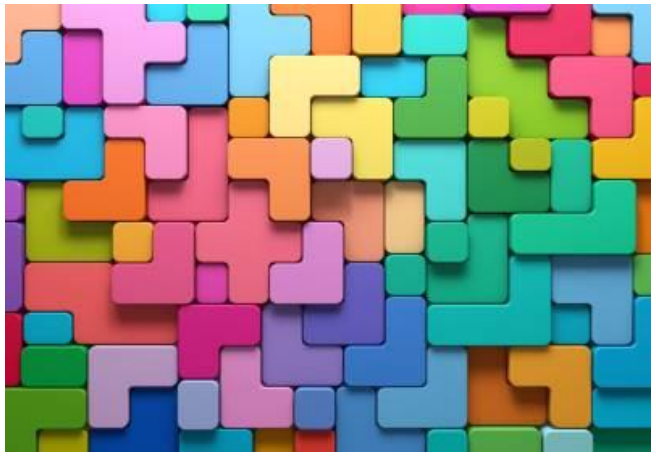
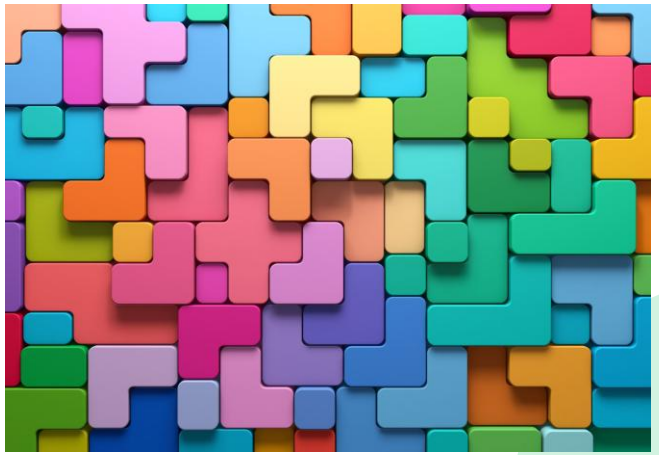
OPPSUMMERING

FORELESNING 25

MANDAG 17/11

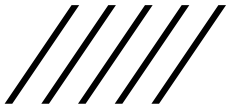




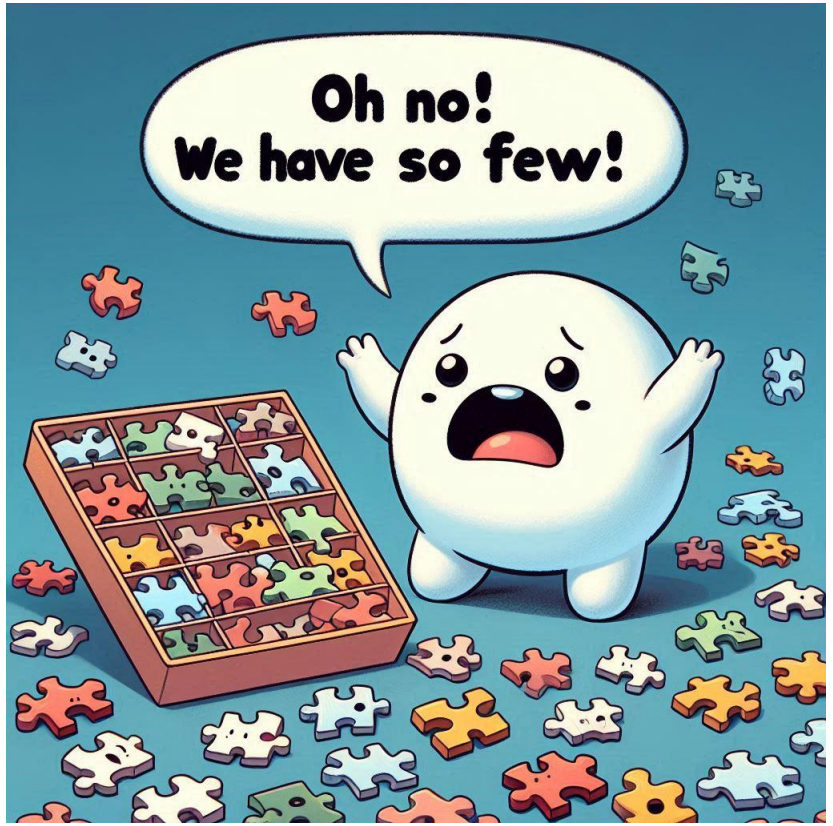
(bilder generert av bing image creator)

Hva er pensum?

- Forelesningene: se timeplanen på emnesiden
  - Inkludert livekoding
- Alle de fire prosjektene



D E T T E E R I K K E
E N
F U L L S T E N D I G
G J E N N O M G A N G !

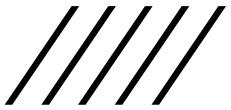


Best Practices for Scientific Computing

1. Skrive programmer så mennesker forstår dem, ikke bare datamaskinen
2. La datamaskinen gjøre det datamaskinen er god til (repetere, automatisere)
3. Gjøre endringer i små steg av gangen (og kunne gå tilbake hvis det oppstår problemer)
4. Unngå å gjenta det du selv (eller andre) har gjort før

Best Practices for Scientific Computing

5. Planlegge for at det oppstår feil (testing, debugging)
6. Gjøre programmet raskt først når det virker som det skal
7. Dokumentere design og formål, ikke tekniske finurligheter (*hva* koden gjør, ikke *hvordan* den gjør det)
8. Samarbeide for å finne feil og holde orden på dem



Læremål

Kort om emnet

Emnet er en fortsettelse av emnet IN1900, og gir en innføring i mer avanserte begreper innenfor programmering og programvareutvikling. Sentrale mål i emnet er å introdusere nyttige verktøy og konsepter for vitenskapelig programmering.

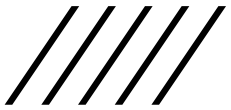
Hva lærer du?

Etter å ha tatt IN1910 vil du:

- Ha kunnskap om og erfaring i mer avansert bruk av programmeringsspråket Python, inkludert arv og objektorientert programmering, samt å bruke Python i kombinasjon med andre programmeringsspråk.
- Ha grunnleggende kunnskap om og erfaring med programmering i C++, inkludert sentrale begreper i objektorientering som abstrakte klasser og virtuelle metoder.
- Ha kjennskap til noen sentrale datastrukturer som f.eks. arrays og lenkede lister, med tilhørende algoritmer
- Kunne generere tilfeldige tall og bruke disse til å kjøre stokastiske simuleringer.
- Kunne enklere former for algoritmeanalyse og optimalisering, som f.eks profilering og parallelisering.
- Kunne bruke verktøy for versjonskontroll og verifikasjon, inkludert enhetstesting og regresjonstesting.

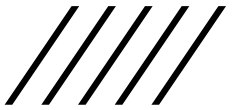
Avansert bruk av Python

- Gjøre programmene lettere å lese og finne fram i
- *Objektorientert programmering* er en måte å tenke på når vi skriver programmer som åpner nye muligheter
- Lar oss lage spesialtilfeller fra mer generelle tilfeller (arv)
- Bryte Python i kombinasjon med andre programmeringsspråk (alle trenger ikke skrive programmer i samme språk for at programmene skal kunne brukes sammen)



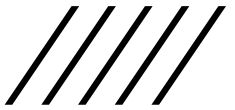
Et nytt språk: C++

- Et litt mer *lavnivå*-språk (nærmere maskinvaren) enn Python
- Derfor mer effektivt og raskere!
- (*numpy* er skrevet i C, som er i nær slekt med C++)
- Programkoden blir ikke tolket "på direkten" men oversatt til maskinkode av et *kompilator*-program
- Å lære ett nytt språk gjør det enklere å lære andre typer språk senere



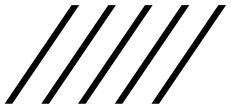
Datastrukturer og algoritmer

- Du har kanskje brukt både *arrays* og *lister* i Python
- Her skal vi gå litt under panseret og se hvordan de fungerer, for eksempel hvordan vi jobber med minnet på datamaskinen
- Dette danner en basis for å kunne lage egne dataobjekter på en god måte når det trengs
- Algoritmer inkluderer bl.a. måter å sortere data på



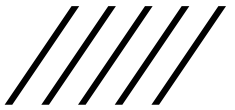
Tilfeldige tall og simuleringer

- Tilfeldighet brukes i mange vitenskapelige sammenhenger
- Sannsynlighet / statistikk
- Kryptering av data i klartekst (ikke pensum for oss)
- Likevektstilstander: La partikler bevege seg tilfeldig og se om de havner i bestemte tilstander



Algoritmeanalyse og optimalisering

- Å kunne analysere hvor rask en algoritme er
  (viktig når vi får enorme mengder data)
- Profilering: finne flaskehalser
  (10% av koden bruker vanligvis 90% av tiden)
- Parallellprogrammering: la datamaskinen gjøre flere ting samtidig (uten å rote det til og gjøre ting i feil rekkefølge)



Versjonskontroll og testing

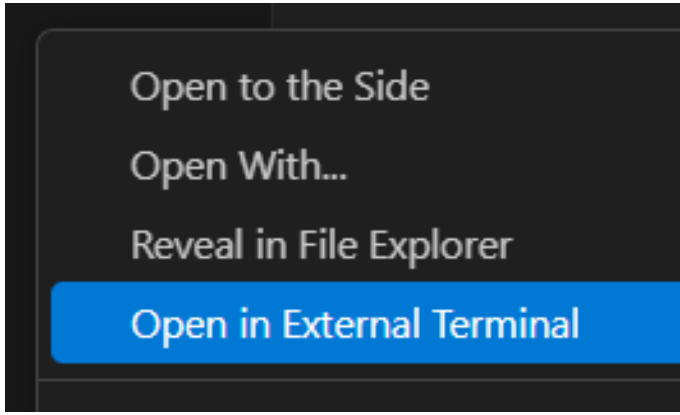
- Bruke GitHub til å holde styr på forskjellige versjoner av programmet (og gå tilbake til noe som fungerte de gangene ting ikke virker eller det bare blir rot)
- Samarbeide uten å ødelegge for hverandre (hver utvikler jobber med sin egen testversjon av moderprogrammet)
- Enhetstesting: automatisk test av at en liten del av et program virker som den skal
- Regresjonstesting: hver gang vi gjør en endring i et program, kjøres tester på nytt for å sikre at ikke en feil har oppstått

Fem ulike måter å forstå på

- Å kunne **forklare** til noen andre hvordan noe virker (dokumentasjon i prosjektene, muntlig eksamen)
- Å kunne **tolke** seg frem til hva som er relevant akkurat her (prosjektene, muntlig eksamen)
- Å kunne **bruke** det du har lært (prosjektene)
- Å kunne se ting fra flere **perspektiv**: utvikler, tester, bruker (prosjektene, muntlig eksamen)
- Å forstå hva du forstår og ikke forstår (**metakognisjon**) (muntlig eksamen)

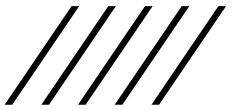
"Hvorfor bruker du ikke play-knappen?"

- Fordi det er en dårlig vane å teste programmet i editoren du bruker til å skrive koden med
- Brukeren av programmet ditt skal bare trenge en terminal og de riktige bibliotekene
- Det hjelper lite å si "ja, men det virker jo på min maskin" hvis brukeren (eller retteren) ikke klarer å kjøre koden din
- Derfor: Bruk terminalen til kjøring og testing!



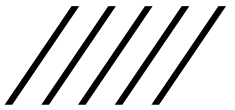
Ikke-muterbare typer

- Objekter av en *ikke-muterbar* type (eller klasse) kan ikke endres
- Eksempler: **bool**, **int**, **float**, **str**, **tuple**
  (en **tuple** er liste som ikke kan endres)



Muterbare typer

- Objekter av en *muterbar* type (eller klasse) kan derimot endres underveis i programmet
- Eksempler: **list**, **dict**, **set**, …



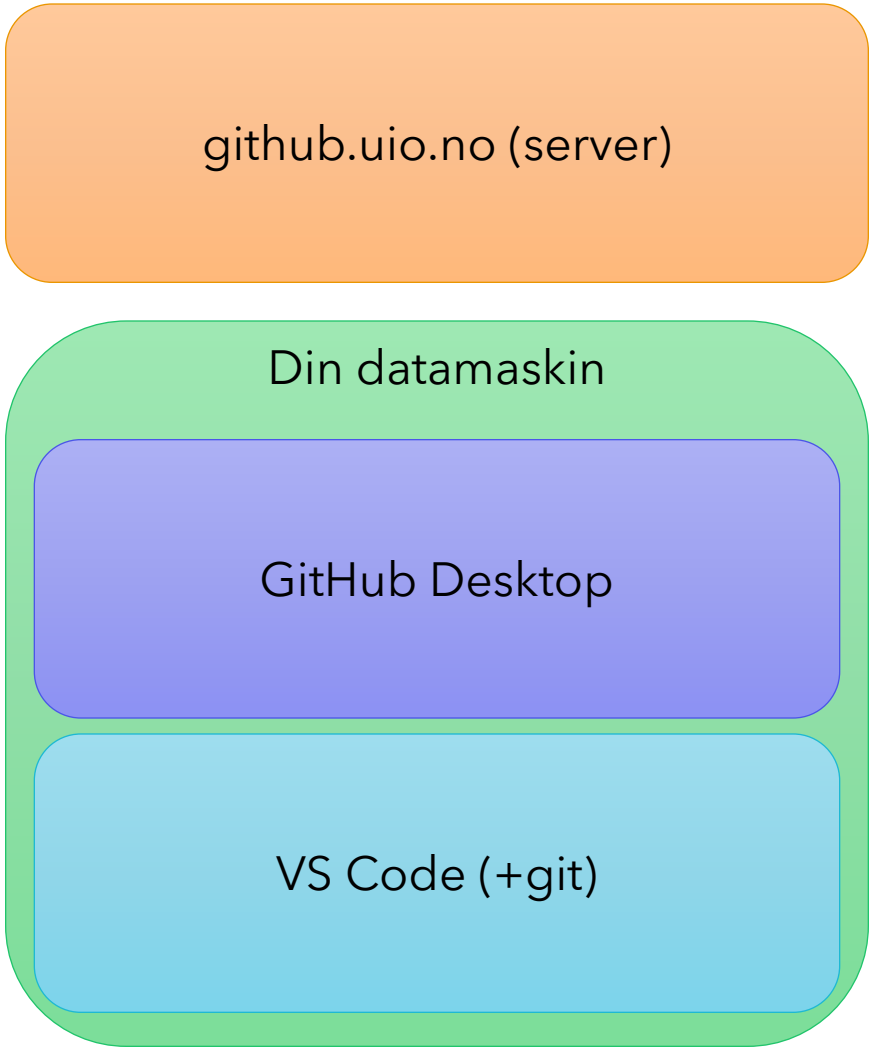
Noen flere begreper

```
1    def f(x):
2        return x*2
3
4    y = f(2)
```

Parameter	**x**
Argument (inndata)	**2**
Returverdi (utdata)	**4**

2 → f(x) → 4

Hva skjer hvor?



- Alle versjoner av filene (med historikk) er lagret her
- Administrere prosjektet ditt
- Lese dokumentasjon (README)

- Opprette lokal kopi av repository på disken (clone)
- Endre aktiv branch som du jobber i

- Inkludere (stage) endringer til en ny commit
- Lage commit med melding om hva som er endret
- Dytte (push) commit opp til server

Når du har skrevet litt kode…

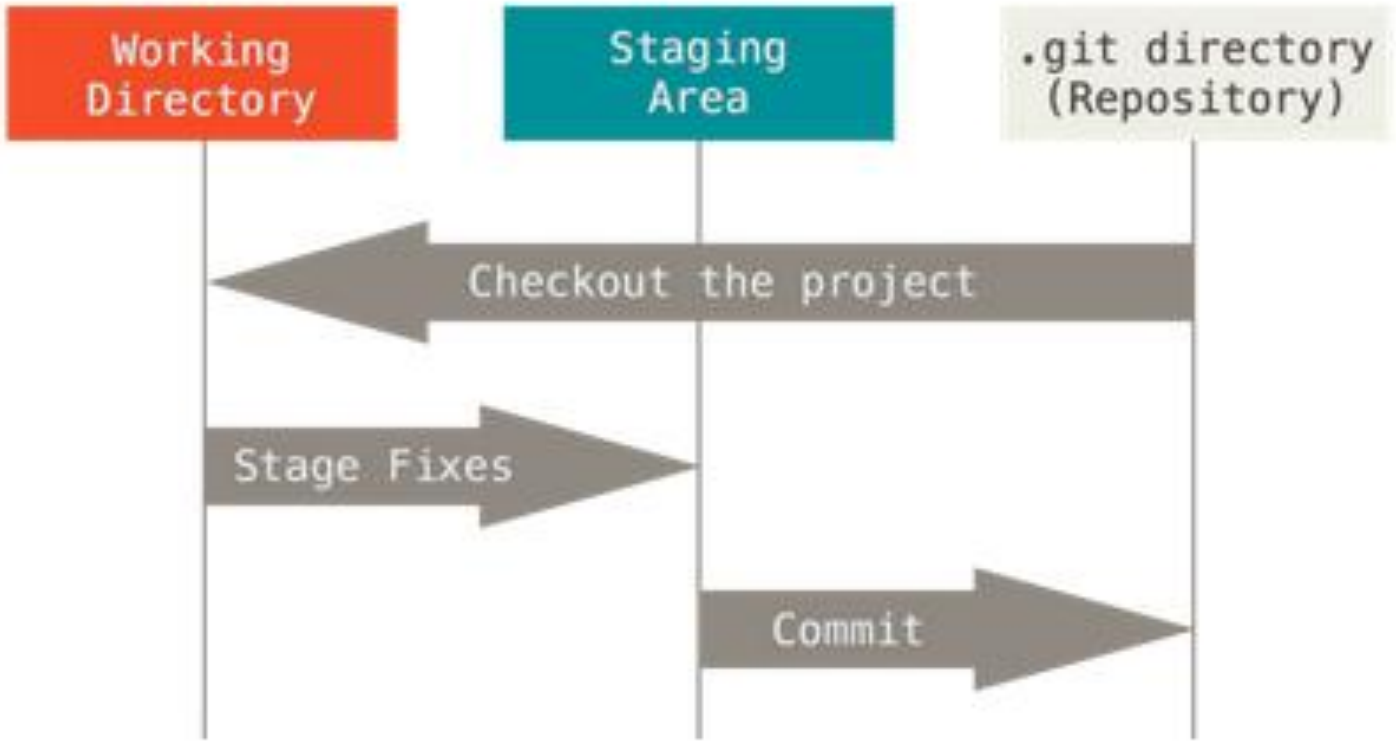
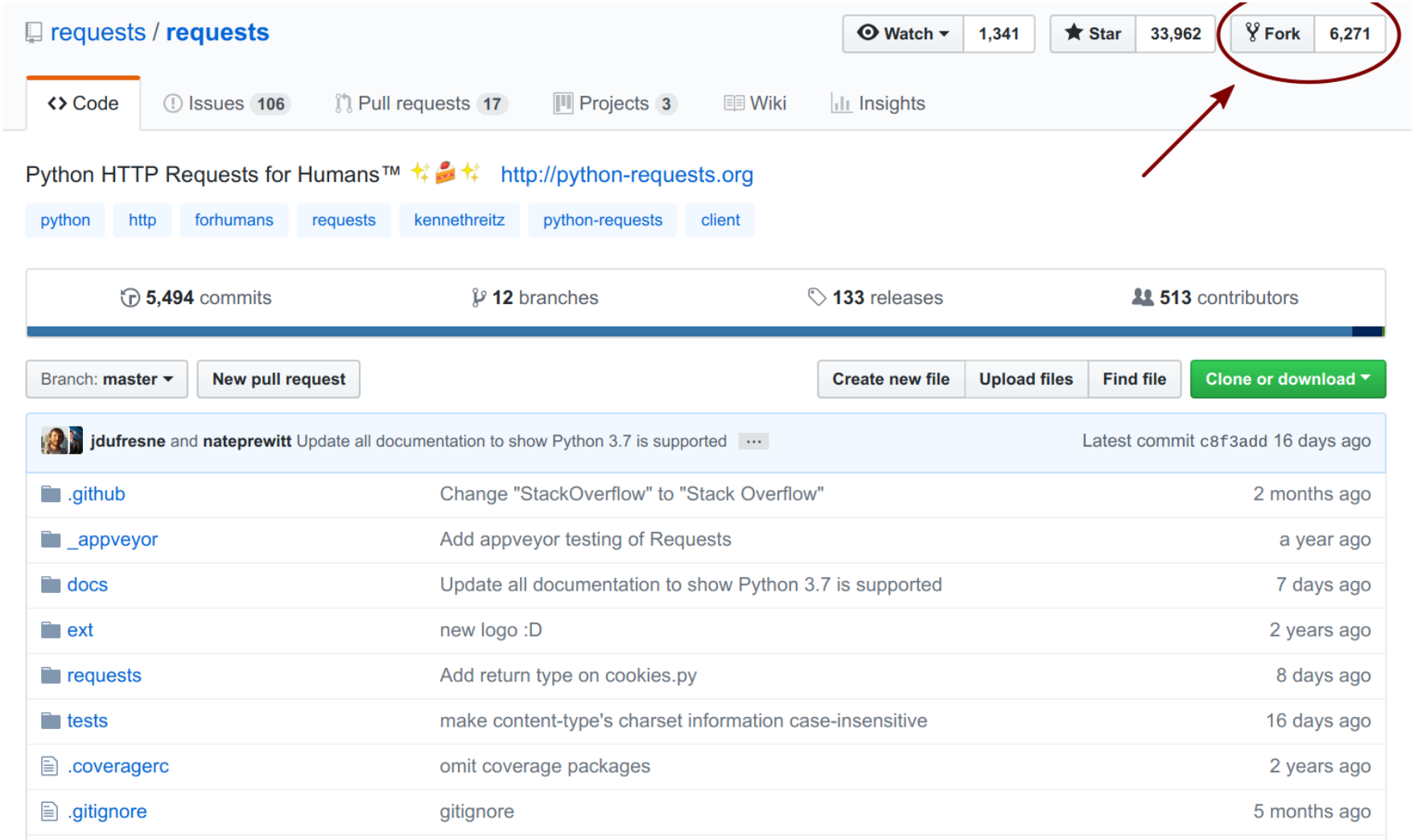
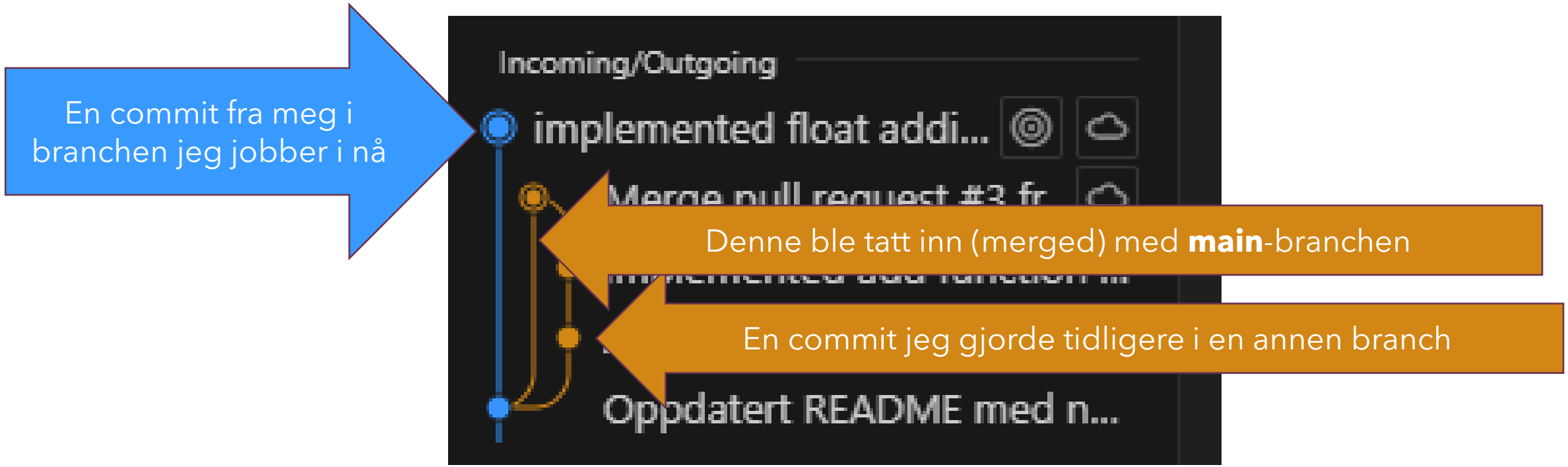


Fig. 16 Source: Chacon & Straub, ProGit (2014)

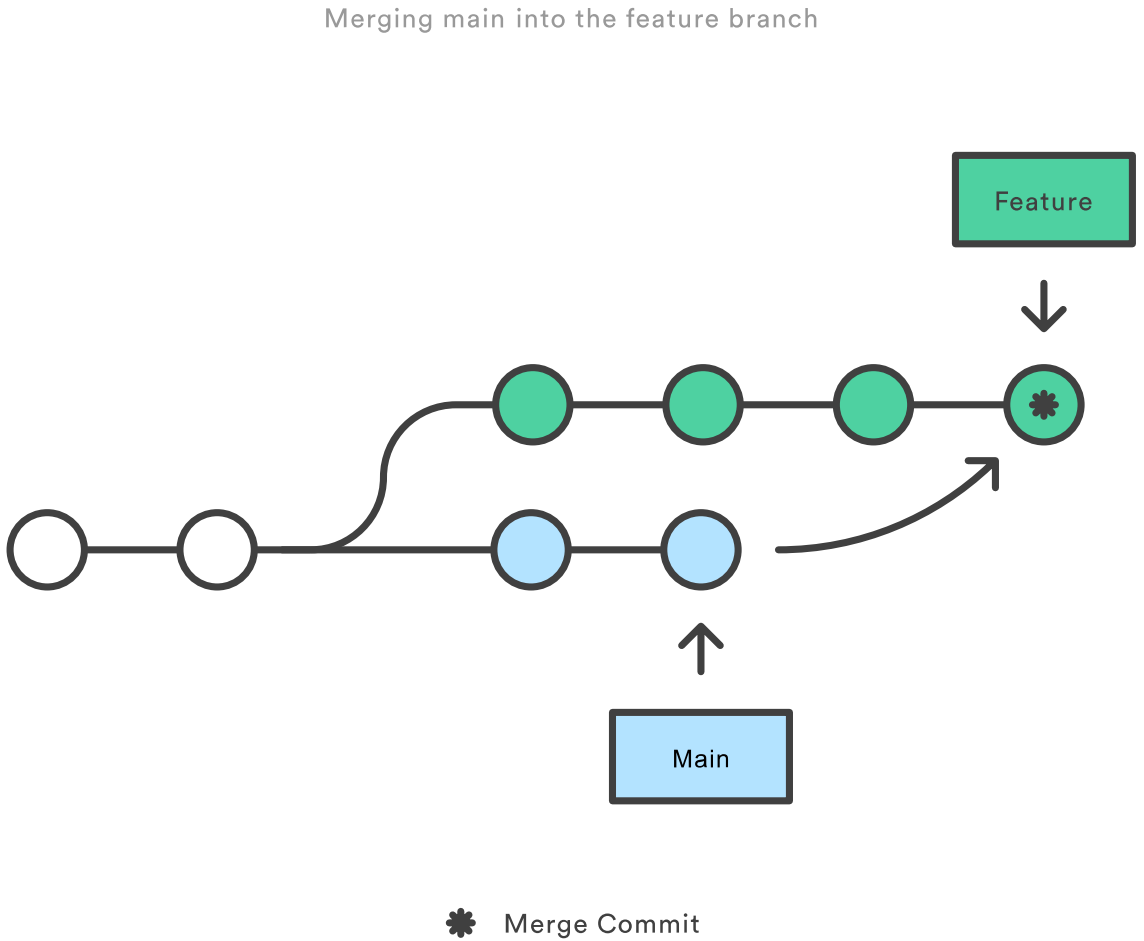
Forking: Lage egen versjon av noen andres repository



Forgrening (branching): Jobbe parallelt i forskjellige grener av prosjektet

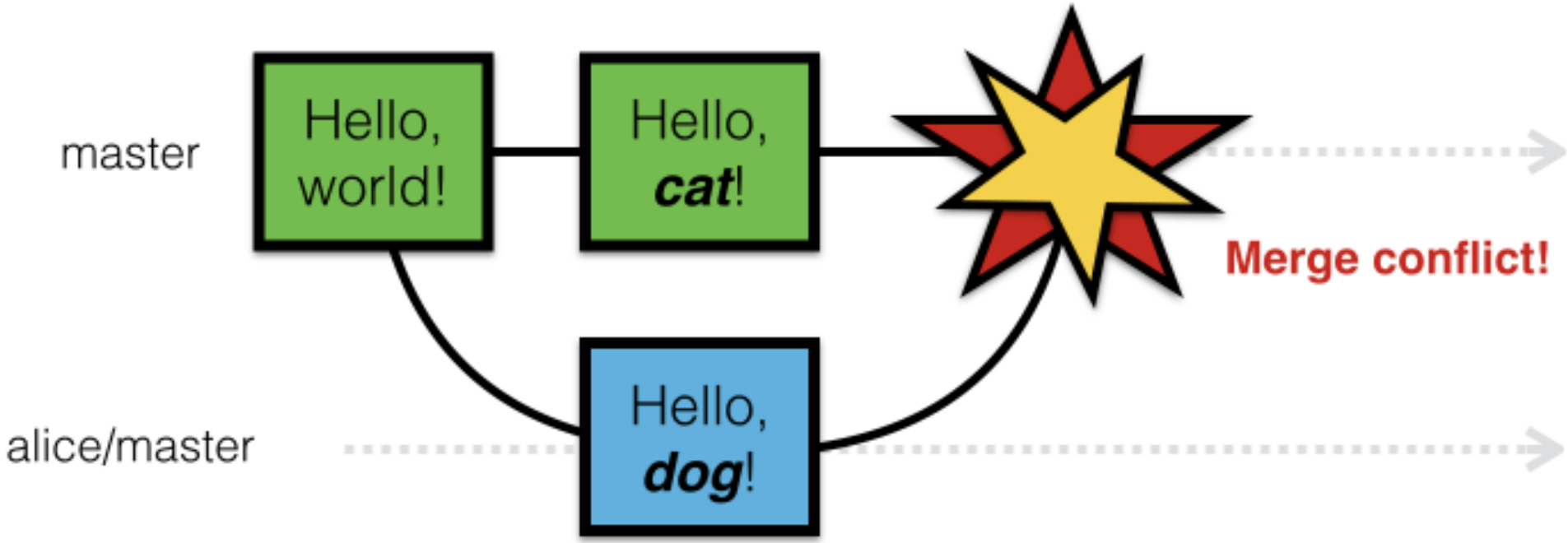


Merge: Slå sammen ulike grener



https://www.atlassian.com/git/tutorials/merging-vs-rebasing

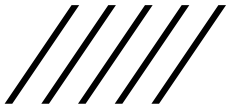
- Konflikter kan oppstå hvis samme kode endres i hver sin branch!



https://www.earthdatascience.org/workshops/intro-version-control-git/pull-request/

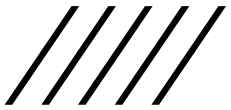
Pull requests

- Når man skal inkludere kode fra en branch inn i **main**, vil man i et større prosjekt lage en *pull request*
- Da må en prosjektleder godkjenne endringene som er gjort, og passe på at andre endringer som er gjort i **main** i mellomtiden ikke forårsaker problemer
- Når pull request'en er godkjent vil endringene merges



Motivasjon: Hvorfor kodestil, dokumentasjon og type-hinting?

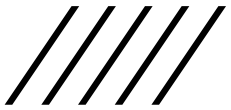
- *"Any fool can write code that a computer can understand. Good programmers write code that humans can understand"* - Martin Fowler
- Programmer som ikke er lette å lese og forstå er de som ikke blir brukt eller videreutviklet – lettere å finne opp kruttet på nytt
- Å vite typer på variabler hjelper på dette
- (I C++ og andre språk *må* vi gi variabler typer!)



Hva er kodestil?

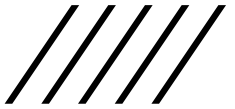
- Hvilken rekkefølge gjør vi ting i?
- Hvilke navn gir vi variabler, funksjoner og klasser?
- Hvordan bruker vi *whitespace* (mellomrom, linjeskift, …)?
- Hvordan kommenterer vi koden?

- Her er det mye som har lite å si for datamaskinen, men som har stor betydning for brukere, testere og de som skal videreutvikle programmet



Den viktigste anbefalingen

- **Ikke følg disse anbefalingene dersom koden blir *mindre* leselig som et resultat av at du gjør det**
- (ingen regel uten unntak – vit når du kan og bør bryte reglene)
- PEP8 (og lignende veiledninger) er ikke endestasjonen for vakker og forståelig kode – kun et sted å starte



Hva er vakker kode?

So what is *beautiful* code then?

Beautiful code = maintainable code. THAT'S IT!
THAT'S THE FORMULA!

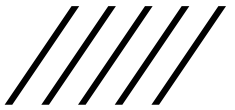
If you can write something, come back to it a few months later, and continue making progress on it, then that's beautiful. If a year later you realize that you want to add functionality as well as tweak an existing feature, and you manage to do it with relative ease, then THAT'S beautiful. If other people can step into your codebase and quickly figure out what's going on because things are organized, they'll have more hair, and also be beautiful.

https://softwareengineering.stackexchange.com/questions/207929/what-is-beautiful-code

Trevor Hickey
1,023 ● 1 ● 10 ● 16

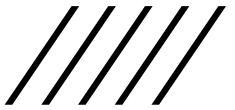
Dokumentasjon

- Se for deg at du er en annen person som åpner ditt GitHub-repository for første gang
- Vil de umiddelbart skjønner hva koden din gjør?
- Vil du *selv* skjønne det etter lang tids fravær fra denne koden?
- **Forklar *hva* koden gjør, ikke (bare) *hvordan* den gjør det**



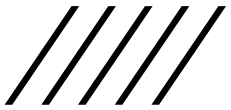
Type-annotasjoner (type-hinting)

- Selv om Python ikke *krever* at vi oppgir typer til variabler, har vi likevel *lov* til å gi beskjed om hvilken type vi forventer!
- Kan gjøres på alle variabler, men vi skal hovedsakelig bruke det til parametre og returverdier i funksjoner



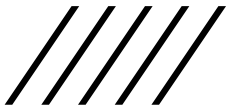
Hvorfor?

- **Dokumentasjon:** Sammen med docstrings forklarer type-hintene hva som er inndata og utdata til en funksjon
- **Editor:** Vi kan få ekstra hjelp fra editoren når den vet typen til en variabel (kan slå opp metoder denne typen har)
- **Feilsøking:** Vi kan bruke en statisk type-sjekker som **mypy** til å se om noen objekter har fått andre typer enn forventet, som en del av testing/debugging



Motivasjon: Test-drevet utvikling

- Vi ønsker å skrive *pålitelige* programmer som…
- …er grundig testet før de tas i bruk
- …heller kræsjer og gir feilmelding enn
  å gi feil resultat uten feilmelding
- …kræsjer på en måte som lar oss forstå hva som gikk galt
- …ikke ødelegger datafiler hvis de kræsjer (kræsjer kontrollert)

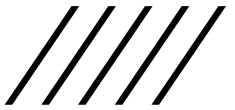


Noen nyttige verktøy

- assert
- exceptions
- pytest
- test-drevet utvikling (en måte å tenke og jobbe på)
- exit-koder

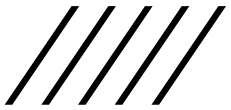
…så hva bør jeg bruke når?

- Bruk **assert** for å teste for feil som normalt *aldri* bør skje (så dette kan rettes opp i før brukere kjører programmet)
- Bruk **exceptions** for å teste for feil som kan *forventes* å skje når brukere kjører programmet, så brukere opplever at feilen håndteres fornuftig av programmet
- Du kan tenke på **assert** som et utviklingsverktøy og **exceptions** som en naturlig del av et brukervennlig program



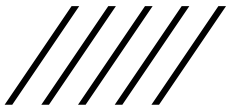
Enhetstesting (unit tests)

- En “unit” er en liten del av et større program
- En enhetstest tester om denne delen, isolert sett, gjør det den skal
- Test-drevet utvikling handler om å skrive enhetstester for hver del samtidig som vi lager selve delen
- En stor fordel er at disse testene kan kjøres automatisk
- Hvis vi senere endrer noe som gjør at en del som fungerte begynner å gjøre feil, så vil dette automatisk fanges opp!



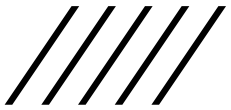
Avrundingsfeil (float)

- Datamaskinen bruker totallssystemet: 0, 1, 10, 11, 100, 101, …
- Så da burde et tall som 2.2 gå veldig fint?
- $2.2 = 2\frac{1}{5} =$
  $1 \cdot 2^1 + 0 \cdot 2^0 + 0 \cdot 2^{-1} + 0 \cdot 2^{-2} + 1 \cdot 2^{-3} + 1 \cdot 2^{-4} + 0 \cdot 2^{-5} + \ldots$
- 2.2 i totallssystemet blir 10.00110011001100110011001100…
- Evig repeterende siffer krever uendelig minne i maskinen
- Det blir en avrunding til slutt, som kan gi *avrundingsfeil*



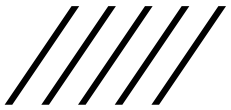
print-statements for å finne feil

- Ulempe: Må fjernes/skjules etterpå, så brukerne ikke trenger å se masse debugging-informasjon
- Kan likevel være nyttig å lagre debugging-informasjon et sted i tilfelle det oppstår en feil i fremtiden
- Bedre alternativ: logging (til fil eller terminal)
  - Nivå 1: Kun advarsler
  - Nivå 2: Advarsler og informasjon
  - Nivå 3: Advarsler, informasjon og debug-meldinger



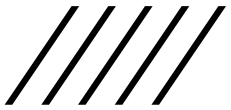
Debuggeren i VS Code

- Kan pause programmet der vi vil og følge med på hva som skjer med objekter/variabler
- Eventuelt pause bare når en betingelse er oppfylt (for å slippe de 100 første stegene der feil *ikke* skjer)
- Kan velge om vi vil jobbe på høyt nivå (ikke se hva som skjer inni en funksjon som kalles) eller lavt nivå (gå inn i funksjonen)
- Kan endre verdier av variabler for å teste
- Kan logge til en egen terminal uten å logge i selve koden



Motivasjon: Hvorfor objektorientering?

- En måte å tenke på som organiserer programmer og gjør ting mer oversiktlig (i mange tilfeller)
- I stedet for *passive* samlinger med data (list, dict ++) som må sendes inn til funksjoner for å bli gjort noe med…
- …kan man lage litt mer *selvstendige* objekter som i tillegg til å inneholde data selv vet hva de skal gjøre med disse dataene
- passiv: **resultat = gjør_noe_med(en_samling)**
- selvstendig: **resultat = et_objekt.gjør_noe_selv()**

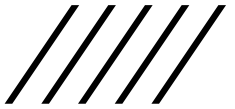


De fire fordelene med OOP (de fire *pillarene*)

- Abstraksjon
- Innkapsling
- Arv
- Polymorfisme

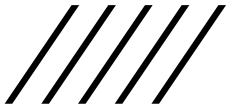
Atributter

- Variabler som alle objekter av en klasse har
  - alle Kuler har en .radius
- Men som kan ha forskjellig verdi for hvert objekt
  - Kuler har forskjellig radius
- Ved å gjøre en atributt til en *property,* kan vi kontrollere hva som skjer når verdien leses (*getter*) og endres (*setter*)



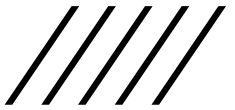
Privat og offentlig informasjon i Python

- _ foran variabel- eller metodenavn er et signal til leseren at denne variabelen eller metoden er til intern bruk
  (ikke en del av grensesnittet)
- __ foran variabel- eller metodennavn gjør det vanskelig (men ikke umulig) å se denne variabelen eller metoden utenfra klassen
- Unntak: __metodenavn__ er reservert for *magiske metoder*



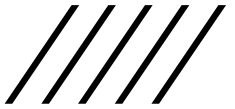
Magiske metoder

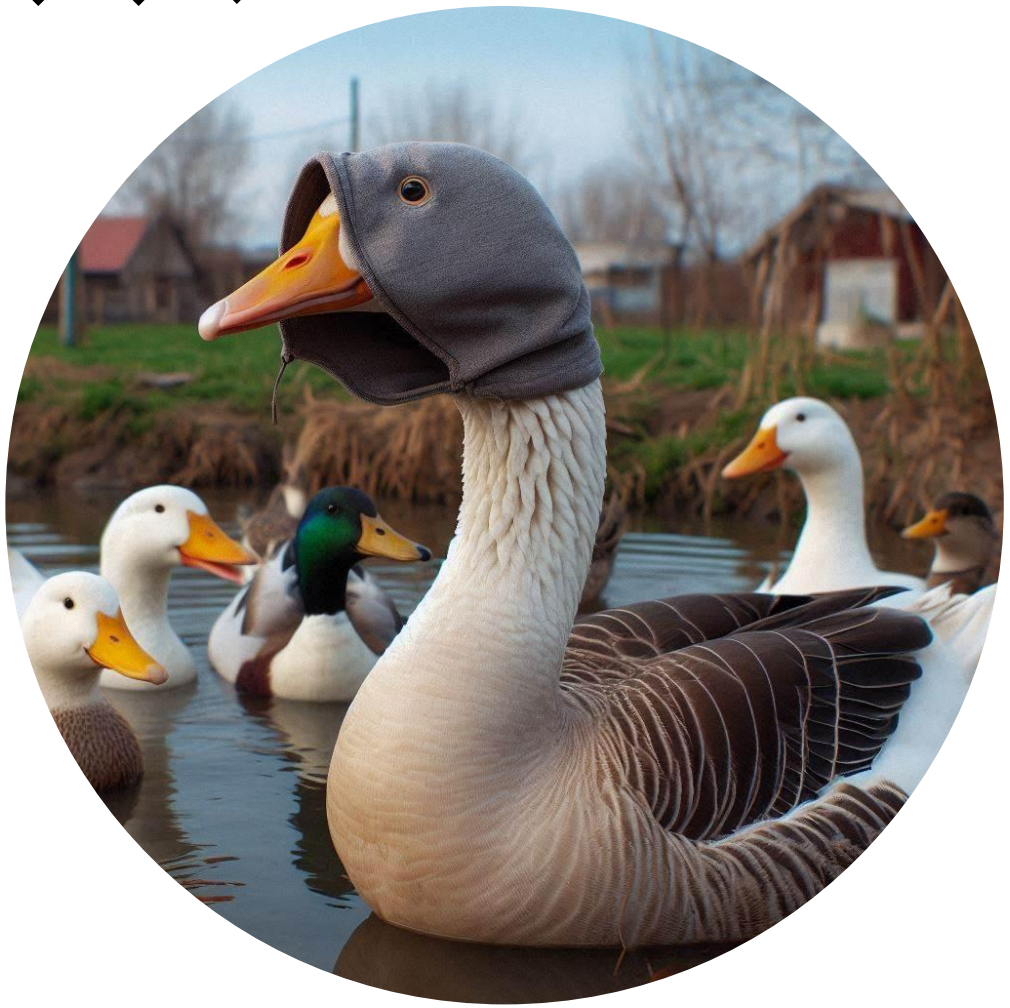
- **__str__** kalles automatisk når vi gjør objektet til en **str**:
  - **str(et_objekt)**
  - **print(et_objekt)**
- **__len__** kalles automatisk når vi kaller
  - **len(et_objekt)**
  - Bør normalt ikke brukes hvis ikke objektet inneholder referanser til et antall andre objekter (en Kortstokk inneholder flere Kort)



Bør vi alltid bruke arv?

- Nei, ikke alltid!
- Arv *kan* være fornuftig hvis
  - Vi vil ha en ny klasse som gjør mye av det en annen klasse gjør + noe ekstra (slipper å copy-paste kode)
  - Du har flere klasser som gjør akkurat det samme, og vil oppdatere dem *ett sted* (i moderklassen) i stedet for å oppdatere dem hver for seg (**Goose** er oppdatert, men du glemte **Duck** og **Swan**)
  - Du bruker type-sjekking og synes **b: Bird** er mer oversiktelig enn **b: Union[Duck, Goose, Albatross]**





Hva er alternativet?

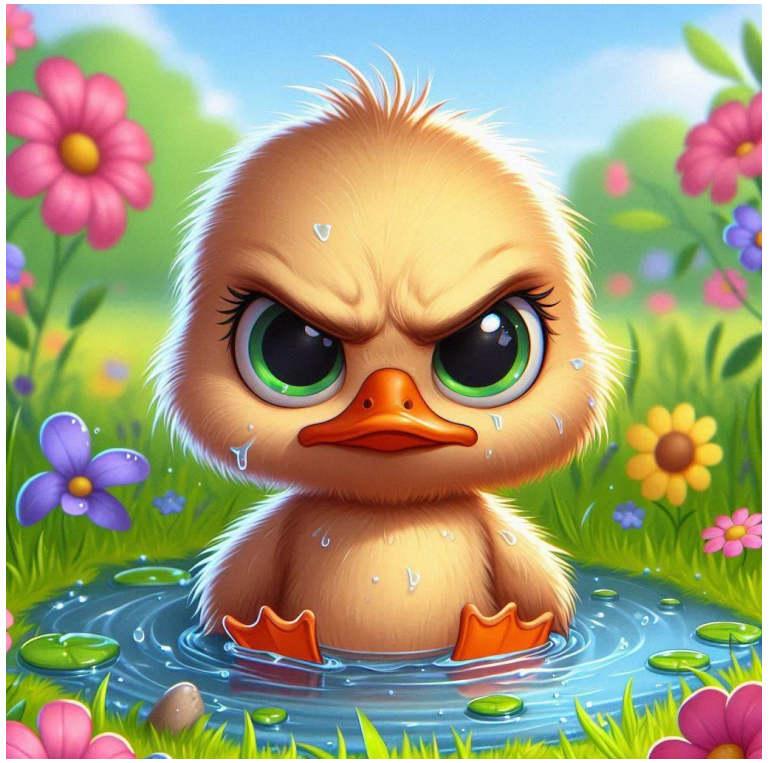
- *Duck typing* (Python):
  - “if it **.swim()** like a **Duck** and **.fly()** like a **Duck**, it must be a **Duck**”
- alle klasser med de rette metodene gjør at koden kjører uten feil, selv om det ikke er “riktig” klasse
- (men **mypy** gjennomskuer det)

Fordeler med duck typing i Python

- Koden blir enkel og fleksibel når du kan fokusere på hva objektene *gjør* og ikke hva de *er*
- Du kan lage enkle *prototyper* av mer kompiserte klasser for å teste annen kode i en tidlig fase
- Du kan gjenbruke en klasse et annet sted uten å importere *alle* klassene som arver fra hverandre

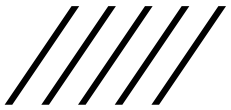
Ulemper med duck typing i Python

- Programmet kan kræsje eller gi feilmelding når det kjøres fordi det mangler en metode eller atributt som man antok var der
- Det kan være vanskelig å vite hva slags objekt det er snakk om når man leser koden
- Det kan bli mer krevende å finne feil når typene er udefinert



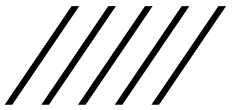
Definere grensesnitt (interface) med abstract base classes (ABCs)

- En annen måte å bruke arv på
- Lager en *abstrakt klasse* = en du ikke kan lage objekter av (du kan ikke lage en **Swimming**, men du kan lage en **Duck**)
- En klasse som *arver* fra denne klassen må *implementere* alle metodene i grensesnittet
- Hvis du glemmer en får du **NotImplementedError**
- Kan dermed *tvinge* et objekt til å ha bestemte metoder fra ett (eller flere) grensesnitt



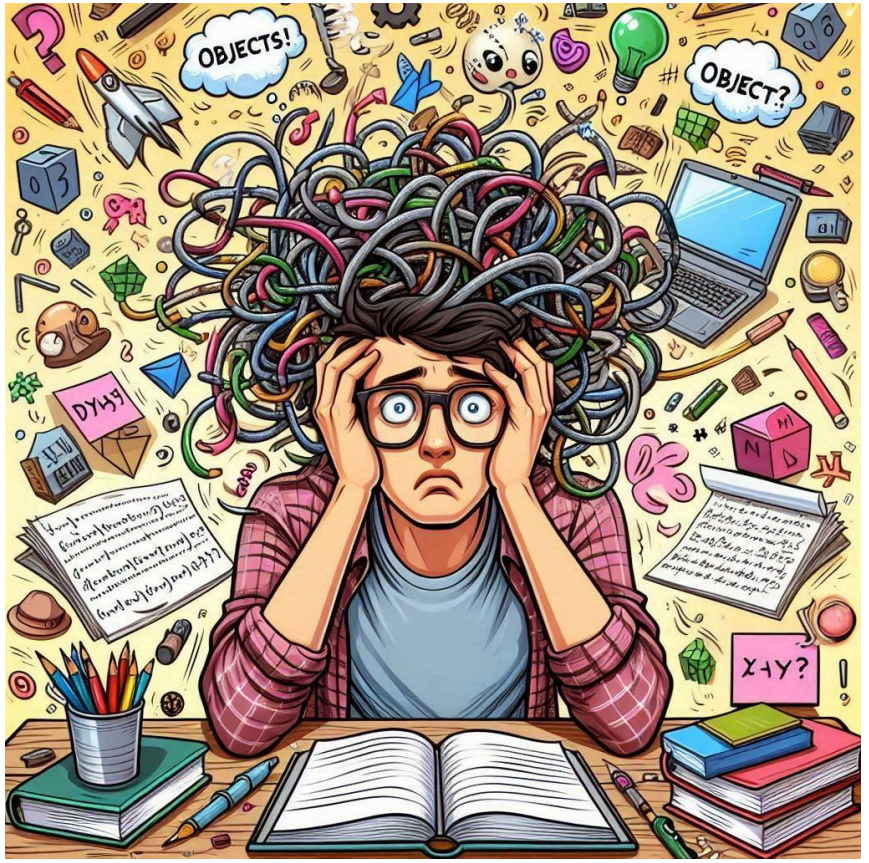
Dataobjekter – en måte å jobbe på:

- Prøv først med å arve fra NamedTuple
  - Hvis det ikke funker (f.eks. hvis noe må endres)…
- Prøv en dataclass
  - Hvis det heller ikke funker
    (f.eks. hvis __init__ må gjøre mer enn å bare sette atributter)…
- Bruk en vanlig klasse
  - (fordel: ingen import nødvendig)
- Dette er ikke en regel, bare et forslag!



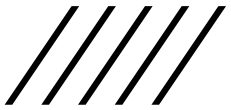
Nesten *alt* i Python er objekter!

- Det tomme objektet **None** er et objekt (av klassen **NoneType**)
- Funksjoner er objekter (av klassen **function** eller **builtin_function_or_method**)
- Objekters metoder er objekter (av klassen **method** eller **builtin_function_or_method**)
- Selv *klasser* er objekter (av klassen **type**)



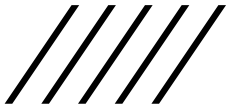
Noen begreper

- Overload = flere metoder med samme navn og ulike parametre
  - **def spill_kamp(k: Kamp) -> None:**
  - **def spill_kamp(hjemmelag: Lag, bortelag: Lag) -> None:**
  - Funker ikke direkte i Python – den siste overskriver den første!
- Override = overskrive metode fra foreldreklassen (ved arv)
  - **KlassiskKortstokk.__init__** kan gjøre noe annet enn **Kortstokk.__init__**
  - **Pendulum._create_result** kan gjøre noe annet enn **ODEModel._create_result**



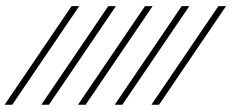
Arrays vs lister

Array	Lenket liste
Kun en datatype	Flere datatyper mulig
Fast størrelse	Vokser og krymper etter behov
Elementvise operasjoner (numpy array i Python)	Må bruke løkke for å gjøre noe med alle elementene
Rask og spesialisert	Treig og generell
Bruker indekser (direkte tilgang til elementer)	Må alltid starte først/sist (Python har indeksering)



Er arrays og pekere samme ting?

- Nei, det er viktige forskjeller (f.eks. **sizeof**)
- Men de bruker indeksering helt likt
- **en_array[2]** returnerer i teorien tredje element i en array
- **en_peker[2]** hopper 2 steg (steglengde i bytes avhenger av type) videre i minnet og tolker hva enn som ligger der som en verdi av typen til pekeren
- Når vi bruke **int* _data = new int[_capacity]** lager vi en peker til første element i en array (men indeksering vil altså funke)

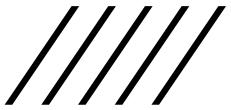


Variabler, pekere, referanser

Symbol	Etter type	Før variabelnavn
(ikke noe symbol)	Variabel (heltall) **int x = ...**	Verdi **... = x;**
*	Peker (minneadresse) **int* x = ...**	Verdi på minneadressen **... = *x**
&	Referanse **int& x = ...**	Minneadresse **... = &x**

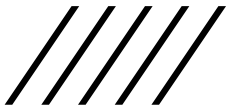
Feilmelding hvis x ikke er en peker

- **int y = x** *kopierer* verdien til et annet sted i minnet (&y != &x)
- **int& y = x** gjør at x og y peker på samme verdi (&y == &x)



Litt om bits og bytes

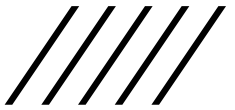
- En bit (*binary digit*) er et siffer i totallsystemet: 0 eller 1
- En *byte* er en etterlevning fra tiden da datamaskiner behandlet 8 bits av gangen (1 byte = 8 bits)
- 01001000 og 01101001 er eksempler på bytes i minnet
  - Kan tolkes som bokstavene “H” og “i” (char)
  - Eller heltallene 72 og 105 (int)
- Nå behandler maskinene *vanligvis* 64 bits av gangen (8 bytes) men byte som enhet har overlevd av historiske årsaker



- Python har mer sikkerhetsnett enn C++
  (C++ går ofte rett på minnet i maskinen)

To måter å sende et argument til en funksjon på når den kalles

- "Call by reference"
  - Du har en variabel, og verdien ligger et sted i minnet
  - Du gir funksjonen er *referanse* (adresse) til hvor i minnet dette er
  - Hvis funksjonen endrer verdien i minnet, er den også endret for variabelen i programmet som kaller funksjonen
- "Call by value"
  - Du kopierer verdien til variabelen, slik at funksjonens variabel er et annet sted i minnet (med samme verdi)
  - Endrer du verdien i funksjonen, er programmets variabel uendret



Lenket liste

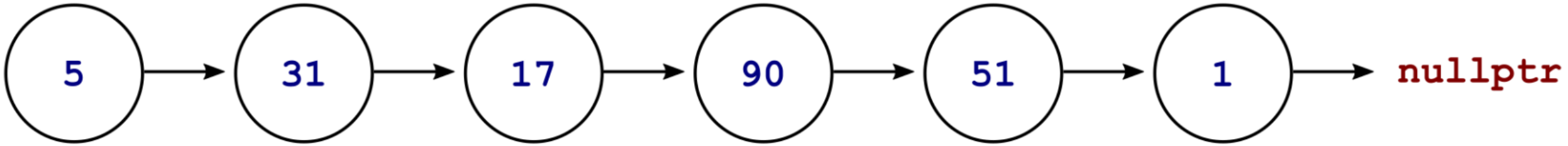


Fig. 33 A simplified drawing of a linked list.

En lenket liste består av Node-objekter med referanser til neste Node

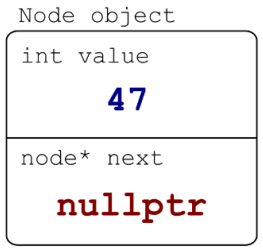


Fig. 31 A single node object. #

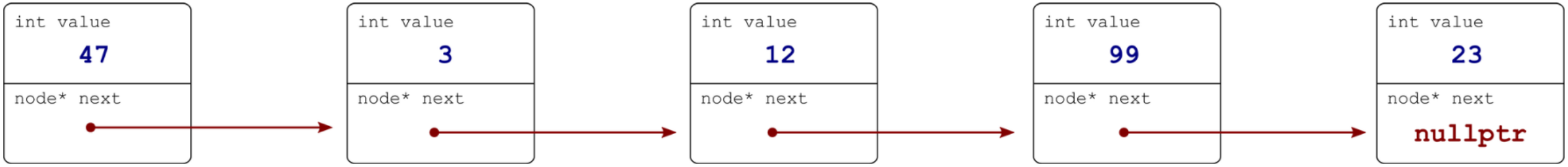
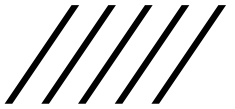
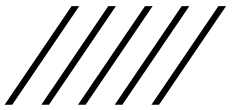


Fig. 32 A chain of node objects. #



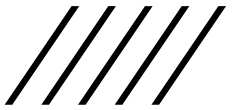
Forskjell på **struct** og **class**

- Det kan være smart å bruke **struct** når vi bare trenger dataobjekter uten metoder (men de kan ha metoder også)
- Alt i en **struct** er **public** som standard
- Alt i en **class** er **private** som standard (hvis ikke vi spesifiserer at det skal være **public**)



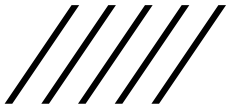
Viktig forskjell på variabel og peker

- En variabel (**Node ny_node**) ryddes opp når den ikke lenger er definert (f.eks. funksjonen der den defineres er ferdig)
- En peker (**Node* ny_node**) ryddes ikke opp av seg selv
- Hvis vi fortsetter å bruke en peker med adressen til en variabel som ikke lenger er definert, kan det være et helt annet objekt der enn vi tror (minnet gjenbrukes når det er ledig!)
- Kompilatoren vil ofte advare oss hvis vi prøver på dette



Minnelekkasje

- Vi kan lage et objekt direkte med en peker isteden:
  Node* ny_node = new Node();
- Men da må vi rydde opp selv: **delete ny_node**
- Hvis vi ikke rydder opp, kan vi få en *memory leak* – vi reserverer mer og mer minne (med **new**) uten å gi beskjed om at disse minneadressene ikke trengs lenger

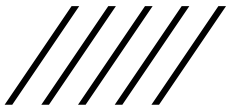


Algoritmeanalyse

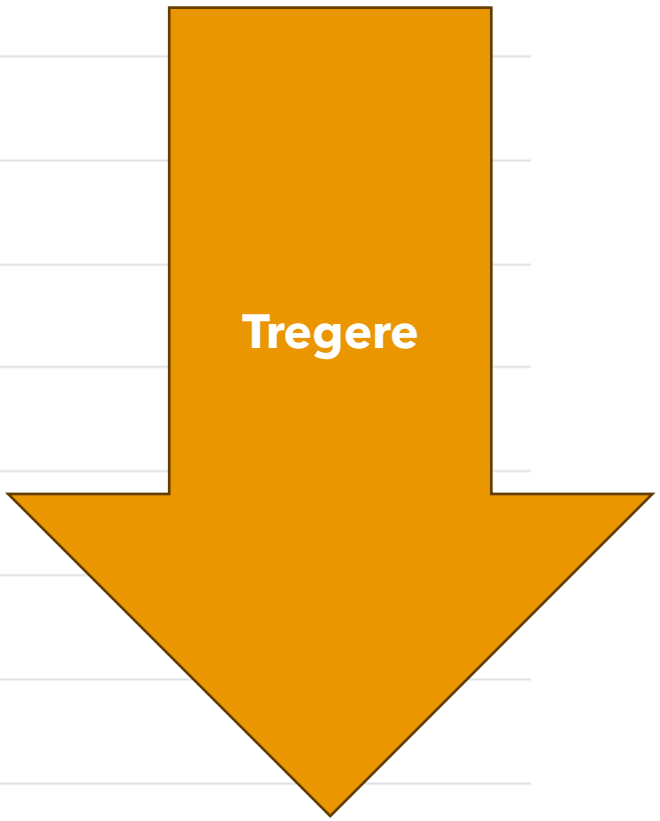
```
1   def find_biggest(numbers: list[int]) -> int:
2       biggest = numbers[0]
3       n = len(numbers)
4
5       i = 1
6       while i < n:
7           if numbers[i] > biggest:
8               biggest = numbers[i]
9           i = i + 1
10
11      return biggest
```

- 2
- 2
- 1
- n
- 2(n-1)
- 0 – 2(n-1)
- 2(n-1)
- 1

- "Best case": $6 + n + 4(n-1) = 5n + 2$
- "Worst case": $6 + n + 6(n-1) = 7n$



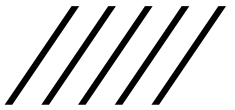
Big O	Name
$\mathcal{O}(1)$	constant
$\mathcal{O}(\log n)$	logarithmic
$\mathcal{O}(n)$	linear
$\mathcal{O}(n \log n)$	loglinear/quasilinear
$\mathcal{O}(n^2)$	Quadratic
$\mathcal{O}(n^3)$	Cubic
$\mathcal{O}(n^k)$	Polynomial
$\mathcal{O}(e^n)$	Exponential
$\mathcal{O}(n!)$	Factorial



Note that an algorithm can also have fractional exponent scaling, such as $\mathcal{O}(n^{1.5})$, but this is rare.

Stor-O-notasjon

- Når kjøretiden er $an + b$ har vi **lineær** scaling med n
- Leddet som dominerer når n blir stort er det vi fokuserer på
- Vi sier at skaleringen er av orden $O(n)$ for denne algoritmen



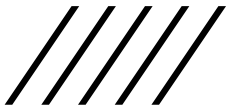
Amortisert kjøretid

In conclusion, while each resize might be costly, we carry out so few of them that the *total cost* of all the resizes also becomes $\mathcal{O}(n)$ and the total cost of appending n elements to an empty `DynamicArray` is

$$(\text{cost of } n \text{ appends}) = \mathcal{O}(n) + \mathcal{O}(n) = \mathcal{O}(n).$$

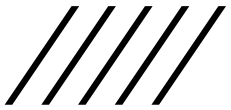
If doing n appends has a total cost of $\mathcal{O}(n)$, the average/amortized cost of a single operation must be

$$(\text{amortized cost of 1 append}) = \frac{1}{n} \cdot (\text{cost of } n \text{ appends}) = \frac{1}{n} \cdot \mathcal{O}(n) = \mathcal{O}(1).$$



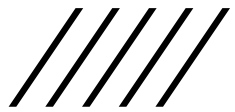
- Bubble Sort: $O(n^2)$

6 5 3 1 8 7 2 4



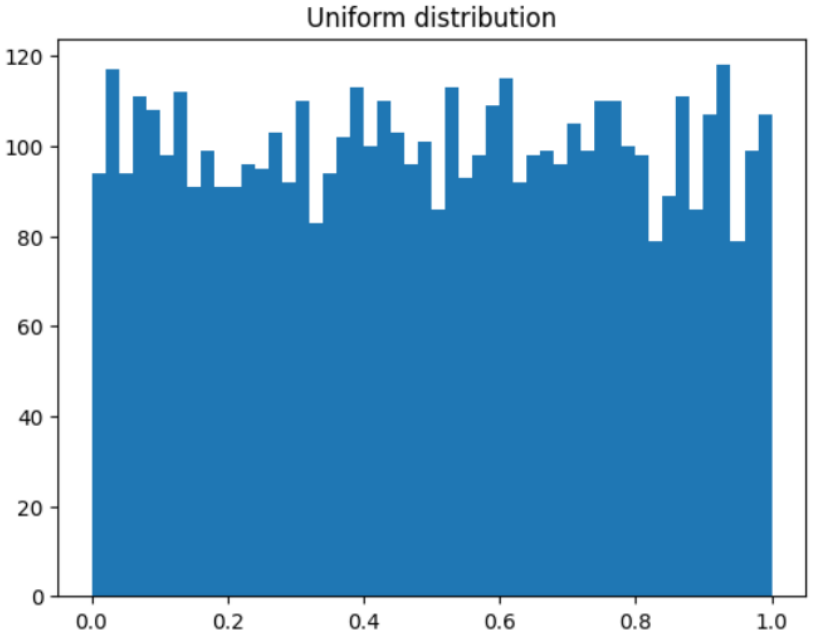
Hvor tilfeldige er tilfeldige tall?

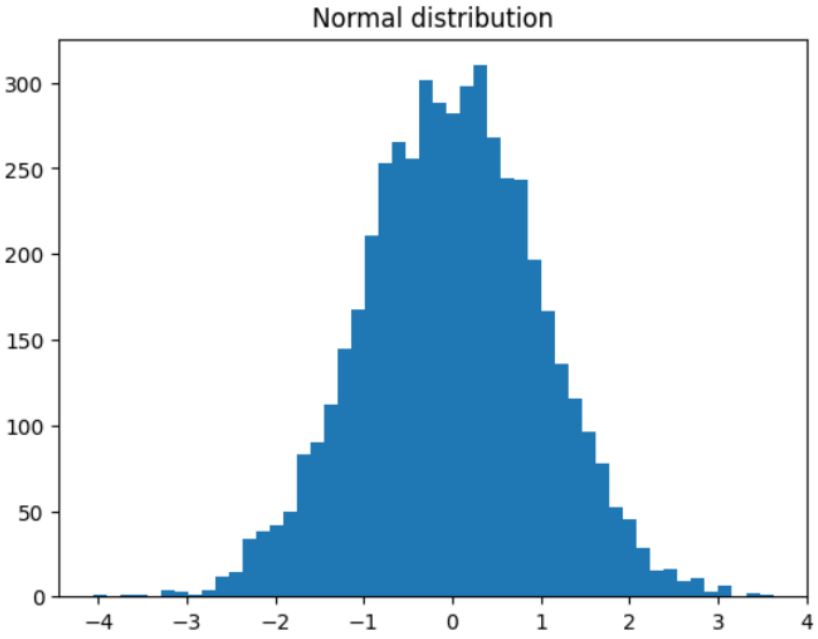
- Egentlig ikke tilfeldige i det hele tatt – datamaskiner er deterministiske
- Hvis man gir en RNG (*random number generator*) samme start-tilstand (*seed*) vil den alltid gjenskape de samme tallene
- Derfor viktig å ikke starte med et fast seed, da vil programmet bare får de samme tallene hver gang det kjøres (unntak: testing)
- En mulighet er å bruke antall sekunder siden midnatt på 1. januar 1970 (**time.time()** i Python) som seed, siden dette vil være forskjellig hver gang.



Sannsynlighet

- En lang pseudorandom tallrekke kan brukes til å produsere uniformt fordelte desimaltall mellom 0 og 1
- Hvis vi f.eks. vil ha tall mellom 5 og 10 kan vi ta $y = 5x + 5$ der $x \in [0, 1)$
- Kan også konvertere til andre sannsynlighetsfordelinger

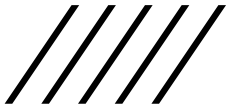




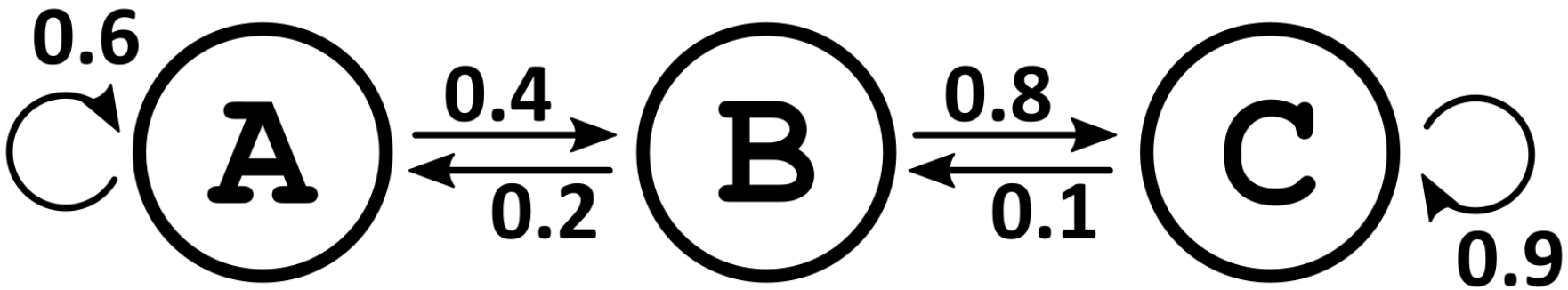
If we want to shift the standard deviation of the normal distribution or its mean, we can simply multiply each sample by σ, and if we want to move the mean, we simply add μ to each sample.

Velg riktig bilbiotek!

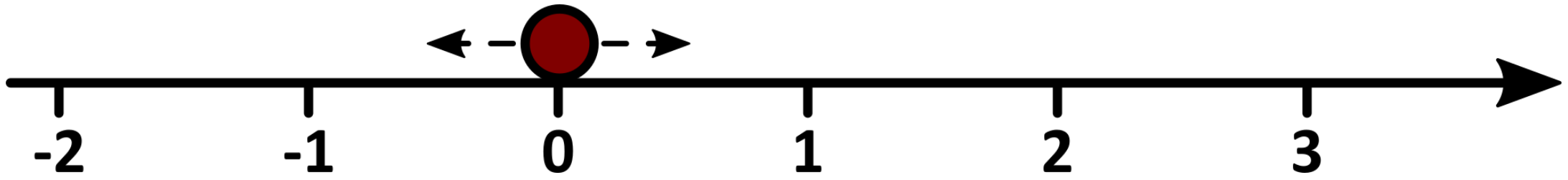
- Skal du lage noe som har med kryptering og sikkerhet å gjøre? **import secrets**
- Trenger du mange tilfeldige tall på en gang? **import numpy**
- Ingen av de to over? **import random**



Markov-kjeder



En type endimensjonal virrevandring



$$X_{N+1} = X_N + K_N,$$

$$K_N = \begin{cases} 1 & \text{with 50\% chance} \\ -1 & \text{with 50\% chance} \end{cases}$$

Gjennomsnittlig avvik fra snittet er generelt ikke så nyttig

- Positive og negative avvik fra snittet nuller hverandre ut
- Ingen forskjell på ingen spredning og symmetrisk spredning!
- I statistikk regner vi isteden ut det gjennomsnittlige *kvadratavviket* slik at både negative og positive avvik bidrar til spredningsmålet
- Dette bruker så ofte at det har fått navnet *varians*
- For å få samme enhet som et datapunkt (og snittet) tar vi ofte kvadratroten etterpå → *standardavvik*

Vi kan også finne RMS (Root Mean Square)

$$\langle X_N^2 \rangle = N,$$

$$\mathrm{RMS} = \sqrt{\langle X_N^2 \rangle} = \sqrt{N}.$$

- OBS: RMS er bare likt standardavviket når gjennomsnittet er 0 (slik som her), ikke generelt!

RMS for 2D-vandreren vår

$$\langle \vec{R}_N \rangle = (\langle X_N \rangle, \langle Y_N \rangle).$$

$$\langle \vec{R}_N \rangle = (0, 0).$$

X og Y uavhengige 1D-størrelser

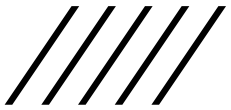
$$\langle |\vec{R}_N|^2 \rangle = \langle X_N^2 \rangle + \langle Y_N^2 \rangle.$$

$$\left|\vec{R}\right|^2 = \left(\sqrt{\vec{R} \cdot \vec{R}}\right)^2 = \vec{R} \cdot \vec{R}$$

$$\langle |\vec{R}_N|^2 \rangle = 2N.$$

Bruker 1D-resultatet

$$\mathrm{RMS} = \sqrt{\langle |\vec{R}_N|^2 \rangle} = \sqrt{2N}.$$



Markov-kjeder og virrevandring

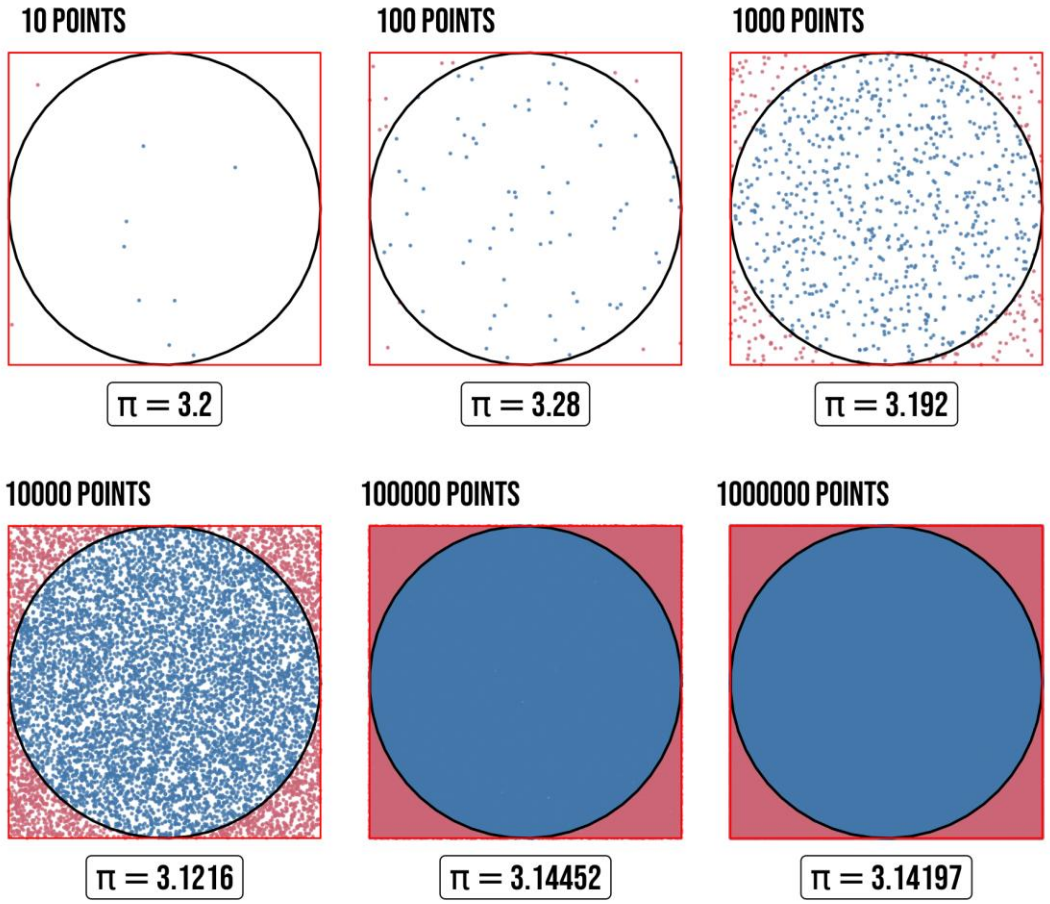
- Virrevandring er spesialtilfelle av de mer generelle Markov-kjedene
- "Minneløst" system – det eneste som påvirker neste bevegelse er tilstanden til systemet i forrige steg (ikke tid, ikke historikken før forrige steg)
- Vi kan *simulere* hvordan systemet oppfører seg i ett eller mange eksperimenter (Monte Carlo-simulering)
- Eller analysere sannsynlighetene uten å simulere

Forplantningsmatrise (Markov-kjede)

$$M = \left[\begin{pmatrix} M_{0,0} & \cdots & M_{0,m} \\ \vdots & \ddots & \vdots \\ M_{m,0} & \cdots & M_{m,m} \end{pmatrix} \right]$$

- Her er $M_{i,j}$ sannsynligheten for å gå fra $x = i$ til $x = j$
- $\vec{p}_{k+1} = \vec{p}_k \cdot M$
- $\vec{p}_k$ og $\vec{p}_{k+1}$ må være radvektorer for at M skal være m x m:
  (1 x m) = (1 x m)(m x m)

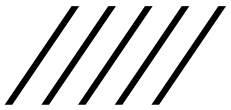
Monte Carlo-integrasjon



https://mauriciocely.github.io/blog/2020/08/05/estimating-pi-using-the-monte-carlo-method/

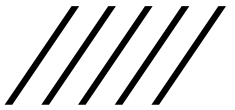
Når skal vi optimalisere kode?

1. Få det til å virke
   - Ikke tenk på kjøretid her
   - Test grundig og nøye at det virker
2. Få det til å virke *bra*
   - Gjør koden leselig og enkel å vedlikeholde (elegant kode)
   - Del opp i klasser/funksjoner, lag bedre variabelnavn osv.
3. Få det til å virke bra og *raskt*
   - Finn flaskehalsene og se om de kan gjøres raskere (må testes)
   - Algoritmeanalyse og evt. blandet programmering



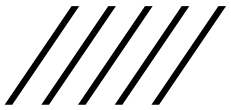
Ulemper med optimalisering

- Tar tid fra andre ting (ny funksjonalitet, bedre testing, osv.)
- Kan gå ut over leseligheten (i noen tilfeller) - hva er viktigst?
- Kan resultere i bugs
- Vi må derfor veie fordelene med raskere kjøretid opp mot disse ulempene



Hvis vi skal optimalisere, hvordan gjør vi det?

- Analysere algoritmer for å finne den teoretiske forskjellen mellom ulike fremgangsmåter
- Ta tiden på koden (benchmarking) for å finne ut hvor rask den er i praksis
- Profilering for å forstå hvilke deler av koden vi trenger å optimalisere

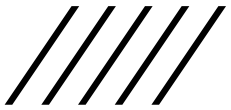


Benchmarking

- I kodesammenheng er det viktig at man da tester ulike versjoner av koden mot *samme* benchmark (og ikke endrer test-caset over tid)
- Kan da teste ulike tilnærminger med samme input og se på kjøretid og nøyaktighet (på output)

Alt vi har gjort til nå har vært *sekvensielt*

- Vi har skrevet programmer som kjører på én kjerne i datamaskinen (de andre brukes ikke)
- *Parallellisering* – å fordele programmets oppgaver mellom flere kjerner som jobber parallelt
- Krever litt ekstra kjøretid for maskinen å fordele og holde styr på de fordelte oppgavene, men lønner seg likevel i mange tilfeller

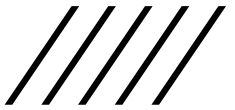


Noen begreper

- **Tråder** er styrt av operativsystemet: Lar en kjerne bytte mellom oppgaver så fort den må vente på et eller annet ("utålmodig prosessor") – hver tråd er da en oppgave

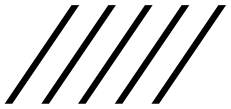
- **Parallellisering** er når en (stor) oppgave fordeles på flere prosessorer som må koordineres (samarbeide) for å løse den

- Disse kan kombineres



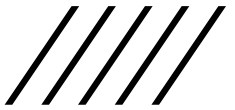
Ulemper med parallellisering

- Ekstra arbeid med å holde styr på prosessene som kjører parallelt (flere ikke alltid bedre)
- Hvis 100 000 personer skal male et hus, må alle få hver sin lille oppgave, hvert sitt utstyr, og vi må passe på at de ikke kolliderer med hverandre – dette kan da ta lenger tid enn å la en person male hele huset
- Prosessene deler ikke minne (vanskelig å samarbeide, bruker mer minne samlet sett)

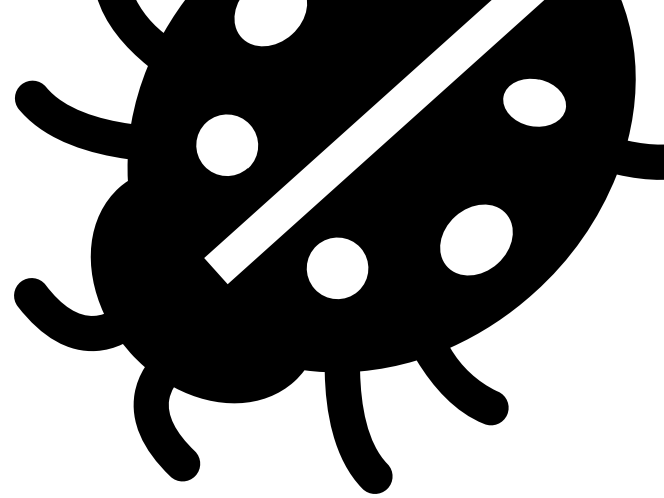


Ulemper med tråder

- Kan være vanskeligere å lese koden
- Får ikke utnyttet at maskinen har flere prosessorer
- Trådene deler minne, så de kan overskrive hverandres data ("race conditions")



“Race conditions”: Hvilken tråd kommer først?



Thread 1	Thread 2		Integer value
			0
read value		←	0
increase value			0
write back		→	1
	read value	←	1
	increase value		1
	write back	→	2

Thread 1	Thread 2		Integer value
			0
read value		←	0
	read value	←	0
increase value			0
	increase value		0
write back		→	1
	write back	→	1

Etter forelesningen

- Fredag 21/11: Spørretime og viktig eksamensinfo