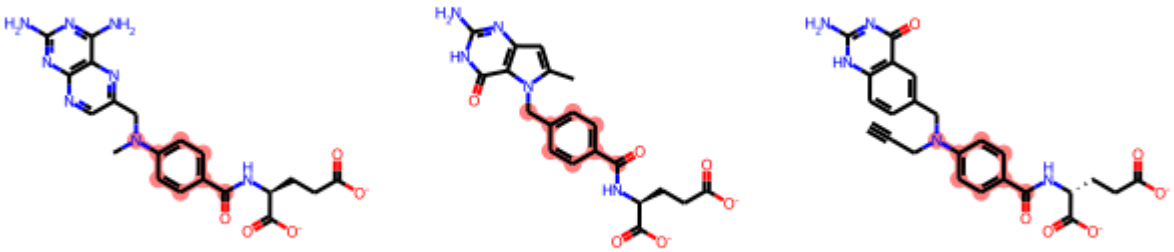
Both of those MCS results are matching significant parts of the molecules, but we saw that the molecules didn't actually align quite that well.

What about if we take atom coordinates into account?

```
>>> ps = rdFMCS.MCSParameters()
>>> ps.AtomCompareParameters.MaxDistance = 1.0
>>> ps.AtomTyper = rdFMCS.AtomCompare.CompareAny
>>> res = rdFMCS.FindMCS(ms,ps)
>>> qry = Chem.MolFromSmarts(res.smartsString)
>>> qry
```




```
>>> matches = [x.GetSubstructMatch(qry) for x in ms2d]
>>> conf = Chem.Conformer(qry.GetNumAtoms())
>>> for i,mi in enumerate(matches[0]):
...     conf.SetAtomPosition(i,ms2d[0].GetConformer().GetAtomPosition(mi))
>>> qry.AddConformer(conf)
>>> for m in ms2d:
...     rdDepictor.GenerateDepictionMatching2DStructure(m,qry)
>>> Draw.MolsToGridImage(ms2d,highlightAtomLists=matches)
```



The MCS gave us a SMARTS which matches, but unfortunately it does not provide the matching atoms. Finding those via substructure search would be easy if we could assume

that the MCS only matches each molecule once, but that's not always going to be the case. This is actually one of those examples. Let's look at how many times the core can match each of the molecules:

```
>>> allMatches = []
>>> for m in ms:
...     allMatches.append(m.GetSubstructMatches(qry,uniquify=False))
>>> allMatches
[((13, 15, 17, 16, 20, 19, 18, 21),
 (13, 15, 18, 19, 20, 16, 17, 21)),
 ((12, 13, 15, 14, 18, 17, 16, 19),
  (12, 13, 16, 17, 18, 14, 15, 19),
  (19, 18, 14, 15, 13, 16, 17, 12),
  (19, 18, 17, 16, 13, 15, 14, 12)),
 ((13, 14, 17, 18, 19, 20, 21, 22),
 (13, 14, 21, 20, 19, 18, 17, 22))]
```

Now define a function which goes through all the possible substructure matches and finds the one which satisfies the 3D distance constraints on the core:

```
def getAlignedSubstructMatch(ms,qry,distTol=1.0):
    allMatches = []
    for m in ms:
        allMatches.append(m.GetSubstructMatches(qry,uniquify=False))

    keepMatches = []
    for match0 in allMatches[0]:
        allMatched = True
        for i in range(1,len(ms)):
            imatched = False
```

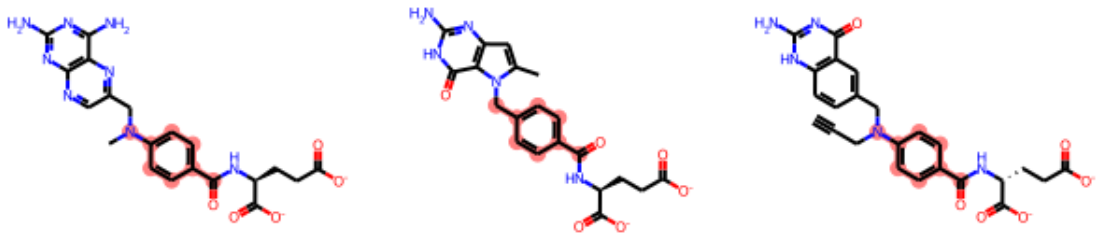
```
            for matchi in allMatches[i]:
                matched = True
                for i0,ii in zip(match0,matchi):
                    dist = (ms[0].GetConformer().GetAtomPosition(i0) -
                    ms[i].GetConformer().GetAtomPosition(ii)).Length()
                    if dist > distTol:
                        matched = False
                        break
                if matched:
                    keepMatches.append(matchi)
                    imatched = True
                    break
            if not imatched:
                allMatched = False
                keepMatches = []
                break
        if allMatched:
            keepMatches = [match0] + keepMatches
            break
        else:
            keepMatches = []
    return keepMatches
```

The results for our molecules and the 3D MCS core:

```
>>> keepMatches = getAlignedSubstructMatch(ms,qry)
>>> keepMatches
[(13, 15, 17, 16, 20, 19, 18, 21),
 (12, 13, 15, 14, 18, 17, 16, 19),
 (13, 14, 17, 18, 19, 20, 21, 22)]
```

And highlight the substructures:

```
>>> Draw.MolsToGridImage(ms2d,highlightAtomLists=keepMatches)
```



Let's redraw the molecules in 3D and highlight the atoms involved in the MCS. It's nice to see, plus I learned some stuff about how to use py3Dmol while doing it... so that's a bonus.

```
>>> import py3Dmol
>>> viewer = py3Dmol.view(width=450, height=450)
>>> IPythonConsole.addMolToView(ms[0],viewer)
>>> IPythonConsole.addMolToView(ms[1],viewer)
>>> IPythonConsole.addMolToView(ms[2],viewer)
>>> for idx,clr in zip((-1,-2,-3),('redCarbon','cyanCarbon','blueCarbon')):
...     viewer.setStyle({'model':idx,},
                        {'stick':{'colorscheme':clr,'radius':.15}})
...     viewer.setStyle({'model':idx,'serial':keepMatches[idx]},
                        {'stick':{'colorscheme':clr},
                         'sphere':{'colorscheme':clr,'radius':.5}})
>>> viewer.zoomTo()
>>> viewer.show()
```

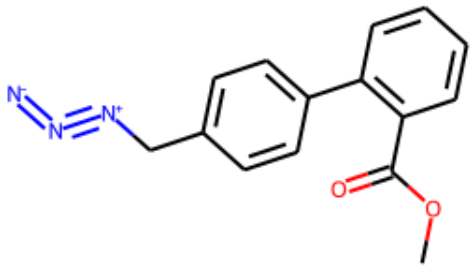
Using single-molecule reactions

This tutorial provides a quick introduction to a piece of chemical reaction functionality which was added to the 2021.09 RDKit release.

```
>>> from rdkit import Chem
>>> from rdkit.Chem import rdChemReactions
>>> from rdkit.Chem.Draw import IPythonConsole
>>> from rdkit.Chem import Draw
>>> import rdkit
>>> print(rdkit.__version__)
2021.09.3
```

Let's use this molecule as an example:

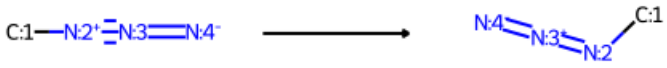
```
>>> m1 = Chem.MolFromSmiles('COC(=O)C1=C(C=CC=C1)C1=CC=C(C[N+]#[N]=[N-])C=C1',
                            sanitize=False)
>>> m1
```



The RDKit won't accept this with default settings because there's an odd representation of an azide group which includes a five-valent neutral nitrogen.
It's straight forward to define a reaction which can convert this odd azide form to the more normal variant:

```
>>> tf1 = rdChemReactions.ReactionFromSmarts('[#6:1]-[N+:2]#[N:3]=[N-:4] \
                                             >> \
```

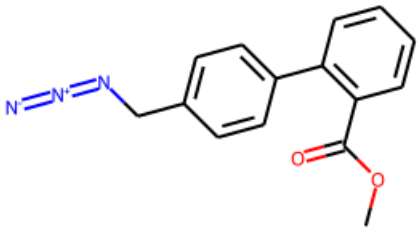
```
                                                    [#6:1]-[N+0:2]=[N+1:3]=[N-:4]')
>>> tf1
```



The usual way to use this would be with the RunReactants() method, which returns a list of lists of new molecules. In this case though, we have a reaction which operates on a single reactant and has a single product, so we can take advantage of the new RunReactantInPlace() method.

As the method name implies, this modifies the reactant molecule in place instead of creating new molecules which are returned as products:

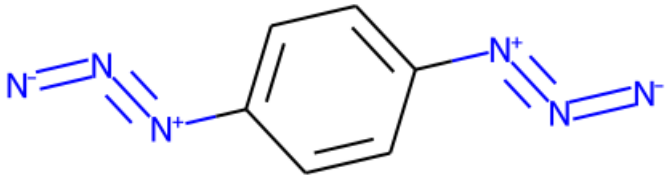
```
>>> tf1.RunReactantInPlace(m1)
# now sanitize the molecule so that we do chemistry perception
" and can get decent drawings:
>>> Chem.SanitizeMol(m1)
>>> m1
```



It's important to note that his only modifies one match at a time, so if we have multiple functional groups which need to be modified, we'll need to call RunReactantInPlace() multiple times.

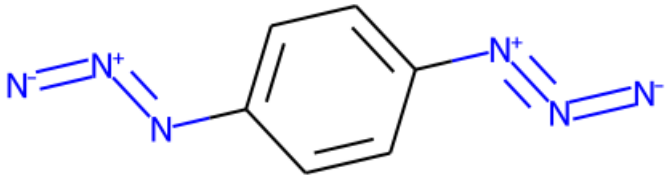
Here's a demonstration of that using a molecule which has two of these weird azide constructions.

```
>>> m1 = Chem.MolFromSmiles('c1cc([N+]#[N]=[N-])ccc1[N+]#[N]=[N-]',
                            sanitize=False)
>>> m1
```



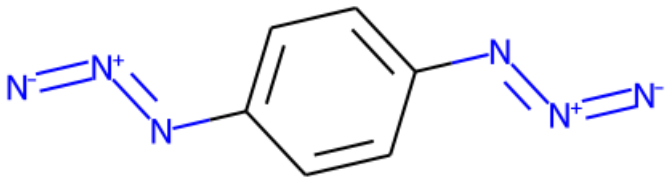
The first application of RunReactantInPlace() changes one of the groups:

```
>>> tf1.RunReactantInPlace(m1)
>>> m1
```



If we call RunReactantInPlace() again, the second occurance is replaced:

```
>>> tf1.RunReactantInPlace(m1)
>>> m1
```

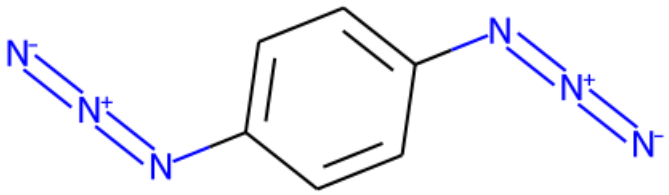


RunReactantInPlace() makes it easy to know when to stop because it returns a boolean letting you know whether or not the molecule was modified. So in this case we'll get false:

```
>>> tf1.RunReactantInPlace(m1)
False
```

This makes it easy to do all the transformations to a molecule with a while loop:

```
>>> m1 = Chem.MolFromSmiles('c1cc([N+]#[N]=[N-])ccc1[N+]#[N]=[N-]',
                            sanitize=False)
>>> while tf1.RunReactantInPlace(m1):
...     pass
>>> Chem.SanitizeMol(m1)
>>> m1
```



RunReactantInPlace() is limited, it can only be used with reactions which only have one reactant and product and which do not add atoms in the product.

```
>>> tf2 = rdChemReactions.ReactionFromSmarts('[#6:1]-[NH2:2]>>[#6:1]-[NH2:2]C')
>>> tmp = Chem.MolFromSmiles('CCN')
>>> tf2.RunReactantInPlace(tmp)
ValueError: ChemicalParserException: single component reactions which add atoms
in the product are not supported
```

Note that it can be used with reactions which remove atoms:

```
>>> tf2 = rdChemReactions.ReactionFromSmarts('[#6:1]-[NH2:2]>>[#6:1]')
>>> tmp = Chem.MolFromSmiles('CCN')
>>> tf2.RunReactantInPlace(tmp)
True
>>> tmp
```

Aside from being easier to use when working with this simple transformations, it's worth pointing out that RunReactantInPlace() is significantly faster than using RunReactants() with the same reaction:

```
>>> smi = 'COC(=O)C1=C(C=CC=C1)C1=CC=C(C[N+]#[N]=[N-])C=C1'
>>> m1 = Chem.MolFromSmiles(smi,sanitize=False)
>>> %timeit tf1.RunReactantInPlace(Chem.Mol(m1))
9.93 µs ± 125 ns per loop (mean ± std. dev. of 7 runs, 100000 loops each)
>>> smi = 'COC(=O)C1=C(C=CC=C1)C1=CC=C(C[N+]#[N]=[N-])C=C1'
>>> m1 = Chem.MolFromSmiles(,sanitize=False)
>>> %timeit tf1.RunReactants((Chem.Mol(m1),))
22.6 µs ± 81.2 ns per loop (mean ± std. dev. of 7 runs, 10000 loops each)
```

Highlighting changing atoms and bonds in reactions

This tutorial is about highlighting the bonds which changed in a reaction. Fair warning: This one is heavy on code and light on words.

```
>>> from rdkit import Chem
>>> from rdkit.Chem import Draw
>>> from rdkit.Chem.Draw import IPythonConsole
>>> from rdkit.Chem import rdChemReactions
>>> import rdkit
>>> print(rdkit.__version__)
2021.09.2
```

Let's consider this reaction:

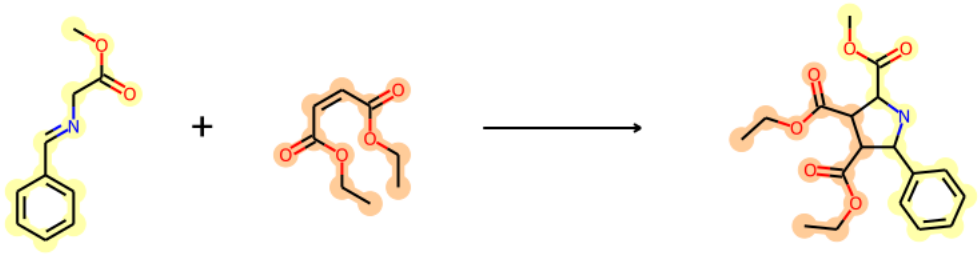
```
>>> rxn1 = rdChemReactions.ReactionFromRxnBlock('''$RXN

     Mrv2102  111820212128

  2  1
$MOL

  Mrv2102 11182121282D

 13 13  0  0  0  0            999 V2000
   -7.5723    2.6505    0.0000 C   0  0  0  0  0  0  0  0  0  1  0  0
   -6.8579    2.2380    0.0000 O   0  0  0  0  0  0  0  0  0  2  0  0
   -6.8580    1.4130    0.0000 C   0  0  0  0  0  0  0  0  0  3  0  0
   -6.1435    1.0004    0.0000 O   0  0  0  0  0  0  0  0  0  4  0  0
   -7.5725    1.0005    0.0000 C   0  0  0  0  0  0  0  0  0  5  0  0
   -7.5725    0.1755    0.0000 N   0  0  0  0  0  0  0  0  0  6  0  0
```

```
   -8.2869   -0.2369    0.0000 C   0  0  0  0  0  0  0  0  0  7  0  0
   -8.2870   -1.0620    0.0000 C   0  0  0  0  0  0  0  0  0  8  0  0
   -9.0015   -1.4745    0.0000 C   0  0  0  0  0  0  0  0  0  9  0  0
   -9.0015   -2.2995    0.0000 C   0  0  0  0  0  0  0  0  0 10  0  0
   -8.2870   -2.7120    0.0000 C   0  0  0  0  0  0  0  0  0 11  0  0
   -7.5726   -2.2995    0.0000 C   0  0  0  0  0  0  0  0  0 12  0  0
   -7.5726   -1.4745    0.0000 C   0  0  0  0  0  0  0  0  0 13  0  0
  1  2  1  0  0  0  0
  2  3  1  0  0  0  0
  3  4  2  0  0  0  0
  3  5  1  0  0  0  0
  5  6  1  0  0  0  0
  6  7  2  0  0  0  0
  7  8  1  0  0  0  0
  8  9  1  0  0  0  0
  8 13  2  0  0  0  0
  9 10  2  0  0  0  0
 10 11  1  0  0  0  0
 11 12  2  0  0  0  0
 12 13  1  0  0  0  0
M  END
$MOL


  Mrv2102 11182121282D

 12 11  0  0  0  0            999 V2000
   -3.7934    0.7703    0.0000 C   0  0  0  0  0  0  0  0  0 14  0  0
   -3.0790    1.1828    0.0000 C   0  0  0  0  0  0  0  0  0 15  0  0
```

```
   -2.3645    0.7703    0.0000 C   0  0  0  0  0  0  0  0  0 16  0  0
   -3.7934   -0.0547    0.0000 C   0  0  0  0  0  0  0  0  0 17  0  0
   -4.5078   -0.4672    0.0000 O   0  0  0  0  0  0  0  0  0 18  0  0
   -3.0789   -0.4671    0.0000 O   0  0  0  0  0  0  0  0  0 19  0  0
   -1.6500    1.1828    0.0000 O   0  0  0  0  0  0  0  0  0 20  0  0
   -2.3645   -0.0547    0.0000 O   0  0  0  0  0  0  0  0  0 21  0  0
   -3.0788   -1.2922    0.0000 C   0  0  0  0  0  0  0  0  0 22  0  0
   -1.6500   -0.4672    0.0000 C   0  0  0  0  0  0  0  0  0 23  0  0
   -2.3644   -1.7046    0.0000 C   0  0  0  0  0  0  0  0  0 24  0  0
   -1.6500   -1.2922    0.0000 C   0  0  0  0  0  0  0  0  0 25  0  0
  1  2  2  0  0  0  0
  1  4  1  0  0  0  0
  2  3  1  0  0  0  0
  3  7  2  0  0  0  0
  3  8  1  0  0  0  0
  4  5  2  0  0  0  0
  4  6  1  0  0  0  0
  6  9  1  0  0  0  0
  8 10  1  0  0  0  0
  9 11  1  0  0  0  0
 10 12  1  0  0  0  0
M  END
$MOL


  Mrv2102 11182121282D


 25 26  0  0  0  0            999 V2000
    5.1328    0.9532    0.0000 C   0  0  0  0  0  0  0  0  0  5  0  0
```

```
   5.8002    0.4683    0.0000 N   0  0  0  0  0  0  0  0  0  6  0  0
   5.5453   -0.3163    0.0000 C   0  0  0  0  0  0  0  0  0  7  0  0
   4.7203   -0.3163    0.0000 C   0  0  0  0  0  0  0  0  0 14  0  0
   4.4654    0.4683    0.0000 C   0  0  0  0  0  0  0  0  0 15  0  0
   5.1328    1.7782    0.0000 C   0  0  0  0  0  0  0  0  0  3  0  0
   3.6807    0.7232    0.0000 C   0  0  0  0  0  0  0  0  0 16  0  0
   4.2354   -0.9838    0.0000 C   0  0  0  0  0  0  0  0  0 17  0  0
   6.0302   -0.9838    0.0000 C   0  0  0  0  0  0  0  0  0  8  0  0
   6.8507   -0.8975    0.0000 C   0  0  0  0  0  0  0  0  0  9  0  0
   7.3356   -1.5650    0.0000 C   0  0  0  0  0  0  0  0  0 10  0  0
   7.0001   -2.3187    0.0000 C   0  0  0  0  0  0  0  0  0 11  0  0
   6.1796   -2.4049    0.0000 C   0  0  0  0  0  0  0  0  0 12  0  0
   5.6947   -1.7375    0.0000 C   0  0  0  0  0  0  0  0  0 13  0  0
   3.4149   -0.8975    0.0000 O   0  0  0  0  0  0  0  0  0 18  0  0
   4.5709   -1.7375    0.0000 O   0  0  0  0  0  0  0  0  0 19  0  0
   4.0860   -2.4049    0.0000 C   0  0  0  0  0  0  0  0  0 22  0  0
   3.2655   -2.3187    0.0000 C   0  0  0  0  0  0  0  0  0 24  0  0
   3.5092    1.5302    0.0000 O   0  0  0  0  0  0  0  0  0 20  0  0
   3.0676    0.1712    0.0000 O   0  0  0  0  0  0  0  0  0 21  0  0
   2.2830    0.4261    0.0000 C   0  0  0  0  0  0  0  0  0 23  0  0
   1.6699   -0.1259    0.0000 C   0  0  0  0  0  0  0  0  0 25  0  0
   5.8473    2.1907    0.0000 O   0  0  0  0  0  0  0  0  0  4  0  0
   4.4183    2.1907    0.0000 O   0  0  0  0  0  0  0  0  0  2  0  0
   4.4183    3.0157    0.0000 C   0  0  0  0  0  0  0  0  0  1  0  0
1  2  1  0  0  0  0
2  3  1  0  0  0  0
3  4  1  0  0  0  0
4  5  1  0  0  0  0
```

```
  1  5  1  0  0  0  0
  1  6  1  0  0  0  0
  5  7  1  0  0  0  0
  4  8  1  0  0  0  0
  3  9  1  0  0  0  0
 10 11  2  0  0  0  0
 11 12  1  0  0  0  0
 12 13  2  0  0  0  0
 13 14  1  0  0  0  0
  9 10  1  0  0  0  0
  9 14  2  0  0  0  0
  8 15  2  0  0  0  0
  8 16  1  0  0  0  0
 16 17  1  0  0  0  0
 17 18  1  0  0  0  0
  7 19  2  0  0  0  0
  7 20  1  0  0  0  0
 20 21  1  0  0  0  0
 21 22  1  0  0  0  0
  6 23  2  0  0  0  0
  6 24  1  0  0  0  0
 24 25  1  0  0  0  0
M  END
''')
>>> IPythonConsole.molSize = (600,250)
>>> IPythonConsole.highlightByReactant = True
>>> rxn1
```



We can ask the reaction to tell us which atoms in the reactants are modified in the reaction:

```
>>> rxn1.Initialize()
>>> rxn1.GetReactingAtoms()
((4, 5, 6), (0, 1))
```

The information about which atoms react is enough to figure out which bonds change, but we have to do some additional work for this:

```
    >>> from collections import namedtuple
    >>> AtomInfo = namedtuple('AtomInfo',
                              ('mapnum','reactant','reactantAtom',
                               'product','productAtom'))
    def map_reacting_atoms_to_products(rxn,reactingAtoms):
        ''' figures out which atoms in the products each
            mapped atom in the reactants maps to '''
        res = []
        for ridx,reacting in enumerate(reactingAtoms):
            reactant = rxn.GetReactantTemplate(ridx)
            for raidx in reacting:
                mapnum = reactant.GetAtomWithIdx(raidx).GetAtomMapNum()
                foundit=False
                for pidx,product in enumerate(rxn.GetProducts()):
                    for paidx,patom in enumerate(product.GetAtoms()):
```

```
                    if patom.GetAtomMapNum()==mapnum:
                        res.append(AtomInfo(mapnum,ridx,raidx,pidx,paidx))
                        foundit = True
                        break
                if foundit:
                    break
    return res
def get_mapped_neighbors(atom):
    ''' test all mapped neighbors of a mapped atom'''
    res = {}
    amap = atom.GetAtomMapNum()
    if not amap:
        return res
    for nbr in atom.GetNeighbors():
        nmap = nbr.GetAtomMapNum()
        if nmap:
            if amap>nmap:
                res[(nmap,amap)] = (atom.GetIdx(),nbr.GetIdx())
            else:
                res[(amap,nmap)] = (nbr.GetIdx(),atom.GetIdx())
    return res


>>> BondInfo = namedtuple('BondInfo',
                          ('product','productAtoms','productBond','status'))
def find_modifications_in_products(rxn):
''' returns a 2-tuple with the modified atoms and bonds from the reaction '''
    reactingAtoms = rxn.GetReactingAtoms()
    amap = map_reacting_atoms_to_products(rxn,reactingAtoms)
```

```
    res = []
    seen = set()
    # this is all driven from the list of reacting atoms:
    for _,ridx,raidx,pidx,paidx in amap:
        reactant = rxn.GetReactantTemplate(ridx)
        ratom = reactant.GetAtomWithIdx(raidx)
        product = rxn.GetProductTemplate(pidx)
        patom = product.GetAtomWithIdx(paidx)

        rnbrs = get_mapped_neighbors(ratom)
        pnbrs = get_mapped_neighbors(patom)
        for tpl in pnbrs:
            pbond = product.GetBondBetweenAtoms(*pnbrs[tpl])
            if (pidx,pbond.GetIdx()) in seen:
                continue
            seen.add((pidx,pbond.GetIdx()))
            if not tpl in rnbrs:
                # new bond in product
                res.append(BondInfo(pidx,pnbrs[tpl],pbond.GetIdx(),'New'))
            else:
                # present in both reactants and products,
                check to see if it changed
                rbond = reactant.GetBondBetweenAtoms(*rnbrs[tpl])
                if rbond.GetBondType()!=pbond.GetBondType():
                    res.append(BondInfo(pidx,pnbrs[tpl],pbond.GetIdx(),'Changed'))
    return amap,res
```

Let's look at what that function returns for our reaction:

```
>>> atms,bnds = find_modifications_in_products(rxn1)
```

```
>>> print(atms)
>>> print(bnds)
[AtomInfo(mapnum=5, reactant=0, reactantAtom=4, product=0, productAtom=0),
 AtomInfo(mapnum=6, reactant=0, reactantAtom=5, product=0, productAtom=1),
 AtomInfo(mapnum=7, reactant=0, reactantAtom=6, product=0, productAtom=2),
 AtomInfo(mapnum=14, reactant=1, reactantAtom=0, product=0, productAtom=3),
 AtomInfo(mapnum=15, reactant=1, reactantAtom=1, product=0, productAtom=4)]
[BondInfo(product=0, productAtoms=(4, 0), productBond=4, status='New'),
 BondInfo(product=0, productAtoms=(2, 1), productBond=1, status='Changed'),
 BondInfo(product=0, productAtoms=(3, 2), productBond=2, status='New'),
 BondInfo(product=0, productAtoms=(4, 3), productBond=3, status='Changed')]
```

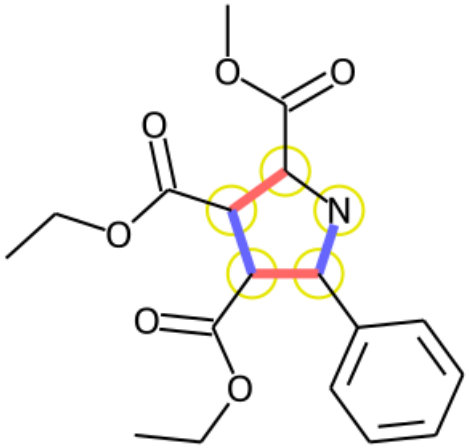
Finally, define the funciton which we'll use to draw the product molecule with highlights shown for bonds and atoms involved in the reaction:

```
>>> from IPython.display import Image
def draw_product_with_modified_bonds(rxn,atms,bnds,
                                     productIdx=None,showAtomMaps=False):
    if productIdx is None:
        pcnts = [x.GetNumAtoms() for x in rxn.GetProducts()]
        largestProduct = list(sorted(zip(pcnts,range(len(pcnts))),
                                     reverse=True))[0][1]
        productIdx = largestProduct
    d2d = Draw.rdMolDraw2D.MolDraw2DCairo(350,300)
    pmol = Chem.Mol(rxn.GetProductTemplate(productIdx))
    Chem.SanitizeMol(pmol)
    if not showAtomMaps:
        for atom in pmol.GetAtoms():
            atom.SetAtomMapNum(0)
    bonds_to_highlight=[]
```

```
highlight_bond_colors={}
atoms_seen = set()
for binfo in bnds:
    if binfo.product==productIdx and binfo.status=='New':
        bonds_to_highlight.append(binfo.productBond)
        atoms_seen.update(binfo.productAtoms)
        highlight_bond_colors[binfo.productBond] = (1,.4,.4)
    if binfo.product==productIdx and binfo.status=='Changed':
        bonds_to_highlight.append(binfo.productBond)
        atoms_seen.update(binfo.productAtoms)
        highlight_bond_colors[binfo.productBond] = (.4,.4,1)
atoms_to_highlight=set()
for ainfo in atms:
    if ainfo.product != productIdx or ainfo.productAtom in atoms_seen:
        continue
    atoms_to_highlight.add(ainfo.productAtom)

d2d.drawOptions().useBWAtomPalette()
d2d.drawOptions().continuousHighlight=False
d2d.drawOptions().highlightBondWidthMultiplier = 24
d2d.drawOptions().setHighlightColour((.9,.9,0))
d2d.drawOptions().fillHighlights=False
atoms_to_highlight.update(atoms_seen)
d2d.DrawMolecule(pmol,highlightAtoms=atoms_to_highlight,
                 highlightBonds=bonds_to_highlight,
                 highlightBondColors=highlight_bond_colors)
d2d.FinishDrawing()
return d2d.GetDrawingText()
```

Now we can draw the highlighted product molecule:

```
>>> Image(draw_product_with_modified_bonds(rxn1,atms,bnds))
```

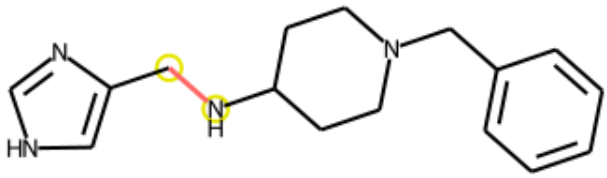


Let's look at some other reactions. I will use the SI data from

```
>>> import pandas as pd
>>> df = pd.read_csv('../data/reaction_data_ci6b00564/dataSetB.csv')
>>> df.head()
```

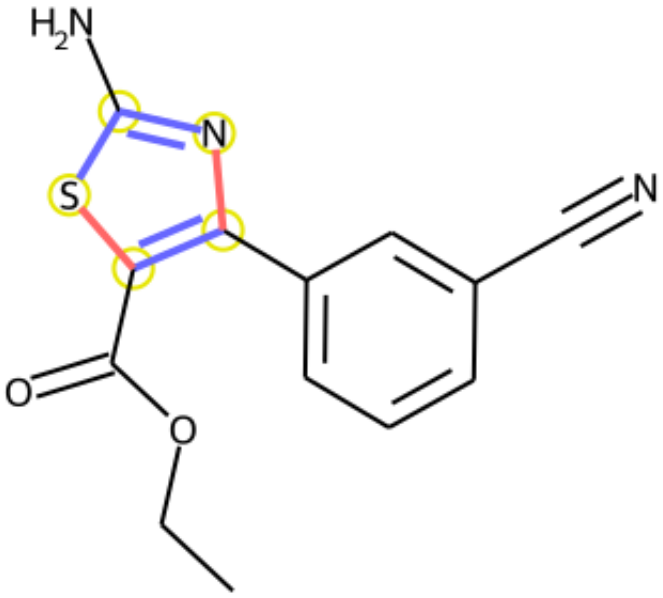
	rxn_Class	patentID	rxnSmiles_Mapping_NameRxn	reactantSet_
0	6	US05849732	C.CCCCCC.CO.O=C(OCc1ccccc1)[NH:1][CH2:2][CH2:3...	set([3, 4])
1	2	US20120114765A1	O[C:1](=[O:2])[c:3]1[cH:4][c:5]([N+:6](=[O:7])...	set([0, 1])
2	1	US08003648B2	Cl.O=[CH:1][c:2]1[cH:3][cH:4][c:5](-[c:6]2[n:7...	set([1, 3])
3	1	US09045475B2	CC(=O)O[BH-](OC(C)=O)OC(C)=O.ClCCl.O=[C:1]([CH...	set([2, 3])
4	2	US08188098B2	CCN(C(C)C)C(C)C.ClCCl.Cl[C:1](=[O:2])[O:3][CH:...	set([2, 5])

```
>>> rxnclass = 1
>>> class_smis = df[df['rxn_Class']==rxnclass].rxnSmiles_Mapping_NameRxn.to_list()
>>> rxn = rdChemReactions.ReactionFromSmarts(class_smis[3],useSmiles=True)
>>> rxn.Initialize()
>>> atms,bnds = find_modifications_in_products(rxn)
>>> print(atms)
>>> print(bnds)
>>> Image(draw_product_with_modified_bonds(rxn,atms,bnds))
[AtomInfo(mapnum=1, reactant=3, reactantAtom=1, product=0, productAtom=0),
 AtomInfo(mapnum=7, reactant=4, reactantAtom=0, product=0, productAtom=6)]
[BondInfo(product=0, productAtoms=(6, 0), productBond=5, status='New')]
```



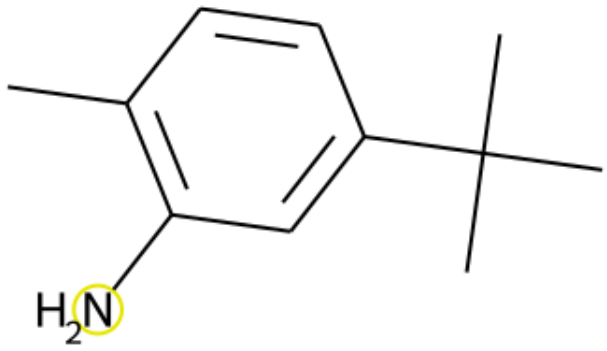

```
>>> rxnclass = 4
>>> class_smis = df[df['rxn_Class']==rxnclass].rxnSmiles_Mapping_NameRxn.to_list()
>>> rxn = rdChemReactions.ReactionFromSmarts(class_smis[1],useSmiles=True)
>>> rxn.Initialize()
>>> atms,bnds = find_modifications_in_products(rxn)
>>> print(atms)
>>> print(bnds)
>>> Image(draw_product_with_modified_bonds(rxn,atms,bnds))
[AtomInfo(mapnum=1, reactant=1, reactantAtom=1, product=0, productAtom=0),
 AtomInfo(mapnum=2, reactant=1, reactantAtom=2, product=0, productAtom=9),
 AtomInfo(mapnum=16, reactant=2, reactantAtom=0, product=0, productAtom=18),
 AtomInfo(mapnum=17, reactant=2, reactantAtom=1, product=0, productAtom=16),
 AtomInfo(mapnum=19, reactant=2, reactantAtom=3, product=0, productAtom=15)]
```

```
[BondInfo(product=0, productAtoms=(9, 0), productBond=8, status='Changed'),
 BondInfo(product=0, productAtoms=(18, 0), productBond=18, status='New'),
 BondInfo(product=0, productAtoms=(15, 9), productBond=14, status='New'),
 BondInfo(product=0, productAtoms=(16, 18), productBond=17, status='Changed'),
 BondInfo(product=0, productAtoms=(15, 16), productBond=15, status='Changed')]
```



Look at an example where there are no changed bonds in the products but where there is a changed atom:

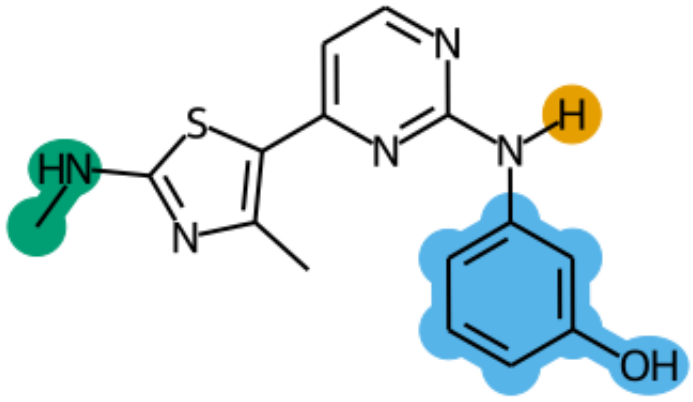
```
>>> rxnclass = 7
>>> class_smis = df[df['rxn_Class']==rxnclass].rxnSmiles_Mapping_NameRxn.to_list()
>>> rxn = rdChemReactions.ReactionFromSmarts(class_smis[1],useSmiles=True)
>>> rxn.Initialize()
>>> atms,bnds = find_modifications_in_products(rxn)
>>> print(atms)
>>> print(bnds)
>>> Image(draw_product_with_modified_bonds(rxn,atms,bnds))
[AtomInfo(mapnum=1, reactant=1, reactantAtom=1, product=0, productAtom=0)]
[]
```


H2N

R-Group Decomposition and Highlighting

This tutorial demonstrates how to perform R-group decomposition (RGD) on a set of molecules that share a common scaffold, generating coordinates for those molecules that are aligned to the scaffold, and generating images of the molecules where the R groups are colored to make them easy to pick out.

The final images we create will look like this:




```
>>> from rdkit import Chem
>>> from rdkit.Chem import Draw
>>> from rdkit.Chem.Draw import IPythonConsole
>>> IPythonConsole.molSize=(450,350)
>>> from rdkit.Chem import rdRGroupDecomposition
>>> from rdkit.Chem import rdqueries
>>> from rdkit.Chem import rdDepictor
>>> from rdkit.Chem.Draw import rdMolDraw2D
>>> from rdkit import Geometry
>>> rdDepictor.SetPreferCoordGen(True)
>>> import pandas as pd
>>> from IPython.display import SVG,Image
>>> from ipywidgets import interact
>>> import rdkit
```

```
>>> print(rdkit.__version__)
2021.03.4
```

Start by reading in the data we will use. This is a collection of ChEMBL compounds with Ki data measured for CDK2. The dataset includes compounds from a number of different documents and, since these are medchem papers, many of the documents contain groups of compounds that share a common scaffold.

```
>>> df = pd.read_csv('../data/cdk2_rgd_dataset.csv')
>>> df.head()
```

	assay_id	doc_id	description	assay_organism	assay_chembl_id	aidx
0	50641	17759	Inhibitory activity against human CDK2 (Cyclin...	NaN	CHEMBL658107	CLD0
1	50641	17759	Inhibitory activity against human CDK2 (Cyclin...	NaN	CHEMBL658107	CLD0
2	50641	17759	Inhibitory activity against human CDK2 (Cyclin...	NaN	CHEMBL658107	CLD0
3	50641	17759	Inhibitory activity against human CDK2 (Cyclin...	NaN	CHEMBL658107	CLD0
4	50641	17759	Inhibitory activity against human CDK2 (Cyclin...	NaN	CHEMBL658107	CLD0

5 rows × 28 columns

We pick a group of compounds by selecting all the rows with a given assay ID:

```
>>> df_doc1 = df[df.assay_chembl_id=='CHEMBL827377']
>>> print(len(df_doc1))
>>> df_doc1.head()
91
```

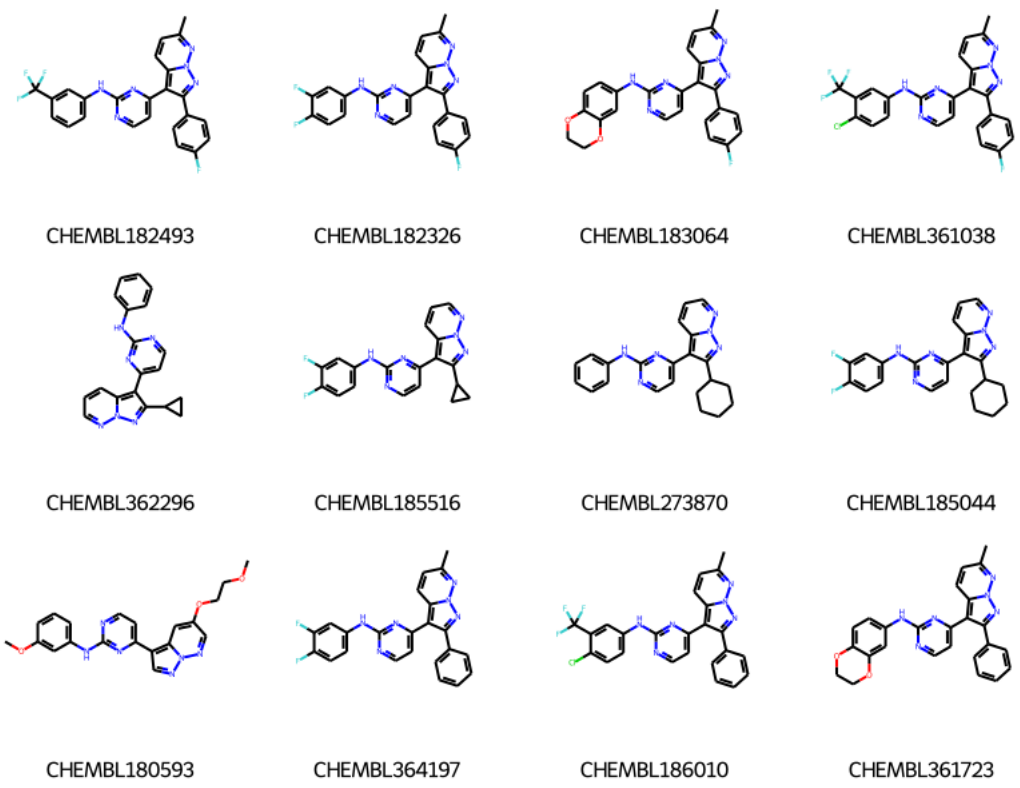
	assay_id	doc_id	description	assay_organism	assay_chembl_id	aidx
47	302524	21080	Binding affinity for human cyclin-dependent ki...	Homo sapiens	CHEMBL827377	CLD0
48	302524	21080	Binding affinity for human cyclin-dependent ki...	Homo sapiens	CHEMBL827377	CLD0
49	302524	21080	Binding affinity for human cyclin-dependent ki...	Homo sapiens	CHEMBL827377	CLD0
50	302524	21080	Binding affinity for human cyclin-dependent ki...	Homo sapiens	CHEMBL827377	CLD0
51	302524	21080	Binding affinity for human cyclin-dependent ki...	Homo sapiens	CHEMBL827377	CLD0

5 rows × 28 columns

Look at some of the compounds:

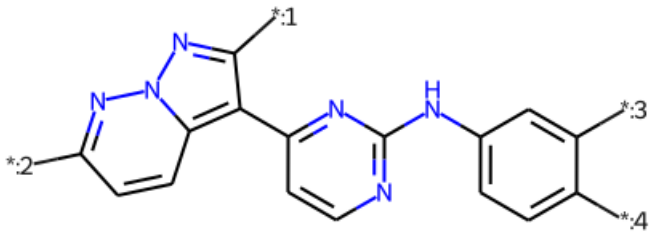
```
>>> rdDepictor.SetPreferCoordGen(True)
>>> smis = df_doc1['canonical_smiles']
>>> cids = list(df_doc1.compound_chembl_id)
>>> ms = [Chem.MolFromSmiles(x) for x in smis]
>>> for m in ms:
```

```
...     rdDepictor.Compute2DCoords(m)
>>> Draw.MolsToGridImage(ms[:12],legends=cids,molsPerRow=4)
```



Define a core. I'm doing this manually and am only specifically labeling four of the seven R-groups in this set of molecules. The others will be labelled automatically by the RGD code.

```
>>> core = Chem.MolFromSmiles('c1cc(-c2c([*:1])nn3nc([*:2])ccc23) \
                              nc(N(c2ccc([*:4])c([*:3])c2))n1')
>>> rdDepictor.SetPreferCoordGen(True)
>>> rdDepictor.Compute2DCoords(core)
>>> core
```



Some pre-processing work we need to do: - convert the dummy atoms in the scaffold into query atoms that match anything - add hydrogens to the molecules - select only the subset of molecules which match the core - set a property on each atom which is used to track its original index (we use this later in the RGD analysis)

```
>>> ps = Chem.AdjustQueryParameters.NoAdjustments()
>>> ps.makeDummiesQueries=True
>>> qcore = Chem.AdjustQueryProperties(core,ps)
>>> mhs = [Chem.AddHs(x,addCoords=True) for x in ms]
>>> mms = [x for x in mhs if x.HasSubstructMatch(qcore)]
>>> for m in mms:
...     for atom in m.GetAtoms():
...         atom.SetIntProp("SourceAtomIdx",atom.GetIdx())
>>> print(len(mhs),len(mms))
91 91
```

Now do the actual RGD:

```
>>> rdkit.RDLogger.DisableLog('rdApp.warning')
>>> groups,_ = rdRGroupDecomposition.RGroupDecompose([qcore],mms,
                                        asSmiles=False,asRows=True)
```

This is the function that actually does the work of generating aligned coordinates and creating the image with highlighted R groups:

```
>>> from collections import defaultdict


def highlight_rgroups(mol,row,core,width=350,height=200,
                      fillRings=True,legend="",
                      sourceIdxProperty="SourceAtomIdx",
                      lbls=('R1','R2','R3','R4')):
    # copy the molecule and core
    mol = Chem.Mol(mol)
    core = Chem.Mol(core)

    # ----------------------------------------
    # include the atom map numbers in the substructure search in order to
    # try to ensure a good alignment of the molecule to symmetric cores
    for at in core.GetAtoms():
        if at.GetAtomMapNum():
            at.ExpandQuery(rdqueries.IsotopeEqualsQueryAtom(200+at.GetAtomMapNum()))

    for lbl in row:
        if lbl=='Core':
            continue
        rg = row[lbl]
        for at in rg.GetAtoms():
            if not at.GetAtomicNum() and at.GetAtomMapNum() and \
            at.HasProp('dummyLabel') and at.GetProp('dummyLabel')==lbl:
                # attachment point. the atoms connected to this
                # should be from the molecule
```

```
            for nbr in at.GetNeighbors():
                if nbr.HasProp(sourceIdxProperty):
                    mAt = mol.GetAtomWithIdx(nbr.GetIntProp(sourceIdxProperty))
                    if mAt.GetIsotope():
                        mAt.SetIntProp('_OrigIsotope',mAt.GetIsotope())
                    mAt.SetIsotope(200+at.GetAtomMapNum())
# remove unmapped hs so that they don't mess up the depiction
rhps = Chem.RemoveHsParameters()
rhps.removeMapped = False
tmol = Chem.RemoveHs(mol,rhps)
rdDepictor.GenerateDepictionMatching2DStructure(tmol,core)

oldNewAtomMap={}
# reset the original isotope values and account for the fact that
# removing the Hs changed atom indices
for i,at in enumerate(tmol.GetAtoms()):
    if at.HasProp(sourceIdxProperty):
        oldNewAtomMap[at.GetIntProp(sourceIdxProperty)] = i
        if at.HasProp("_OrigIsotope"):
            at.SetIsotope(at.GetIntProp("_OrigIsotope"))
            at.ClearProp("_OrigIsotope")
        else:
            at.SetIsotope(0)

# ------------------
#  set up our colormap
#   the three choices here are all "colorblind" colormaps
```

```
# "Tol" colormap from https://davidmathlogic.com/colorblind
colors = [(51,34,136),(17,119,51),(68,170,153),(136,204,238),
          (221,204,119),(204,102,119),(170,68,153),(136,34,85)]
# "IBM" colormap from https://davidmathlogic.com/colorblind
colors = [(100,143,255),(120,94,240),(220,38,127),(254,97,0),
          (255,176,0)]
# Okabe_Ito colormap from https://jfly.uni-koeln.de/color/
colors = [(230,159,0),(86,180,233),(0,158,115),(240,228,66),
          (0,114,178),(213,94,0),(204,121,167)]
for i,x in enumerate(colors):
    colors[i] = tuple(y/255 for y in x)

#----------------------
# Identify and store which atoms, bonds, and rings we'll be highlighting
highlightatoms = defaultdict(list)
highlightbonds = defaultdict(list)
atomrads = {}
widthmults = {}

rings = []
for i,lbl in enumerate(lbls):
    color = colors[i%len(colors)]
    rquery = row[lbl]
    Chem.GetSSSR(rquery)
    rinfo = rquery.GetRingInfo()
    for at in rquery.GetAtoms():
        if at.HasProp(sourceIdxProperty):
            origIdx = oldNewAtomMap[at.GetIntProp(sourceIdxProperty)]
```

```
                highlightatoms[origIdx].append(color)
                atomrads[origIdx] = 0.4
        if fillRings:
            for aring in rinfo.AtomRings():
                tring = []
                allFound = True
                for aid in aring:
                    at = rquery.GetAtomWithIdx(aid)
                    if not at.HasProp(sourceIdxProperty):
                        allFound = False
                        break
                    tring.append(oldNewAtomMap[at.GetIntProp(sourceIdxProperty)])
                if allFound:
                    rings.append((tring,color))
        for qbnd in rquery.GetBonds():
            batom = qbnd.GetBeginAtom()
            eatom = qbnd.GetEndAtom()
            if batom.HasProp(sourceIdxProperty) and eatom.HasProp(sourceIdxProperty):
                origBnd = tmol.GetBondBetweenAtoms(
                        oldNewAtomMap[batom.GetIntProp(sourceIdxProperty)],
                        oldNewAtomMap[eatom.GetIntProp(sourceIdxProperty)])
                bndIdx = origBnd.GetIdx()
                highlightbonds[bndIdx].append(color)
                widthmults[bndIdx] = 2

d2d = rdMolDraw2D.MolDraw2DCairo(width,height)
dos = d2d.drawOptions()
dos.useBWAtomPalette()
```

```
#----------------------
# if we are filling rings, go ahead and do that first so that we draw
# the molecule on top of the filled rings
if fillRings and rings:
    # a hack to set the molecule scale
    d2d.DrawMoleculeWithHighlights(tmol,legend,dict(highlightatoms),
                                   dict(highlightbonds),
                                   atomrads,widthmults)
    d2d.ClearDrawing()
    conf = tmol.GetConformer()
    for (aring,color) in rings:
        ps = []
        for aidx in aring:
            pos = Geometry.Point2D(conf.GetAtomPosition(aidx))
            ps.append(pos)
        d2d.SetFillPolys(True)
        d2d.SetColour(color)
        d2d.DrawPolygon(ps)
    dos.clearBackground = False

#----------------------
# now draw the molecule, with highlights:
d2d.DrawMoleculeWithHighlights(tmol,legend,dict(highlightatoms),
                               dict(highlightbonds),atomrads,widthmults)
d2d.FinishDrawing()
png = d2d.GetDrawingText()
return png
```

Interactively try that out on all the molecules in our set:

```
>>> @interact(idx=range(0,len(ms)))
    def draw_it(idx=0):
        m = mms[idx]
        row = groups[idx]
        return Image(highlight_rgroups(m,row,qcore,lbls=('R1','R2','R3','R4')))
```

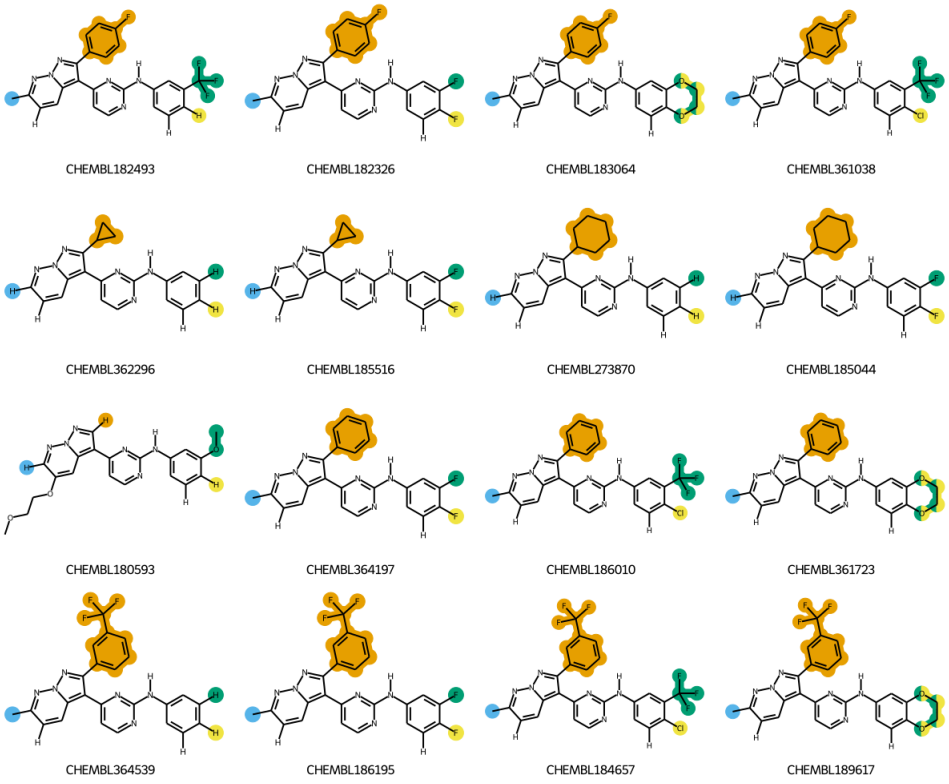
It would be cool to do see multiple molecules at once. Unforunately DrawMolsToGridImage() doesn't support the multiple highlighting we're doing here (we decided that the API for that would just be too complex; this may change in the future if we can figure out a sensible API for it), so we have to manually combine the images. Fortunately the pillow package makes that easy:

```
>>> from PIL import Image as pilImage
>>> from io import BytesIO


def draw_multiple(ms,groups,qcore,lbls,legends=None,nPerRow=4,subImageSize=(250,200)):
    nRows = len(ms)//nPerRow
    if len(ms)%nPerRow:
        nRows+=1
    nCols = nPerRow
    imgSize = (subImageSize[0]*nCols,subImageSize[1]*nRows)
    res = pilImage.new('RGB',imgSize)

    for i,m in enumerate(ms):
        col = i%nPerRow
        row = i//nPerRow
        if legends:
            legend = legends[i]
```

```
        else:
            legend = ''
        png = highlight_rgroups(m,groups[i],qcore,lbls=lbls,legend=legend,
                                width=subImageSize[0],height=subImageSize[1])
        bio = BytesIO(png)
        img = pilImage.open(bio)
        res.paste(img,box=(col*subImageSize[0],row*subImageSize[1]))
    bio = BytesIO()
    res.save(bio,format='PNG')
    return bio.getvalue()
```

Now let's look at the first 16 molecules in the dataset:

```
>>> Image(draw_multiple(mms[:16],groups,qcore,('R1','R2','R3','R4'),
                        legends=cids,subImageSize=(300,250)))
```



Repeat that analysis with the compounds from another document just to make sure we did everything sufficiently generally:

```
>>> df_doc2 = df[df.assay_chembl_id=='CHEMBL658107']
>>> print(len(df_doc2))
>>> df_doc2.head()
```

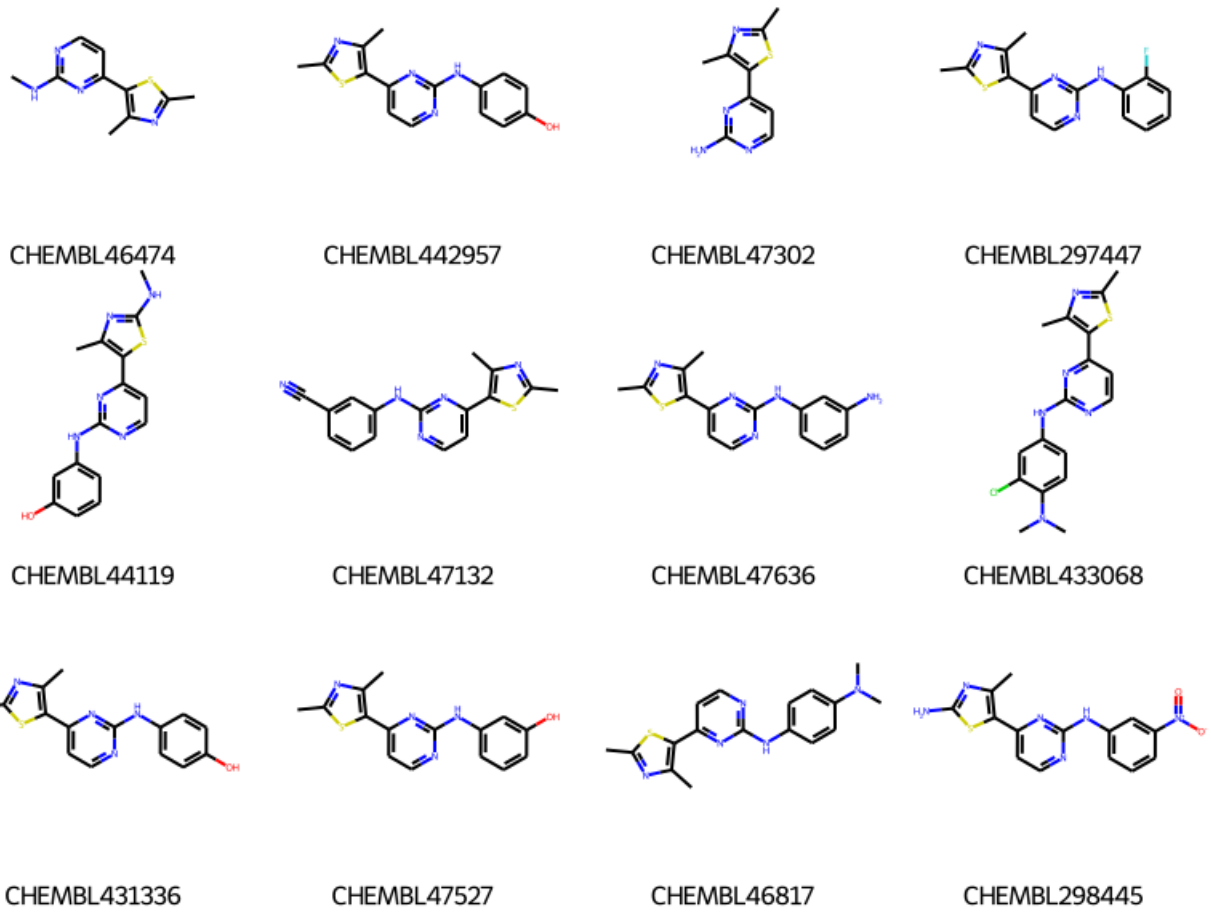
33

	assay_id	doc_id	description	assay_organism	assay_chembl_id	aidx
0	50641	17759	Inhibitory activity against human CDK2 (Cyclin...	NaN	CHEMBL658107	CLD0
1	50641	17759	Inhibitory activity against human CDK2 (Cyclin...	NaN	CHEMBL658107	CLD0
2	50641	17759	Inhibitory activity against human CDK2 (Cyclin...	NaN	CHEMBL658107	CLD0
3	50641	17759	Inhibitory activity against human CDK2 (Cyclin...	NaN	CHEMBL658107	CLD0
4	50641	17759	Inhibitory activity against human CDK2 (Cyclin...	NaN	CHEMBL658107	CLD0

5 rows × 28 columns

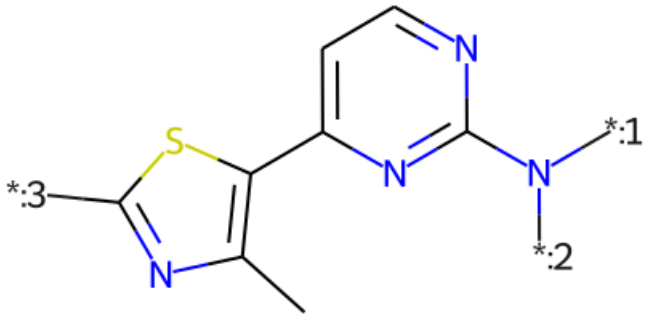
```
>>> smis = df_doc2['canonical_smiles']
>>> cids = list(df_doc2.compound_chembl_id)
>>> ms = [Chem.MolFromSmiles(x) for x in smis]
>>> for m in ms:
```

```
...     rdDepictor.Compute2DCoords(m)
>>> Draw.MolsToGridImage(ms[:12],legends=cids,molsPerRow=4)
```

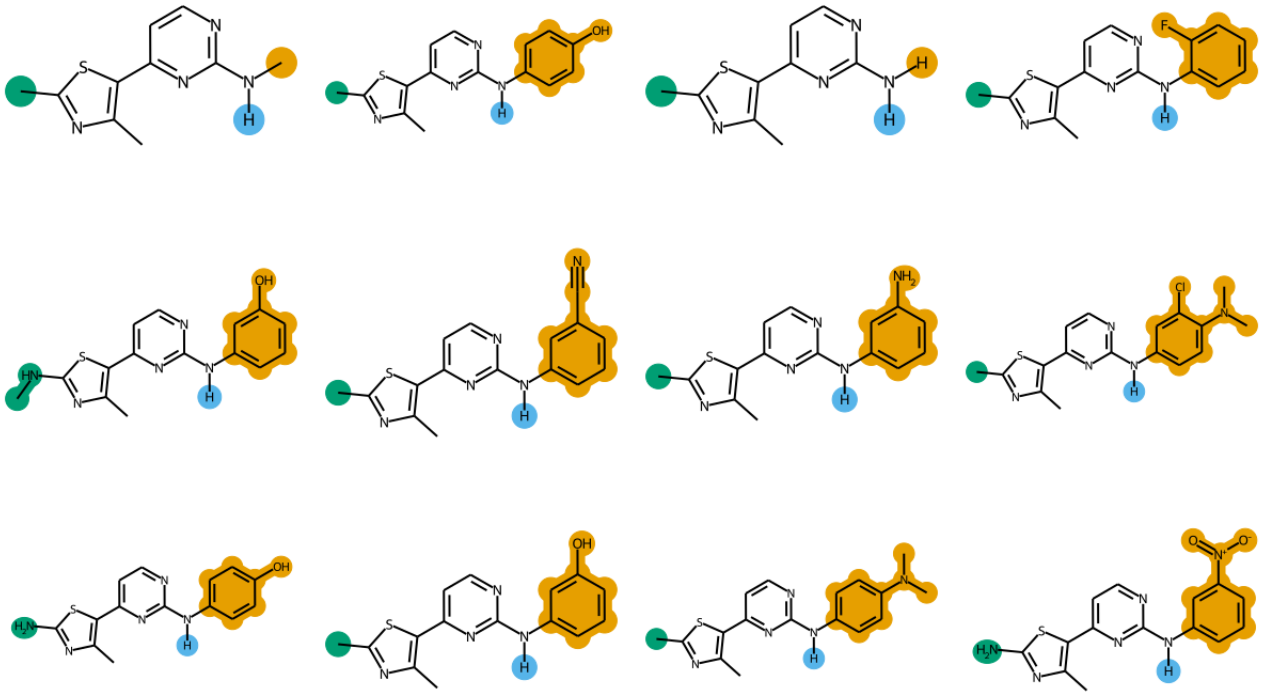



```
>>> core = Chem.MolFromSmiles('Cc1nc([*:3])sc1-c1ccnc(N([*:1])[*:2])n1')
>>> ps = Chem.AdjustQueryParameters.NoAdjustments()
>>> ps.makeDummiesQueries=True
>>> qcore = Chem.AdjustQueryProperties(core,ps)
>>> mhs = [Chem.AddHs(x,addCoords=True) for x in ms]
>>> mms = [x for x in mhs if x.HasSubstructMatch(qcore)]
>>> for m in mms:
...     for atom in m.GetAtoms():
...         atom.SetIntProp("SourceAtomIdx",atom.GetIdx())
>>> print(len(mhs),len(mms))
```

```
>>> rdDepictor.SetPreferCoordGen(True)
>>> rdDepictor.Compute2DCoords(qcore)
>>> qcore
33 33
```




```
>>> rdkit.RDLogger.DisableLog('rdApp.warning')
>>> groups,_ = rdRGroupDecomposition.RGroupDecompose([qcore],mms,
                                        asSmiles=False,asRows=True)
>>> @interact(idx=range(0,len(mms)))
    def draw_it(idx=0):
        m = mms[idx]
        row = groups[idx]
        return Image(highlight_rgroups(m,row,qcore,lbls=('R1','R2','R3')))

>>> Image(draw_multiple(mms[:12],groups,qcore,
                        ('R1','R2','R3'),subImageSize=(300,250)))
```



```
>>> df_doc3 = df[df.assay_chembl_id=='CHEMBL3101313']
>>> print(len(df_doc3))
>>> df_doc3.head()
25
```

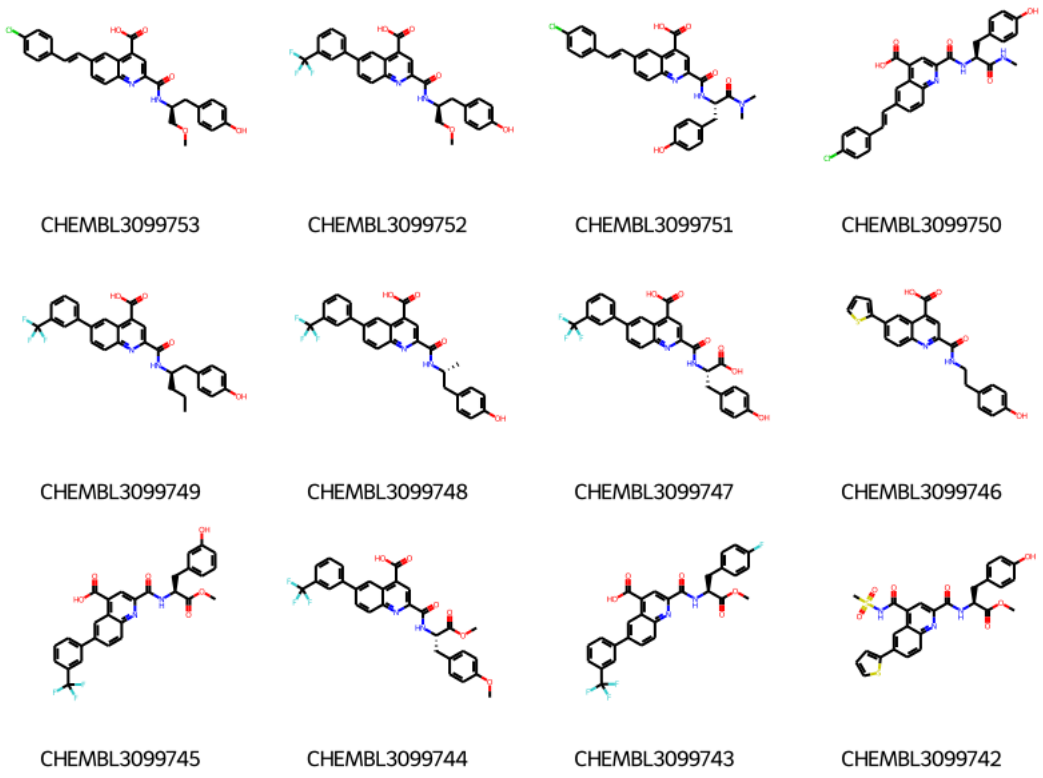
	assay_id	doc_id	description	assay_organism	assay_chembl_id
1129	1281340	76402	Displacement of B-Alexa-Fluor647 from CDK2 (un...	Homo sapiens	CHEMBL3101313
1130	1281340	76402	Displacement of B-Alexa-Fluor647 from CDK2 (un...	Homo sapiens	CHEMBL3101313
1131	1281340	76402	Displacement of B-Alexa-Fluor647 from CDK2 (un...	Homo sapiens	CHEMBL3101313
1132	1281340	76402	Displacement of B-Alexa-Fluor647 from CDK2 (un...	Homo sapiens	CHEMBL3101313
1133	1281340	76402	Displacement of B-Alexa-Fluor647 from CDK2 (un...	Homo sapiens	CHEMBL3101313

5 rows × 28 columns

Finally, do another document, just because it's fun. :-)

```
>>> smis = df_doc3['canonical_smiles']
>>> cids = list(df_doc3.compound_chembl_id)
>>> ms = [Chem.MolFromSmiles(x) for x in smis]
>>> for m in ms:
...     rdDepictor.Compute2DCoords(m)
```

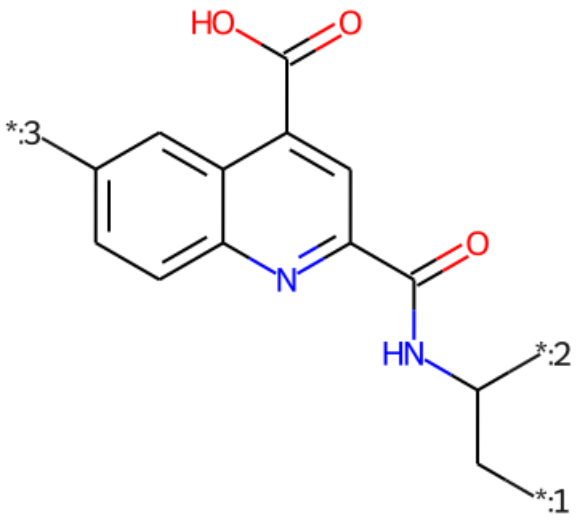
```
>>> Draw.MolsToGridImage(ms[:12],legends=cids,molsPerRow=4)
```



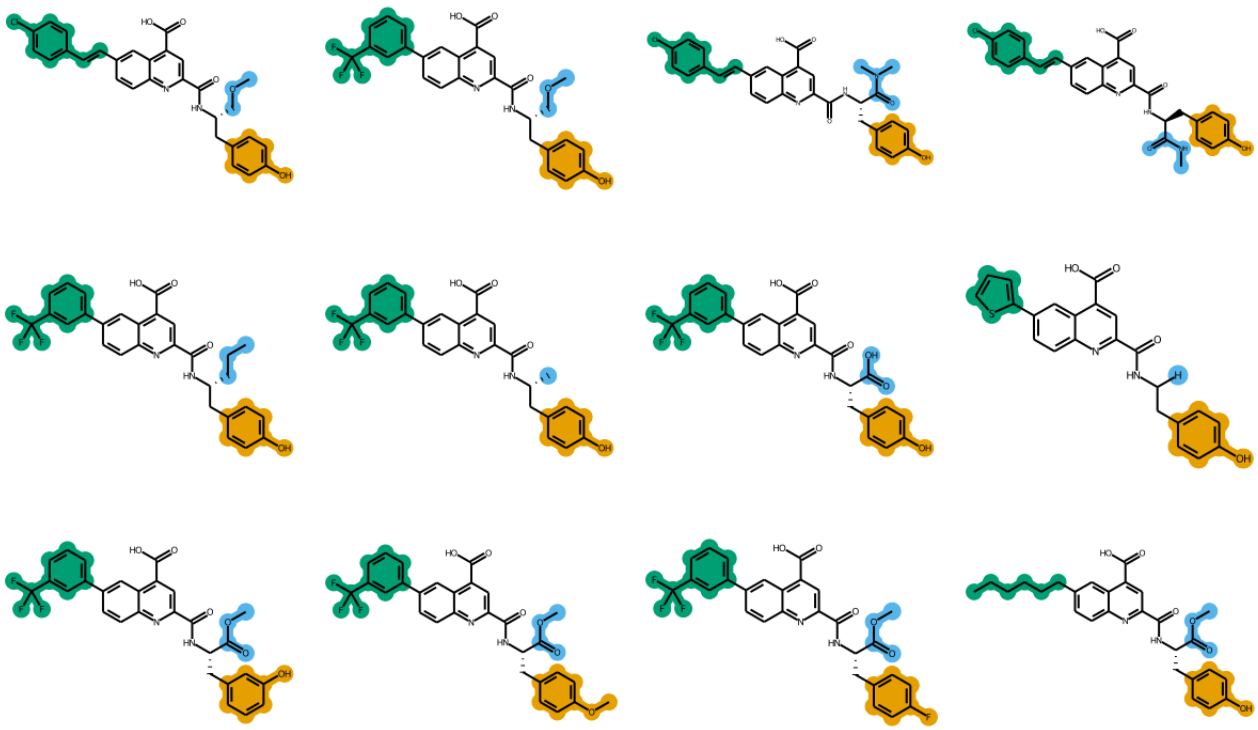

```
>>> smi = 'OC(=O)c1cc(C(=O)NC(C[*:1])[*:2])nc2ccc([*:3])cc12'
>>> core = Chem.MolFromSmiles(smi)
>>> ps = Chem.AdjustQueryParameters.NoAdjustments()
>>> ps.makeDummiesQueries=True
>>> qcore = Chem.AdjustQueryProperties(core,ps)
>>> mhs = [Chem.AddHs(x,addCoords=True) for x in ms]
>>> mms = [x for x in mhs if x.HasSubstructMatch(qcore)]
>>> for m in mms:
...     for atom in m.GetAtoms():
...         atom.SetIntProp("SourceAtomIdx",atom.GetIdx())
>>> print(len(mhs),len(mms))
>>> rdDepictor.SetPreferCoordGen(True)
>>> rdDepictor.Compute2DCoords(qcore)
```

```
>>> qcore
```

```
25 22
```




```
>>> rdkit.RDLogger.DisableLog('rdApp.warning')
>>> groups,_ = rdRGroupDecomposition.RGroupDecompose([qcore],mms,
                                                     asSmiles=False,asRows=True)
>>> @interact(idx=range(0,len(mms)))
    def draw_it(idx=0):
        m = mms[idx]
        row = groups[idx]
        return Image(highlight_rgroups(m,row,qcore,lbls=('R1','R2','R3')))
>>> Image(draw_multiple(mms[:12],groups,qcore,
                        ('R1','R2','R3'),subImageSize=(300,250)))
```



RDKit video lectures

1. Ed Griffen & David Cosgrove: Efficiently delivering better molecule depictions and highlighting - 30 min
2. Eduardo Mayo MScreen: Molecular docking benchmarking made easy - 18 min
3. Amol Thakkar: Browser based exploration of the GDB chemical space using AI planned synthesis - 28 min
4. Hadrien Mary - Datamol: Molecular Manipulation Made Easy - 20 min
5. Rachael Pirie: Can you hear the shape of a drug? - 22 min
6. Paolo Tosco: Interactive rendering of 2D structures with RDKit - 31 min
7. Marco Bertolini: Beyond atoms and bonds: contextual explanability via molecular graphical depictions - 33 min
8. Hao Shen: Extending Rdkit Reactions and Some Use Cases - 30 min
9. Dmytro Radchenko: Fragment-based substructure search in Enamine REAL Space - 28 min

SQL courses content

SQL: A practical Introduction for Querying Databases

Learning outcome

- Analyze data within a database using SQL.
- Create a relational database on Cloud and work with tables.
- Write SQL statements including SELECT, INSERT, UPDATE, and DELETE.
- Build more powerful queries with advanced SQL techniques like views, transactions, stored procedures and joins.

Course content

Getting started with SQL

In this module, you will be introduced to databases. You will create a database instance on the cloud. You will learn some of the basic SQL statements. You will also write and practice basic SQL hands-on on a live database.

1. Video lectures - 19 min:
   (a) Course introduction
   (b) Introduction to databases,

(c) SELECT, COUNT, DISTINCT, LIMIT, INSERT, UPDATE and DELETE statements

2. Reading - 5 minutes
3. Quizzes - 24 minutes
4. Hands-on labs - 75 minutes:
   - Simple SELECT Statements - 20 minutes
   - COUNT, DISTINCT, LIMIT - 35 minutes
   - INSERT, UPDATE, and DELETE - 20 minutes
5. Plugin - 5 minutes: SELECT statement examples

Introduction to Relational Databases and Tables

In this module, you will explore the fundamental concepts behind databases, tables, and the relationships between them. You will then create an instance of a database, discover SQL statements that allow you to create and manipulate tables, and then practice them on your own live database.

1. Video lectures - 22 min:
   (a) Relational Database Concepts
   (b) Types of SQL statements (DDL vs. DML)
   (c) CREATE TABLE Statement
   (d) ALTER, DROP, and Truncate tables
   (e) How to create a Database instance on Cloud•5 minutes
2. Reading - 6 minutes
3. Quizzes - 24 minutes

4. Hands-on labs - 50 minutes:

    - CREATE, ALTER, TRUNCATE, DROP - 20 minutes
    - Create and Load Tables using SQL Scripts - 20 minutes
    - Obtain IBM Cloud Feature Code and Activate Trial Account - 10 minutes

5. Plugin - 90 minutes:

    (a) Examples to ALTER and TRUNCATE tables using MySQL - 15 minutes

    (b) Examples to CREATE and DROP tables - 15 minutes

    (c) (Optional) Db2 Lab: Create Db2 service instance and Get started with the Db2 console - 15 minutes

    (d) (Optional) Db2 Lab: CREATE, ALTER, TRUNCATE, DROP - 15 minutes

    (e) (Optional) Db2 Lab: Create and Load Tables using SQL Scripts - 30 minutes

Intermediate SQL

In this module, you will learn how to use string patterns and ranges to search data and how to sort and group data in result sets. You will also practice composing nested queries and execute select statements to access data from multiple tables.

1. Video lectures - 31 min:

    (a) Using String Patterns and Ranges

    (b) Sorting Result Sets

    (c) Grouping Result Sets

    (d) Built-in Database Functions

    (e) Date and Time Built-in Functions Sub-Queries and Nested Selects

    (f) Working with Multiple Tables•6 minutes

2. Reading - 10 minutes

3. Quizzes - 48 minutes

4. Hands-on labs - 80 minutes:

    - String Patterns, Sorting and Grouping - 20 minutes
    - Built-in functions - 20 minutes
    - Sub-queries and Nested SELECTS - 20 minutes
    - Working with Multiple Tables - 20 minutes

5. Plugin - 100 minutes:

    (a) (Optional) Db2 Lab : String Patterns, Sorting Grouping - 35 minutes

    (b) (Optional) Db2 Lab: Built-in functions - 15 minutes

    (c) (Optional) Db2 Lab: Sub-queries and Nested SELECTs - 20 minutes

    (d) (Optional) Db2 Lab: Working with Multiple Tables - 30 minutes

Working with real-world datasets, final project and exam

1. Video lectures - 10 min:

    (a) Working with Real World Datasets

    (b) Getting Table and Column Details

2. Reading - 3 minutes

3. Quizzes - 90 minutes

4. Hands-on labs - 40 minutes:

    - Working with a real world data-set - 20 minutes
    - Final project - 20 minutes

5. Plugin - 77 minutes:

(a) (Optional) LOADing Data - 15 minutes

(b) (Optional) Db2 Lab: Working with a Real World Dataset using SQL and IBM Cloud Db2 - 45 minutes

(c) Final Project Overview - 2 minutes

(d) (Optional) Db2 Lab: Exploratory Data Analysis with SQL - 15 minutes

Advanced SQL (Honors)

This module covers some advanced SQL techniques that will be useful for Data Engineers. If you are following the Data Engineering track, you must complete this module. Completion of this module is not required for those completing the Data Science or Data Analyst tracks. In this module, you will learn how to build more powerful queries with advanced SQL techniques like views, transactions, stored procedures and joins.

1. Video lectures - 23 min:

   (a) Views

   (b) Stored Procedures

   (c) ACID Transactions

   (d) Join Overview

   (e) Inner Join

   (f) Outer Join

2. Reading - 12 minutes

3. Quizzes - 60 minutes

4. Advanced SQL for Data Engineers (peer reviewed)

5. Hands-on labs - 160 minutes:

   - Using Views - 20 minutes

- Stored Procedures - 20 minutes
- Committing and Rolling back a Transaction - 20 minutes
- Joins - 40 minutes
- Final project: Advanced SQL Techniques - 60 minutes

6. Plugin - 145 minutes:

   (a) Using Views - 10 minutes

   (b) Stored Procedures - 10 minutes

   (c) Committing and Rolling back a Transaction - 10 minutes

   (d) Joins - 55 minutes

   (e) Final project: Advanced SQL For Data Engineers - 60 minutes

Databases and SQL for Data Science with Python

Learning outcome

- Analyze data within a database using SQL and Python.
- Create a relational database and work with multiple tables using DDL commands.
- Construct basic to intermediate level SQL queries using DML commands.
- Compose more powerful queries with advanced SQL techniques like views, transactions, stored procedures, and joins.

Course content

Getting started with SQL

In this module, you will be introduced to databases. You will learn how to use basic SQL statements like SELECT, INSERT, UPDATE and DELETE. You will also get an understanding of how to refine your query results with the WHERE clause as well as using COUNT, LIMIT and DISTINCT.

1. Video lectures - 18 min:
   (a) Course Introduction•3 minutes
   (b) Introduction to Databases
   (c) SELECT Statement
   (d) COUNT, DISTINCT, LIMIT
   (e) INSERT Statement
   (f) UPDATE and DELETE Statements
2. Reading - 8 minutes
3. Quizzes - 20 minutes

4. Hands-on labs - 70 minutes:

    - Simple SELECT Statements - 20 minutes
    - COUNT, DISTINCT, LIMIT - 30 minutes
    - INSERT, UPDATE, and DELETE - 20 minutes

5. Plugin - 23 minutes:

    (a) Helpful Tips for Course Completion - 3 minutes

    (b) SELECT statement examples - 5 minutes

    (c) SQL Cheat Sheet: Basics - SELECT, INSERT, UPDATE, DELETE, COUNT, DISTINCT, LIMIT - 15 minutes

Introduction to Relational Databases and Tables

In this module, you'll learn more about relational database concepts and their importance. This module helps you to understand the process of creating a table in your database on MySQL using the graphical interface and SQL scripts. Further, you will also learn how to alter the entries or delete the entries for any table in the database, or even delete the table itself. •Preview module •2 minutes •3 minutes •4 minutes •5 minutes

1. Video lectures - 22 min:

    (a) Relational Database Concepts

    (b) Types of SQL statements (DDL vs. DML)

    (c) CREATE TABLE Statement

    (d) ALTER, DROP, and Truncate tables

    (e) How to create a Database instance on Cloud

2. Reading - 3 minutes

3. Quizzes - 25 minutes

4. Hands-on labs - 70 minutes:

    - CREATE, ALTER, TRUNCATE, DROP - 20 minutes
    - Create and Load Tables using SQL Scripts - 20 minutes
    - Obtain IBM Cloud Feature Code and Activate Trial Account - 30 minutes

5. Plugin - 90 minutes:

    (a) Examples to ALTER and TRUNCATE tables using MySQL - 5 minutes

    (b) Examples to CREATE and DROP tables - 5 minutes

    (c) SQL Scripts - Uses and Applications - 7 minutes

    (d) SQL Cheat Sheet: CREATE TABLE, ALTER, DROP, TRUNCATE - 5 minutes

    (e)

Intermediate SQL

This module helps you learn how to use string patterns and ranges to search data and how to sort and group data in result sets. You will also practice composing nested queries and execute select statements to access data from multiple tables.

1. Video lectures - 31 min:

    (a) Using String Patterns and Ranges

    (b) Sorting Result Sets

    (c) Grouping Result Sets

    (d) Built-in Database Functions

    (e) Date and Time Built-in Functions

    (f) Sub-Queries and Nested Selects

    (g) Working with Multiple Tables

2. Reading - 12 minutes

3. Quizzes - 50 minutes

4. Hands-on labs - 90 minutes:

   - String Patterns, Sorting and Grouping - 30 minutes
   - Built-in functions - 20 minutes
   - Sub-queries and Nested Selects - 20 minutes
   - Working with Multiple Tables•20 minutes

5. Plugin - 20 minutes:

   (a) SQL Cheat Sheet: Intermediate - LIKE, ORDER BY, GROUP BY - 5 minutes

   (b) SQL Cheat Sheet: FUNCTIONS and Implicit JOIN - 15 minutes

Accassing Databases using Python

In this module you will learn the basic concepts of using Python to connect to databases. In a Jupyter Notebook, you will create tables, load data, query data using SQL magic and SQLite python library. You will also learn how to analyze data using Python.

1. Video lectures - 33 min:

   (a) How to Access Databases Using Python

   (b) Writing code using DB-API

   (c) Accessing Databases with SQL Magic

   (d) Analyzing data with Python

(e) Connecting to a database using ibm$_db API Creating tables, loading data and querying data$

(2) Reading - 3 minutes

3. Quizzes - 25 minutes

4. Hands-on labs - 200 minutes:

- Creating tables, inserting and querying Data - 20 minutes
- Accessing Databases with SQL magic 20 minutes
- Analyzing a Real-World Data Set - 45 minutes
- (Optional) Db2 Lab: Connecting to a database instance - 20 minutes
- (Optional) Db2 Lab: Creating tables, inserting and querying Data - 30 minutes
- (Optional) Db2 Lab: Tutorial, Accessing Databases with SQL magic - 20 minutes
- (Optional) Db2 Lab: Analyzing a real World Data Set - 45 minutes

5. Plugin - 15 minutes:
   (a) SQL Cheat Sheet: Accessing Databases using Python - 15 minutes

Course Assignment

In this module, you will be working with multiple real-world datasets for the city of Chicago. You will be asked questions that will help you understand the data just as you would in the real world. You will be assessed on the correctness of your SQL queries and results.

1. Video lectures - 8 min:
   (a) Working with Real World Datasets
   (b) Getting Table and Column Details
2. Reading - 4 minutes
3. Final exam - 60 minutes
4. Hands-on labs - 50 minutes:
   - Working with a real world data-set - 30 minutes
   - (Optional) Practice Querying Real World Datasets - 45 minutes
   - Final Assignment: Database Querying using SQLite - 30 minutes

Advances SQL for Data Engineers (Honors)

This module covers some advanced SQL techniques that will be useful for Data Engineers. In this module, you will learn how to build more powerful queries with advanced SQL techniques like views, transactions, stored procedures, and joins. If you are following the Data Engineering track, you must complete this module. Completion of this module is not required for those completing the Data Science or Data Analyst tracks.

1. Video lectures - 23 min:
    (a) Views
    (b) Stored Procedures
    (c) ACID Transactions
    (d) Join Overview
    (e) Inner Join
    (f) Outer Joins
2. Reading - 14 minutes
3. Quizzes - 80 minutes
4. Hands-on labs - 200 minutes:
    - Using Views - 20 minutes
    - Stored Procedures - 20 minutes
    - Committing and Rolling Back a Transaction - 20 minutes
    - Joins - 20 minutes
    - Final Project: Advanced SQL Techniques - 60 minutes
5. Plugin - 15 minutes:
    (a) SQL Cheat Sheet: Views, Stored Procedures and Transactions - 15 minutes
    (b) SQL Cheat Sheet: JOIN Statements - 15 minutes

UNIVERSITETET I BERGEN
Institutt for biomedisin

Referanse

Dato
February 2, 2024

Recommendation of individual curriculum for the PhD degree

For PhD student Alberto Rovetta, with supervisor professor Ruth Brenk.

I refer to application from candidate and main supervisor for approval of an individual curriculum of 3 ECTS for the course part of the PhD degree. It is applied for a curriculum consisting of two main parts, within bioinformatics / chemoinformatics, molecule databases and programming.

Three ECTS correspond to two weeks of full-time work. From the description of the two parts of the individual curriculum, the Department of Biomedicine finds that the curriculum fully corresponds to 3 ECTS.

The assessment of the individual curriculum is described in the application, and an external sensor is also identified.

The Department of Biomedicine recommends the approval of the application.

Regards,

Arne Tjølsen
professor
vice chair of education

Attachments: Application, Description of curriculum

Institutt for biomedisin
Telefon 55586300
post@biomed.uib.no

Postadresse
Postboks 7804
5020 Bergen

Besøksadresse
Jonas Lies vei 91
BB-bygget
Bergen

Saksbehandler