Introduksjon til Python

September 22, 2022

Dette heftet inneholder støttematriell til **Introduksjon til Python**-kurset som ble holdt hos NVE i september 2022.

Ta gjerne kontakt med kursholder **Geir Arne Hjelle** på epost `geirarne@gmail.com` dersom du har spørsmål om innholdet.

Ditt Første Program

September 22, 2022

1 Introduksjon til Python

Python er et effektivt programmeringsspråk, hvor man raskt (lite tid) og konsist (få kodelinjer) kan utvikle skript og programmer. Python er allsidig og kan brukes til mange forskjellige oppgaver:

- Webutvikling: Python har flere rammeverk (for eksempel **Django**, **Flask** og **FastAPI**) for å sette opp nettsider og nettsteder.
- Dataanalyse og databehandling: Python er veldig populært innen dataanalyse, med støtte fra pakker som **Numpy** og **Pandas**.
- Devops og skripting: Python kan effektivt binde sammen og automatisere andre prosesser.

Python er veldig tilpasningsdyktig, og avhengig av oppgaven kan språket se ut som et rett fram skriptespråk, et kraftig programmeringsspråk, Python kan fremheve objekt-orienteringen eller være et tilsynelatende funksjonelt programmeringsspråk. Alt dette er pakket inn i en relativt enkel ramme hvor det er mye fokus lesbarhet og at Pythonkode skal være forståelig og lett å endre.

Python er et dynamisk programmeringsspråk. Kort sagt betyr det at variabler kan endre type mens programmet kjører. Variabeltyper trenger heller ikke spesifiseres i koden. I stedet benyttes noe som kalles **Duck Typing**: *If it walks like a duck, and it quacks like a duck, it is a duck*. Det vil si at fokus er på egenskapene og metodene til objektene i stedet for typen. Samtidig har Python avansert støtte for eksplisitt typing ved behov.

Relatert til dette er Python tolket og ikke kompilert. Med andre ord leses koden først idet den kjøres, og det er faktisk mulig (men stort sett en dårlig ide) å endre Python-kode mens programmet kjøres.

Python beskrives ofte som **Batteries included**. Med dette menes at (nesten) alt man trenger for å skrive Pythonprogrammer er inkludert i grunninstallasjonen. En av styrkene til Python er at mange datastrukturer som `list` og `dict` er en del av kjernespråket. I tillegg er veldig mye funksjonalitet tilgjengelig i standardbiblioteket som alltid er installert. For eksempel inneholder dette biblioteket kode for behandling av filer (`pathlib` og `os`), og prosesser (`subprocess`), en fullverdig (men enkel) database (`sqlite3`) og avanserte tid og kalender-rutiner (`datetime` og `calendar`). Dokumentasjon for standardbiblioteket er tilgjengelig på docs.python.org.

1.1 Installer Python

På python.org vil du alltid finne siste versjon av Python. Det er likevel enklere å installere en Python-distribusjon kalt Anaconda. Anaconda inneholder flere nyttige tilleggspakker og programmer:

- Python

- De mest brukte Pythonpakkene
- Conda - Pakkebehandler
- Jupyter - Miljø for interaktiv programmering og utforskning
- Flere editorer tilpasset Python

Anaconda er tilgjengelig for både Windows, Mac og Linux.

- Last ned og installer **Anaconda**

*Python er allerede installert på Mac og Linux. Du bør **ikke** bruke disse preinstallerte versjonene av Python, fordi de brukes av systemet. Det gjør det vanskelig å oppdatere til nyere versjoner, og om noe skulle gå galt er det fare for at du ødelegger for hele operativsystemet. Bruk Anaconda også på Mac og Linux.*

Etter at du har installert Anaconda er programmet **Anaconda Navigator** tilgjengelig. Du kan bruke dette programmet til å starte forskjellige programmeringsverktøy, inkludert Pythonterminaler, Jupyter og editorer.

Anaconda er en ganske tung installasjon. Om du ønsker en lettere installasjon, hvor du bare installerer pakker og verktøy ved behov, kan du installere Miniconda i stedet.

1.2 Hei Python

Med Python installert er det på tide å se hvordan man bruker det. La oss skrive vårt første program:

- Start **Anaconda Navigator**
- Klikk på **Environments**
- Klikk på ▷ til høyre for **Base**
- Velg **Open Terminal**

Dette åpner en terminal hvor Anaconda Python er tilgjengelig.

- Skriv `python` og trykk **Enter**

Du kommer nå inn i et miljø kalt **Python REPL**. Her kan du skrive Pythonkommandoer og de vil bli utført umiddelbart.

- Skriv inn følgende kode:

[1]:
```
print("Hei Python!")
```

```
Hei Python!
```

1.3 Interaktiv Programmering i REPL

REPL står for **Read-Eval-Print-Loop**. Denne er veldig nyttig for å teste ut små kodesnutter interaktivt, men kan ikke brukes til å skrive fullstendige programmer som skal brukes flere ganger.

- Skriv inn følgende kode i REPL. Du må svare på spørsmålet etter at du skriver inn `input()`-linjen:

[2]:
```
navn = input("Hva heter du? ")
print(f"Hei {navn}")
```

```
Hva heter du?  Geir Arne

Hei Geir Arne
```

En forskjell mellom REPL og vanlige Pythonskript (som du vil se senere) er at verdien av uttrykk automatisk skrives til skjermen:

[3]:
```
navn
```

[3]:
```
'Geir Arne'
```

[4]:
```
(6048 + 1729) ** 2
```

[4]:
```
60481729
```

Dersom du skriver noe `python` ikke forstår vil du få en feilmelding. Disse kan bli ganske voldsomme. En grei angrepsmåte er å starte fra bunnen, og lese oppover til du ser noe du kjenner igjen

[5]:
```
print(f"Hei {navn})
```

```
  Cell In [5], line 1
    print(f"Hei {navn})
          ^
SyntaxError: unterminated string literal (detected at line 1)
```

1.4 En Forbedret REPL: `ipython`

`ipython` gir deg tilgang på en kraftigere REPL. `i` står for **interactive**. `ipython` har flere forbedringer:

- Kraftigere historikk
- Bedre **Tab**-komplettering
- Enklere tilgang til **hjelp**
- Spesielle (*magic*) kommandoer for enklere testing og utprøving av kode

Du kjører `ipython` på samme måte som `python`:

- Avslutt den kjørende REPL-sessjonen ved å skrive **`exit()`** (Du kan også bruke enten *Ctrl-Z* eller *Ctrl-D* avhengig av systemet ditt)
- Skriv **`ipython`** og trykk **Enter**
- Skriv inn samme kode som tidligere:

[6]:
```
navn = input("Hva heter du? ")
print(f"Hei {navn}")
```

```
Hva heter du?  Geir Arne

Hei Geir Arne
```

Dette er noen av triksene i `ipython`:

- Kommandoen `whos` gir deg en oversikt over definerte variabler
- Du kan bruke `_<tall>` for å referere til tidligere resultater. For eksempel refererer `_3` til resultatet av kommando nummer 3
- Avslutt en kommando med `?` for å se hjelpetekst. For eksempel `print?` viser informasjon om `print()`. `??` vil vise enda mer informasjon
- Bruk **Tab** for å se forslag på kommandoer. For eksempel `pr<Tab>` viser alle kommandoer som starter med `pr`

1.5 Programmer og Skript

Kode som skal gjenbrukes lagres vanligvis i en fil som kan kjøres av Python. Disse filene skal være rene tekstfiler med endelsen `.py`.

- Åpne **PyCharm** (eller en annen teksteditor). Du kan enten starte PyCharm gjennom **Anaconda Navigator** eller som et vanlig program.
- Åpne en **New File** (*Ctrl-N*) og **Save as ...** (*Ctrl-Shift-S*) `heisann.py`
- Skriv inn følgende kode. Avslutt med **Save** (*Ctrl-S*)

[7]:
```
navn = input("Hva heter du? ")
print(f"Hei {navn}")
```

```
Hva heter du?  Geir Arne
```

```
Hei Geir Arne
```

Den mest grunnleggende måten å kjøre et Pythonskript på er å bruke `python`-kommandoen fra terminalen

- Åpne en terminal fra **Anaconda Navigator** som tidligere. I Windows kan du også åpne **Anaconda Terminal** fra startmenyen. I Mac og Linux kan du bruke den innebygde terminalen
- Gå til samme katalog som der du lagret `heisann.py`. Bruk `cd <katalognavn>` for å flytte deg rundt. `cd ..` flytter deg en katalog høyere. I Windows kan du bruke `dir` for å se hvilke filer som ligger i gjeldende katalog. I Mac og Linux bruker du `ls`
- Skriv **`python heisann.py`** og trykk **Enter** for å kjøre programmet ditt

1.6 Jupyter

Jupyter er et miljø for programmering som kombinerer interaktiv utforskning, programmer og skript, og dokumentasjon i ett miljø. Jupyter har sin opprinnelse i `ipython`, men har vokst til å bli et eget miljø uavhengig av Python.

*Navnet **Jupyter** henspeiler på de tre programmeringsspråkene **Julia**, **Python** og **R***

Tradisjonelt har Jupyter vært tilgjengelig gjennom **Jupyter Notebooks**. Dette er arbeidsbøker som kombinerer kode med tekst og bilder i samme dokument. Nå er **JupyterLab** å foretrekke. JupyterLab kombinerer Notebooks sammen med et fullstendig utviklermiljø hvor du også har tilgang til en editor og en terminal. Du kan starte JupyterLab på to forskjellige måter:

- Åpne **Anaconda Navigator**. Fra **Home**-menyen: Klikk **Launch**-knappen under **JupyterLab**

eller

- Åpne en terminal som tidligere. Skriv `jupyter lab` og trykk **Enter**

JupyterLab vil åpne som en nettside i nettleseren din. Når JupyterLab starter viser den en **Launcher**. Fra denne kan du starte flere forskjellige dokumenter:

- **Notebook**: Dette er arbeidsboken som kalles **Jupyter Notebooks**, vi vil snart se nærmere på dem
- **Console**: Dette tilsvarer REPL som du har sett tidligere
- **Terminal**: Dette tilsvarer terminalen du allerede har jobbet med
- **Text File**: Editor for tekstfiler, inkludert Pythonkode
- **Markdown File**: Markdown gjør enkel formattering av tekst
- **Contextual Help**: Lett tilgjengelig hjelp for Pythonkommandoer

Du kan prøve å gjenskape noe av det du har gjort tidligere i Console eller Terminal. Deretter kan du starte en Jupyter Notebook:

- Klikk **Python 3** under **Notebook**

En Jupyter Notebook er en fil med endelse `.ipynb` (opprinnelig IPYthon NoteBook). Slike arbeidsbøker består av **celler**. Hver celle er enten en **code**-celle eller en **markdown**-celle. Du endrer typen på en celle i nedtrekksmenyen øverst på siden.

Både code- og markdown-celler **editeres** og **kjøres**. Trykk **Enter** for å begynne å editere en celle. Du kjører en celle ved å trykke **Shift-Enter** eller **Ctrl-Enter**. Førstnevnte hopper videre til neste celle. Du kan navigere mellom celler med piltastene.

- Skriv følgende kode i en **code**-celle. Du kan skrive begge linjene i samme celle.
- Trykk **Shift-Enter** for å kjøre cellen

[8]:
```
navn = input("Hva heter du? ")
print(f"Hei {navn}")
```

```
Hva heter du?  Geir Arne
```

```
Hei Geir Arne
```

Markdown-celler er nyttige for å dokumentere resultater og innsikt sammen med koden din. Markdown er et eget språk for enkel formattering av tekst. Her er noen enkle eksempler:

```
# Overskrift

## Underoverskrift

###### Veldig liten overskrift

**fet skrift**

_uthevet skrift_

`kodeskrift`
```

```
- liste
- med flere
- elementer
    - underelementer
    - underelementer
```

Du kan også inkludere matematiske uttrykk som $\sqrt{x^2 + y^2}$ ved å skrive LaTeX-uttrykk omgitt av $-tegn. For eksempel `$\sqrt{x^2 + y^2}$`.

Jupyter Notebooks bruker `ipython` i bakgrunnen. Du kan derfor bruke de samme funksjonene for historikk, tab-komplettering, hjelp osv.

- Skriv en markdown-celle hvor du tester ut forskjellige formatteringskommandoer.

Du kan også lage tabeller, inkludere bilder, lenke til nettsider, osv i Markdown. Det finnes mange nettsider som viser hvordan Markdown fungerer, for eksempel på guides.github.com.

JupyterLab har mange **hurtigtaster** for å jobbe effektivt. Klikk **Commands** (Ctrl-Shift-C) helt til venstre på skjermen for å se en oversikt.

Grunnleggende Python

September 22, 2022

1 Variabler, funksjoner og moduler

Vi vil i denne seksjonen se på de grunnleggende byggeklossene vi har i Python.

1.1 Variabler og typer

Variabler defineres på vanlig måte med `variabel = uttrykk`. Variabelnavn følger samme regler som de fleste andre språk:

- Variabelnavn kan bestå av bokstaver, tall og understrek, `_`. De kan ikke begynne med et tall.
- Det er forskjell på små og store bokstaver. Det vil si `variabel`, `Variabel` og `vaRIaBeL` er alle forskjellige.
- Pythonkode følger vanligvis konvensjonen om å bruke små bokstaver og understrek i variabelnavn, som for eksempel `counter` og `num_letters`.

Som nevnt er Python et dynamisk typet språk. I praksis betyr det at vi slipper å spesifisere typen til en variabel. Python finner typen automatisk.

[1]:
```
a = 3 + 4
a
```

[1]:
```
7
```

[2]:
```
type(a)
```

[2]:
```
int
```

[3]:
```
a * 3
```

[3]:
```
21
```

Videre medfører den dynamiske typingen at variabler kan endre type underveis i et program.

[4]:
```
a = "hei"
type(a)
```

[4]:
```
str
```

[5]:
```
a * 3
```

[5]:
```
'heiheihei'
```

[6]:
```
a = True
type(a)
```

[6]:
```
bool
```

[7]:
```
a * 3
```

[7]:
```
3
```

[8]:
```
a + "Neida"
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Cell In [8], line 1
----> 1 a + "Neida"

TypeError: unsupported operand type(s) for +: 'bool' and 'str'
```

Legg også merke til hvordan betydningen av operatorer som for eksempel `*` er avhengig av typen til variabelen.

De grunnleggende datatypene i Python er disse:

- **`bool`:** Enten `False` eller `True`.
- **`int`:** Heltall. Det er ingen øvre begrensning på størrelsen på disse.
- **`float`:** Flyttall. Disse er typisk implementert som 64-bits flyttall. Identifiseres av desimalskilletegnet, for eksempel er `3.0` en `float`, mens `3` er en `int`.
- **`complex`:** Komplekse tall håndteres direkte av Python, hvor den imaginære enheten skrives `j`. For eksempel er `1 + 2j` et komplekst tall.
- **`str`:** Tekststrenger, skrives med enkle eller doble fnutter. For eksempel er `'Hurra'` og `"I'm happy!"` tekststrenger.

Senere vil vi også møte mer avanserte datatyper som følger med Python (for eksempel `tuple`, `list` og `dict`). Det er også lett å lage egne datatyper ved å definere dem som klasser. Dette blir ikke dekt i denne introduksjonen.

Så langt det gir mening kan man konvertere mellom datatypene ved å bruke `bool`, `int`, `float`, `complex` og `str` som funksjoner:

[9]:
```
str(3)
```

[9]:
```
'3'
```

[10]:
```
float(10)
```

[10]:
```
10.0
```

[11]:
```
bool("Hei")
```

[11]:
```
True
```

[12]:
```
bool("")
```

[12]:
```
False
```

[13]:
```
int(3.88)
```

[13]:
```
3
```

[14]:
```
float("3.14")
```

[14]:
```
3.14
```

[15]:
```
int("Neida")
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
Cell In [15], line 1
----> 1 int("Neida")

ValueError: invalid literal for int() with base 10: 'Neida'
```

1.2 Formattering av tekst

Som nevnt skrives tekst med enten enkle eller doble fnutter. I tillegg finnes det spesielle **formatterte** tekststrenger, som du kan kjenne igjen ved at de starter med `f` foran åpningsfnutten. Disse kalles også f-strenger. Formatterte strenger kan inkluderte variabler og andre uttrykk direkte i tekststrengen, så lenge de er markert med krøllparanteser:

[16]:
```
name = "Guido"
language = "Python"
year = 1989

f"{name} invented {language.upper()} {2022 - year} years ago"
```

[16]:
```
'Guido invented PYTHON 33 years ago'
```

Tradisjonelt har Python hatt flere måter å formattere tekst på. Du vil muligens se alle disse, men f-strenger blir mer og mer populært.

- Manuell sammensetting med `+`
  - `name + " invented " + language.upper() + " " + str(2019 - year) + " years ago"`
- `printf`-inspirert med `%`

  - `"%s invented %s %d years ago" % (name, language.upper(), 2019 - year)`
- Kraftigere formattering med `.format()`
  - `"{} invented {} {} years ago".format(name, language.upper(), 2019 - year)`
- f-strenger
  - `f"{name} invented {language.upper()} {2019 - year} years ago"`

Både `.format()` og f-strenger støtter et eget minispråk for mer kontroll over formatteringen. Dette består av spesielle koder du skriver etter `:` inne i krøllparantesene.

[17]:
```
pi = 3.14159

print("Navn       Språk      År         Pi\n" + "-" * 35)
print(f"{name:<10s} {language:10} {year:>4} {pi:8.3f}")
```

```
Navn       Språk      År         Pi
-----------------------------------
Guido      Python     1989    3.142
```

1.3 Funksjoner

Funksjoner defineres med nøkkelordet `def`. Her er et enkelt eksempel:

[18]:
```
def say_hello(greeting, name):
    print(f"{greeting} {name}")


say_hello("Hei", "alle sammen")
say_hello("Yo!", "Geir Arne!")
```

```
Hei alle sammen
Yo! Geir Arne!
```

Ofte vil du returnere en verdi fra funksjonene. Dette gjøres med `return` på denne måten:

[19]:
```
def create_greeting(greeting, name):
    return f"{greeting} {name}"


create_greeting("Howdy", "Boss")
```

[19]:
```
'Howdy Boss'
```

Alle funksjoner i Python returnerer en verdi. Om du ikke spesifiserer verdien med `return` returneres verdien `None`.

1.4 Bruk av indentering

En ting du raskt legger merke til er at Python bruker veldig få spesialtegn for å markere for eksempel hva som er en funksjondefinisjon (og tilsvarende utstrekningen av en løkke eller test og lignende).

I stedet markerer indenteringen av koden dette.

I eksempelet over består funksjonen `say_hello` kun av den ene linjen `print(hilsen + ' ' + navn)` siden dette er den eneste linjen som er rykket inn under funksjonsdefinisjonen. Legg også merke til `:`'et som avslutter `def`-linjen. Et slikt kolon brukes overalt før en indentering starter.

Indenteringer i Python skal være 4 mellomrom. Ikke bruk tab (de fleste teksteditorer vil automatisk gjøre om en tab til 4 mellomrom i Pythonkode).

1.5 Import av moduler

Mye av funksjonaliteten til Python ligger i standardbiblioteket. Dette er funksjonalitet som alltid er tilgjengelig, men som ikke lastes inn i programmet ditt uten at du ber om det. For å ta dette i bruk bruker du `import`.

[20]:
```
import math

math.pi
```

[20]:
```
3.141592653589793
```

[21]:
```
math.cos(math.pi)
```

[21]:
```
-1.0
```

En programpakke slik som `math` kalles en **modul**. Dersom du bare er interessert i et par elementer fra en modul kan disse lastes eksplisitt.

[22]:
```
from math import pi, cos

pi
```

[22]:
```
3.141592653589793
```

[23]:
```
cos(pi)
```

[23]:
```
-1.0
```

Moduler brukes aktivt for å strukturere **navnerommene** (*namespaces*) i Python, siden hver modul definerer sitt eget navnerom hvor det ikke er fare for kollisjoner med annen kode.

Det er mulig å endre navnet på et element idet det importeres. For noen pakker er dette vanlig fordi det gjør koden mer konsis.

[24]:
```
import numpy as np

np.cos(np.linspace(0, np.pi, 4))
```

[24]:
```
array([ 1. ,  0.5, -0.5, -1. ])
```

Det er også mulig å bruke `import as` for å unngå navnekollisjoner.

[25]:
```
from numpy import sqrt as sqrt1
from math import sqrt as sqrt2

sqrt1(1729)
```

[25]:
```
41.58124577258358
```

[26]:
```
sqrt2(1729)
```

[26]:
```
41.58124577258358
```

Pakken `numpy` brukes til matematikk og numeriske beregninger. Den er ikke tilgjengelig i standardbiblioteket og må derfor installeres eksplisitt.

Både `math.sqrt` og `numpy.sqrt` jobber med 64-bits flyttall. Det er noen begrensninger i nøyaktigheten i slike flyttallsoperasjoner. En bedre approksimasjon til kvadratroten av 1729 vil være 41.581245772583581890280209 19. Pakken `decimal` kan hjelpe mot problemer med flyttall.

1.6 En fil er en modul

Det er enkelt å lage sine egne moduler i Python, dersom man for eksempel vil gjenbruke kode i forskjellige programmer. Alle `.py`-filer er automatisk moduler som kan importeres. Lagre `say_hello`-funksjonen fra tidligere i filen `hello.py`:

[27]:
```
%%writefile hello.py
def say_hello(greeting, name):
    print(f"{greeting} {name}")

say_hello('Hei', 'alle sammen')
say_hello('Yo!', 'Geir Arne!')
```

```
Writing hello.py
```

Du kan nå importere `hello` på samme måte som alle andre Pythonmoduler:

[28]:
```
import hello
```

```
Hei alle sammen
Yo! Geir Arne!
```

[29]:
```
hello.say_hello("Bond", "James Bond")
```

```
Bond James Bond
```

Legg merke til at filen kjøres som et skript under importen. I eksempelet over blir de to kallene til `say_hello` utført idet modulen `hello` importeres.

1.7 Spesielle navn

Python benytter seg av en del **spesielle navn** internt. Disse har alle navn omkranset av to understreker, for eksempel `__name__`. Siden **double underscore** er litt slitsomt å si eller skrive i

lengden omtales slike navn også som **dunder**s.

En nyttig (og veldig vanlig) bruk av disse spesielle navnene i kode er for å skille mellom når en Python-fil importeres og når den kjører som et eget skript. Dette gjøres ved å sjekke `__name__` som settes til modulnavnet om filen importeres, men bare til `__main__` om filen kjøres som et skript.

[30]:
```
%%writefile hello.py
def say_hello(greeting, name):
    print(f"{greeting} {name}")

if __name__ == '__main__':
    say_hello('Hei', 'alle sammen')
    say_hello('Yo!', 'Geir Arne!')
```

```
Overwriting hello.py
```

Andre nyttige spesielle navn inkluderer `__doc__` som inneholder dokumentasjon for en modul, klasse eller funksjon. Det finnes også mange spesielle metodenavn som brukes ved implementering av egne typer og objekter. For eksempel defineres et objekts konstruktor i `__init__()`, mens `__repr__()` bestemmer hvordan et objekt representeres som en streng.

2 Tester og enkle løkker

Python støtter de vanligste konstruksjonene for tester og løkker.

2.1 `if`-tester

Vi så såvidt en `if`-test i koden ovenfor. Slike tester er ganske rett fram i Python. Til forskjell fra mange andre språk brukes vanligvis ikke parantes rundt uttrykket det testes på.

[31]:
```
number = 14

if number < 20:
    print("Tallet er mindre enn 20")
```

```
Tallet er mindre enn 20
```

For å kombinere tester brukes `not`, `and` og `or`. Disse operatorene er kortsluttende (*short-circuiting*) slik at det ikke gjøres mer arbeid enn nødvendig for å vite resultatet av testen. For eksempel, om man har en variabel som kan være `None` eller et tall er det vanskelig å direkte teste på tallet.

[32]:
```
x = None
x < 3
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Cell In [32], line 2
      1 x = None
----> 2 x < 3
```

```
TypeError: '<' not supported between instances of 'NoneType' and 'int'
```

[33]:
```
x is not None and x < 3
```

[33]:
```
False
```

For å teste om et tall faller i et spesielt intervall kan man kombinere to tester helt uten `and`.

[34]:
```
y = 1992
1970 <= y < 2000
```

[34]:
```
True
```

2.2 Flere valg

For mer kompliserte tester brukes også `else` og `elif` (som en konsis måte å skrive *else if* på).

[35]:
```
def ordinal(num):
    if 10 <= num % 100 <= 20:
        suffix = "th"
    elif num % 10 == 1:
        suffix = "st"
    elif num % 10 == 2:
        suffix = "nd"
    elif num % 10 == 3:
        suffix = "rd"
    else:
        suffix = "th"

    return str(num) + suffix


ordinal(23)
```

[35]:
```
'23rd'
```

Det finnes ingen `switch` i Python. I stedet kan man bruke `if elif else` som i eksempelet ovenfor. Senere skal vi se en annen versjon av `switch`.

2.3 `while`-løkker

Det finnes to typer løkker i Python, `while`-løkker og `for`-løkker. Som vanlig brukes `while`-løkker når man vil fortsette løkken til en gitt betingelse ikke lenger gjelder uten at man på forhånd nødvendigvis vet hvor mange runder gjennom løkken man behøver. Syntaksen er omtrent som for `if`-løkker.

[36]:
```python
import random

number = random.randint(1, 1000)
guess = None
print("Mitt hemmelige number er mellom 1 og 1000")

while guess != number:
    guess = int(input("Gjett mitt hemmelige tall: "))
    if number < guess:
        print("Mitt tall er lavere")
    elif number > guess:
        print(f"Mitt tall er over {guess}")
    else:
        print(f"Riktig! Mitt tall var {number}")
```

```
Mitt hemmelige number er mellom 1 og 1000
Gjett mitt hemmelige tall:  500
Mitt tall er over 500
Gjett mitt hemmelige tall:  777
Mitt tall er lavere
Gjett mitt hemmelige tall:  600
Mitt tall er lavere
Gjett mitt hemmelige tall:  555
Mitt tall er lavere
Gjett mitt hemmelige tall:  532
Mitt tall er over 532
Gjett mitt hemmelige tall:  543
Mitt tall er over 543
Gjett mitt hemmelige tall:  550
Mitt tall er lavere
Gjett mitt hemmelige tall:  547
Mitt tall er over 547
Gjett mitt hemmelige tall:  548
Mitt tall er over 548
Gjett mitt hemmelige tall:  549
Riktig! Mitt tall var 549
```

2.4 Mer kontroll på løkkene

For å ha enda bedre kontroll på hvordan en `while`-løkke oppfører seg kan du bruke `break` og `continue`. Førstnevnte avslutter hele løkka, mens `continue` hopper til begynnelsen av løkka igjen. For eksempel vil det følgende eksempelet skrive ut alle oddetallene mellom 1 og 20 (det finnes mye enklere og bedre måter å gjøre akkurat dette på).

[37]:
```
number = 0

while True:
    number += 1
    if number % 2 == 0:
        continue
    if number > 20:
        break
    print(number)
```

```
1
3
5
7
9
11
13
15
17
19
```

Legg merke til at `while True:` starter en evig løkke. For at programmet skal komme ut av en slik løkke trenger vi `break` eller `return`.

3 Datastrukturer, iteratorer og flere løkker

Et viktig konsept i Python som gjør språket morsomt å bruke er iteratorer. Vi kommer tilbake til disse etter å sett litt på lister og tekststrenger.

3.1 Lister

Datastrukturen **`list`** er innebygd i Python, og står for stort sett den samme funksjonaliteten som en **array** i mange andre språk. En liste defineres vanligvis ved hjelp av hakeparanteser, `[ ... ]`, kan inneholde vilkårlige datatyper og er 0-indeksert.

[38]:
```
ting = [True, 3.14, "Heisann", 10, 12]
ting[2]
```

[38]:
```
'Heisann'
```

[39]:
```
len(ting)
```

[39]:
```
5
```

[40]:
```
ting.append("En ny ting")
ting
```

[40]:
```
[True, 3.14, 'Heisann', 10, 12, 'En ny ting']
```

[41]:
```
ting[1:4]
```

[41]:
```
[3.14, 'Heisann', 10]
```

[42]:
```
ting[2:3]
```

[42]:
```
['Heisann']
```

Som eksemplene viser kan man hente ting fra listen ved å sette indekser i hakeparenteser. Man kan også bruke en **slice** for å hente ut en delliste. En slice lager et intervall av indekser ved å putte kolon mellom dem. Man kan la indeksene være blanke, de vil i så fall bli tolket som henholdsvis første og siste element. Legg spesielt merke til at i det siste eksempelet bruker vi `ting[2:3]` for å hente ut en delliste med bare ett element. Dette er forskjellig fra å hente ut elementet selv med `ting[2]`.

3.2 En tekststreng ligner en liste

På mange måter oppfører en **tekststreng** seg veldig likt en liste. Vi kan bruke de samme operasjonene for å hente ut enkeltbokstaver eller delstrenger.

[43]:
```
tekst = "Tekststreng"
tekst[2]
```

[43]:
```
'k'
```

[44]:
```
len(tekst)
```

[44]:
```
11
```

[45]:
```
tekst[1:4]
```

[45]:
```
'eks'
```

[46]:
```
tekst[:7]
```

[46]:
```
'Tekstst'
```

[47]:
```
tekst[::2]
```

[47]:
```
'Tktteg'
```

I det siste eksempelet har vi også spesifisert steglengde i slicen. Ved å skrive et andre kolon kan man spesifisere `start:slutt:steg`. Det siste eksempelet skriver ut annenhver bokstav fra tekststrengen. En Pythonsk måte å reversere en streng på er enkelt og greit `tekst[::-1]`.

3.3 Iteratorer

Både `list` og `str` er eksempler på en struktur som kalles **sequence** og som støtter indeksering og slicing. Disse er igjen del av en større klasse objekter som kalles iteratorer.

En **iterator** er et objekt som kan returnere elementene sine en etter en. Disse finnes overalt i Python, og i tillegg til lister og strenger er `set`, `dict` og filer eksempler på iteratorer (mer om disse senere).

3.4 `for`-løkker

I Python fungerer `for`-løkker litt annerledes enn i mange andre språk, da `for`-løkkene baserer seg på iteratorer. Slike løkker defineres ved å skrive `for element in iterator:`, for eksempel som dette:

[48]:
```
words = ["Norge", "Oslo", "November", "Python"]

for word in words:
    print(f"Ordet {word} har {len(word)} bokstaver")
```

```
Ordet Norge har 5 bokstaver
Ordet Oslo har 4 bokstaver
Ordet November har 8 bokstaver
Ordet Python har 6 bokstaver
```

Denne måten å skrive `for`-løkker på kan føles litt uvant i starten, men siden man stort sett slipper å tenke på indekser er det mange vanlige *off-by-one*-bugs som aldri blir skrevet i Python.

3.5 Avansert iterasjon

En vanlig bruk av indekser i andre språk er å kombinere to lister. Dette kan man gjøre med `zip` i Python (tenk glidelås).

[49]:
```
places = ["Andenes", "Hamar", "Stavanger"]
regions = ["Nordland", "Hedmark", "Rogaland"]

for place, region in zip(places, regions):
    print(f"{place} ligger i {region}")
```

```
Andenes ligger i Nordland
Hamar ligger i Hedmark
Stavanger ligger i Rogaland
```

Dersom man behøver å generere en liste med tall kan man bruke `range`. For eksempel gir `range(1, 20, 2)` alle oddetallene mellom 1 og 20. Dersom man trenger indeksene kan man bruke `enumerate`.

[50]:
```
cities = ["Oslo", "Bergen", "Trondheim", "Stavanger", "Drammen"]

for index, city in enumerate(cities, start=1):
    print(f"{city} er nummer {index}")
```

```
Oslo er nummer 1
Bergen er nummer 2
Trondheim er nummer 3
Stavanger er nummer 4
Drammen er nummer 5
```

3.6 List comprehensions

Enkle `for`-løkker kan ofte formuleres som en **list comprehension** i stedet. En list comprehension omformer en liste til en annen, og har formen `[expression for element in iterator]`.

[51]:
```
numbers = range(1, 16)
[x for x in numbers]
```

[51]:
```
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15]
```

[52]:
```
[x**2 for x in numbers]
```

[52]:
```
[1, 4, 9, 16, 25, 36, 49, 64, 81, 100, 121, 144, 169, 196, 225]
```

[53]:
```
[x**2 for x in numbers if x % 2]
```

[53]:
```
[1, 9, 25, 49, 81, 121, 169, 225]
```

I det siste eksempelet ser vi at vi også kan filtrere hvilke elementer som skal være med i den nye listen. En list comprehension kan alltid skrives om som en `for`-løkke. For eksempel er koden `squares = [x**2 for x in numbers]` omtrent ekvivalent med følgende:

[54]:
```
numbers = range(1, 16)
squares = list()
for x in numbers:
    squares.append(x**2)

squares
```

[54]:
```
[1, 4, 9, 16, 25, 36, 49, 64, 81, 100, 121, 144, 169, 196, 225]
```

Et bra kriterie for å velge når man skal bruke list comprehensions er lesbarhet.

3.7 Muterbare og ikke-muterbare objekter

Et annet sentralt konsept i Python er muterbarhet. Et muterbart objekt er et objekt som kan endre verdi etter at det er laget. For eksempel er en `list` muterbar, mens en `str` ikke er det.

[55]:
```
byer = ["Oslo", "Bergen", "Trondheim"]
id(byer)
```

[55]:
```
139703811734976
```

[56]:
```
byer[2] = "Stavanger"
byer
```

[56]:
```
['Oslo', 'Bergen', 'Stavanger']
```

[57]:
```
byer.insert(2, "Trondheim")
byer
```

[57]:
```
['Oslo', 'Bergen', 'Trondheim', 'Stavanger']
```

[58]:
```
id(byer)
```

[58]:
```
139703811734976
```

Funksjonen `id` returnerer en unik identifikator for et objekt (typisk minneadressen til objektet). I eksempelet over ser vi at objektet som `byer` refererer til er det samme gjennom hele koden. Det er objektet selv som har endret verdi.

[59]:
```
tekst = "Vaskdrag"
id(tekst)
```

[59]:
```
139703811667760
```

[60]:
```
tekst[3] = "s"
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Cell In [60], line 1
----> 1 tekst[3] = "s"

TypeError: 'str' object does not support item assignment
```

[61]:
```
tekst = "Vassdrag"
id(tekst)
```

[61]:
```
139703811410800
```

Tekststrenger er ikke muterbare, som vi oppdager dersom vi prøver å forandre bare en bokstav i strengen. For å endre verdien av `tekst` må vi i stedet tilordne et helt nytt strengobjekt til variabelen. Ved hjelp av `id` bekrefter vi at `tekst` peker til et nytt objekt.

Muterbarhet har noen subtile konsekvenser. I utgangspunktet opererer Python som hovedregel på pekere til objekter (*pass-by-reference*), ikke på verdiene til objektene (*pass-by-value*). De følgende

eksemplene illustrerer igjen forskjellen på muterbare og ikke-muterbare objekter.

[62]:
```
byer = ["Oslo", "Narvik", "Trondheim"]
byer2 = byer
byer2[1] = "Bergen"
byer
```

[62]:
```
['Oslo', 'Bergen', 'Trondheim']
```

Siden `byer` og `byer2` peker til samme objekt, og det objektet er en muterbar `list`, endres også verdien av `byer` når `byer2` oppdateres.

[63]:
```
tekst = "Perl"
tekst2 = tekst
id(tekst), id(tekst2)
```

[63]:
```
(139704592783728, 139704592783728)
```

[64]:
```
tekst2 = "Python"
tekst
```

[64]:
```
'Perl'
```

[65]:
```
id(tekst), id(tekst2)
```

[65]:
```
(139704592783728, 139704618084464)
```

Her peker i utgangspunktet `tekst` og `tekst2` til samme objekt. Men siden dette strengobjektet ikke er muterbart endres pekeren fra `tekst2` når denne oppdateres. Dermed endres ikke verdien av `tekst`.

Den vanligste måten å faktisk kopiere elementene i en `list` på er ved hjelp av slicing eller `copy()`-metoden.

[66]:
```
byer = ["Oslo", "Narvik", "Trondheim"]
byer2 = byer[:]
byer3 = byer.copy()
id(byer), id(byer2), id(byer3)
```

[66]:
```
(139703811934016, 139703811410816, 139703811922944)
```

[67]:
```
byer2[1] = "Bergen"
byer
```

[67]:
```
['Oslo', 'Narvik', 'Trondheim']
```

[68]:
```
[id(by) for by in byer]
```

[68]:
```
[139703811657328, 139703811829552, 139703811656048]
```

[69]:
```
[id(by) for by in byer2]
```

[69]:
```
[139703811657328, 139703811657904, 139703811656048]
```

Merk at dette er en såkalt grunn kopi *shallow copy*. Det vil si at elementene i den nye listen peker til elementene i den gamle listen. Det vil si at i eksempelet over er `id(byer2[0])` lik `id(byer[0])`. Den praktiske konsekvensen av dette er at om du kopierer nestede lister, vil de innerste listene fortsatt peke til de samme objektene og kan derfor påvirke hverandre. I standardbiblioteket `copy` finnes funksjonen `deepcopy()` som kan utføre en komplett kopi av nestede strukturer.

3.8 Flere datastrukturer

I tillegg til `list` er flere avanserte datastrukturer standard i Python. En **`tuple`** ligner mye på en `list`. Hovedforskjellen er at en `tuple` ikke er muterbar. Konsekvensen av dette er at en `tuple` ofte brukes for å beskrive informasjon som passer i en *database record*. For eksempel `person = ('Geir Arne', 'Hjelle', 'Oslo')` eller `koordinater = (-3.2, 2.9)`.

Merk at en `tuple` defineres av kommaet `,`, men det er vanlig å stort sett bruke paranteser i tillegg. Dersom det er behov for et `tuple` med ett element må derfor fortsatt kommaet skrives: `tall = (12, )`.

Python støtter **tuple unpacking** som betyr at man kan pakke ut et `tuple` og tilordne verdiene til forskjellige variable. Dette gjør det lett å flippe variable eller simulere funksjoner med flere returverdier.

[70]:
```
fornavn, etternavn, by = ("Geir Arne", "Hjelle", "Oslo")
etternavn
```

[70]:
```
'Hjelle'
```

[71]:
```
fornavn, etternavn = etternavn, fornavn
etternavn
```

[71]:
```
'Geir Arne'
```

[72]:
```
divmod(23, 3)
```

[72]:
```
(7, 2)
```

[73]:
```
kvotient, rest = divmod(23, 3)
rest
```

[73]:
```
2
```

En `dict` eller **dictionary** er Pythons versjon av **hashmap** eller **associative arrays** som finnes i mange andre språk. Disse er sterkt optimaliserte og kraftige datastrukturer, som også brukes mye internt i Python. En `dict` er en samling av `key: value`-par. Den defineres ved hjelp av krøllparanteser, som i eksempelet under.

[74]:
```
fylker = {"Andenes": "Nordland", "Hamar": "Hedmark"}
fylker["Hamar"]
```

[74]:
```
'Hedmark'
```

[75]:
```
fylker["Bergen"] = "Hordaland"
fylker
```

[75]:
```
{'Andenes': 'Nordland', 'Hamar': 'Hedmark', 'Bergen': 'Hordaland'}
```

Som eksempelet over hinter til er også `dict` en muterbar datastruktur. Nøklene må være unike og ikke-muterbare, mens verdiene kan være hvilke som helst Python-objekter inkludert nye `list`er eller `dict`er.

Den siste datastrukturen vi vil nevne her er **`set`**. Et `set` er en ikke-ordnet samling av objekter. Elementene i et `set` må være ikke-muterbare og duplikater vil bli ignorert. Vanlige mengdeoperasjoner som union (`|`), snitt (`&`) og komplement (`-`) er tilgjengelig. Rekkefølgen på elementene i et `set` tilfeldig.

[76]:
```
hedmark = {"Hamar", "Elverum", "Os"}
hordaland = {"Bergen", "Os"}

hedmark | hordaland
```

[76]:
```
{'Bergen', 'Elverum', 'Hamar', 'Os'}
```

[77]:
```
hedmark & hordaland
```

[77]:
```
{'Os'}
```

[78]:
```
hedmark - hordaland
```

[78]:
```
{'Elverum', 'Hamar'}
```

En vanlig måte å bruke `set` på er å fjerne duplikater fra en liste.

[79]:
```
counties = ["Hamar", "Elverum", "Os", "Bergen", "Os"]
list(set(counties))
```

[79]:
```
['Hamar', 'Os', 'Bergen', 'Elverum']
```

Siden `set` representerer en ikke-ordnet mengde vil ikke nødvendigvis rekkefølgen i listen bevares. Dersom vi må bevare rekkefølgen i listen mens vi fjerner duplikatene kan vi for eksempel bruke en list comprehension sammen med et `set` på denne måten:

[80]:
```
counties = ["Hamar", "Elverum", "Os", "Bergen", "Os"]
seen = set()
[x for x in counties if x not in seen and not seen.add(x)]
```

[80]:
```
['Hamar', 'Elverum', 'Os', 'Bergen']
```

Tabellen nedenfor oppsummerer disse mer avanserte datastrukturene.

- **`tuple`:** ikke-muterbare, ordnede følger
  - for eksempel `posisjon = (1202462.712, 252734.419, 6237766.073)`
- **`list`:** muterbare, ordnede følger
  - for eksempel `byer = ['Oslo', 'Bergen', 'Trondheim', 'Stavanger']`
- **`dict`:** muterbare, ikke-ordnede avbildninger
  - for eksempel `kommunenr = {'0716': 'Re', '0441': 'Os', '1243': 'Os'}`
- **`set`:** muterbare, ikke-ordnede mengder (duplikater blir ignorert)
  - for eksempel `fornavn = {'Geir', 'Arne', 'Hans', 'Geir', 'Nils'}`

3.9 Funksjoner er også objekter

I Python er funksjoner objekter på lik linje med alle andre objekter. En litt morsom anvendelse av dette, som overtar for en del bruksområder til `switch` (som ikke eksisterer i Python), er å kalle forskjellige funksjoner basert på et gitt nøkkelord.

[81]:
```
def add(x, y):
    return x + y


def multiply(x, y):
    return x * y


def calculate(op, x, y):
    return OPERATIONS[op](x, y)


OPERATIONS = {
    "pluss": add,
    "gange": multiply,
}

calculate("gange", 3, 6)
```

[81]:
```
18
```

Legg merke til at i `calculate`-funksjonen slår vi først opp i `OPERATIONS` og deretter kaller vi funksjonen vi finner med parametrene `x` og `y`.

4 Filstier og filer

Python har god støtte for å arbeide med filstier og filer, og leser lett både tekst- og binærfiler. I standardbiblioteket finnes også moduler for å jobbe direkte med zippede filer.

4.1 Filstier

Biblioteket `pathlib` er veldig nyttig når du jobber med filer. Det passer blant annet på at filstier kan behandles på en uniform måte uavhengig av operativsystemet du kjører på. Det er veldig nyttig om du for eksempel utvikler et skript på Windows, men skal kjøre det på en Linuxserver.

[82]:
```
import pathlib

path = pathlib.Path("hello.py")
path
```

[82]:
```
PosixPath('hello.py')
```

[83]:
```
path.exists()
```

[83]:
```
True
```

[84]:
```
full_path = path.resolve()
full_path
```

[84]:
```
PosixPath('/home/gahjelle/Dropbox/presentasjoner/20220922_python_nve/intro/hello
.py')
```

[85]:
```
full_path.parent.name, full_path.suffix, full_path.stem
```

[85]:
```
('intro', '.py', 'hello')
```

[86]:
```
full_path.with_name("goodbye.py")
```

[86]:
```
PosixPath('/home/gahjelle/Dropbox/presentasjoner/20220922_python_nve/intro/goodb
ye.py')
```

Du kan opprette `Path`-objekter fra hele filstier, eller bruke `/` for å sette sammen `Path`-objekter. På Windows bør du bruke en `r` foran filsti-strengen for å unngå problemer med `\`.

[87]:
```
code_dir = pathlib.Path.home() / "python"
code_dir.mkdir(parents=True, exist_ok=True)

code_file = code_dir / "cool_program.py"
code_file.touch()
```

4.2 Åpning og lesing av filer

Python åpner filer gjennom den innebygde funksjonen `open()`. I all enkelhet trenger man bare å spesifisere filstien og om filen skal åpnes i lese- eller skrivemodus.

[88]:
```
with open("hello.py") as fid:
    print(fid.readlines())
```

```
['def say_hello(greeting, name):\n', '    print(f"{greeting} {name}")\n', '\n',
"if __name__ == '__main__':\n", "    say_hello('Hei', 'alle sammen')\n", "
say_hello('Yo!', 'Geir Arne!')\n"]
```

Du kan også gjøre det samme ved å kalle `open()` direkte på et filstiobjekt:

[89]:
```
with path.open() as fid:
    print(fid.readlines())
```

```
['def say_hello(greeting, name):\n', '    print(f"{greeting} {name}")\n', '\n',
"if __name__ == '__main__':\n", "    say_hello('Hei', 'alle sammen')\n", "
say_hello('Yo!', 'Geir Arne!')\n"]
```

Videre kan du kalle `read_text()` eller `read_binary()` på filstiobjekter, som leser inn hele filen på en gang:

[90]:
```
path.read_text()
```

[90]:
```
'def say_hello(greeting, name):\n    print(f"{greeting} {name}")\n\nif __name__
== \'__main__\':\n    say_hello(\'Hei\', \'alle sammen\')\n
say_hello(\'Yo!\', \'Geir Arne!\')\n'
```

Uttrykket `with` starter en **context manager**. Slike kan gjøre mange nyttige ting for oss. I sammenheng med fillesing håndterer den lukking av filen, samt at ressurser blir skikkelig ryddet opp dersom feil oppstår under filhåndteringen.

[91]:
```
%%writefile cat.py
import sys

for filename in sys.argv[1:]:
    with open(filename) as fid:
        for line in fid:
            print(line.rstrip())
```

```
Writing cat.py
```

[92]:
```
!python cat.py hello.py cat.py
```

```
def say_hello(greeting, name):
    print(f"{greeting} {name}")

if __name__ == '__main__':
    say_hello('Hei', 'alle sammen')
    say_hello('Yo!', 'Geir Arne!')
import sys

for filename in sys.argv[1:]:
    with open(filename) as fid:
        for line in fid:
            print(line.rstrip())
```

Med `with` blir filen automatisk lukket når kodeblokken indentert under `with` avsluttes.

Som eksempelet ovenfor viser er filpekere (som `fid`) også iteratorer som returnerer innholdet i filen, linje for linje. Alternativt, kan hele filen leses inn i en liste med `readlines()` som i det første eksempelet.

4.3 Skriving av filer

For å skrive til filer bruker man også `open()`, men nå må man spesifisere `mode='w'` (*write*) eller `mode='a'` (*append*) for at filen skal åpnes i skrivemodus. Om man bruker `w` vil filen overskrives om den allerede finnes, mens `a` gjør at man legger til linjer i en allerede eksisterende fil.

Selve innholdet man vil skrive sender man til funksjonen `write()` som man kaller på filpekeren (som `fid` eller `fil_out` nedenfor). Det følgende eksempelet skriver ut den lille gangetabellen til filen `gangetabell.txt`.

[93]:
```
with open("gangetabell.txt", mode="w") as fid:
    for number1 in range(1, 11):
        for number2 in range(1, 11):
            fid.write(f" {number1 * number2:3d}")
        fid.write("\n")
```

[94]:
```
!python cat.py gangetabell.txt
```

```
  1   2   3   4   5   6   7   8   9  10
  2   4   6   8  10  12  14  16  18  20
  3   6   9  12  15  18  21  24  27  30
  4   8  12  16  20  24  28  32  36  40
  5  10  15  20  25  30  35  40  45  50
  6  12  18  24  30  36  42  48  54  60
  7  14  21  28  35  42  49  56  63  70
  8  16  24  32  40  48  56  64  72  80
  9  18  27  36  45  54  63  72  81  90
 10  20  30  40  50  60  70  80  90 100
```

En typisk oppgave er å lese gjennom en fil, gjøre små endringer på den, og deretter skrive den ut igjen. Avhengig av hvor store endringene er kan dette gjøres på forskjellige måter. Det følgende eksempelet viser en vanlig mønster hvor man leser og skriver ut en og en linje, i dette tilfellet skriver vi ut linjenummer først på linjen. Legg spesielt merke til at vi kombinerer to `open`-kommandoer i den samme `with`-blokken. Vi bruker `start=1` fordi Python vanligvis begynner å telle på 0, mens vi vil at første linje skal være nummer 1.

[95]:
```
%%writefile line_numbers.py
import pathlib

path_in = pathlib.Path('line_numbers.py')
path_out = pathlib.Path('linjenummer.txt')

with path_in.open() as f_in, path_out.open(mode='w') as f_out:
    for number, line in enumerate(f_in, start=1):
```

```
        f_out.write(f'{number:04d} {line}')
```

```
Writing line_numbers.py
```

De to siste eksemplene viser flere eksempler på formaterte tekststrenger i Python. Om man kjører programmet over får man følgende resultat:

[96]:
```
!python line_numbers.py
```

[97]:
```
!python cat.py linjenummer.txt
```

```
0001 import pathlib
0002
0003 path_in = pathlib.Path('line_numbers.py')
0004 path_out = pathlib.Path('linjenummer.txt')
0005
0006 with path_in.open() as f_in, path_out.open(mode='w') as f_out:
0007     for number, line in enumerate(f_in, start=1):
0008         f_out.write(f'{number:04d} {line}')
```

Tilsvarende `read_text()` og `read_bytes()` kan du også bruke `write_text()` og `write_bytes()` for å skrive til fil.

4.4 Et mer realistisk eksempel

Dessverre er ekte data som regel litt trøblete å jobbe med. For å se eksempler på utfordringer som kan dukke opp vil vi nå se på et mer realistisk eksempel. Vi ønsker å laste ned og lese en fil som inneholder mye informasjon om alle verdens land. Deretter vil vi plukke ut informasjon fra denne filen og lage en quiz om hovedsteder.

OpenGeocode pleide å lage datasettet **Countries of the World (COW)**. Det virker ikke å være like tilgjengelig lengre, men på github.com/Delanir/LatitudePlay finnes en arkivert kopi. Last ned `Countries of the World - CSV`-filen til datamaskinen din. Dette er en tekstfil som heter `cow.txt`. Pass på at den ligger i samme katalog som du lagrer pythonprogrammene i. Åpne filen og se på den. Det er mye informasjon her!

La oss først se om vi klarer å lese filen i Python. Lag et nytt program:

[99]:
```
with open("cow.txt") as f_in:
    for line in f_in:
        print(line)
```

```
# Countries of the World (COW)

# Created by OpenGeoCode.Org, Submitted into the Public Domain August 10, 2012
(version 11)

#

# Abbreviations:
```

```
# ISO : International Standards Organization

# BGN : U.S. Board on Geographic Names

# UNGEGN : United Nations Group of Experts on Geographic Names

# PCGN: U.K. Permanent Committee on Geographic Names

# UN : United Nations

# FAO : Food and Agriculture Organization of the United Nations

# FIPS : Federal Information Processing Standard

# EKI : Institute of Estonian Language
```

Dette programmet skal skrive hele filen til skjermen. Om du har problemer med innlesingen kan du prøve å spesifisere at filen er kodet i UTF-8 ved å bruke `open("cow.txt", encoding="utf-8")`.

Merk: Det er en god vane å **alltid** spesifisere `encoding`. Spesielt på Windows er standardkodingen avhengig av oppsettet slik at uten en spesifsert koding kan programmer som virker på en maskin plutselig ikke virke på en annen.

La oss nå se litt nærmere på selve filen `cow.txt`. Først kan vi legge merke til to ting:

- Øverst i filen er det flere linjer som begynner med tegnet `#` som ikke inneholder informasjon vi er interessert i. Vi bør ignorere disse linjene.
- Hver linje (unntatt linjene som starter med `#`) inneholder informasjon om et land, med hvert informasjonsfelt skilt med semikolon. Vi vil plukke ut noen av disse informasjonsfeltene.

Utvid programmet ditt slik at det ser slik ut:

[100]:
```
with open("cow.txt") as f_in:
    next(f_in)
    for line in f_in:
        if line.startswith("#"):
            continue

        fields = line.split(";")
        print(fields[4], fields[38])
```

```
Namibia  Windhoek
Nauru  Yaren
Nepal  Kathmandu
Netherlands  Amsterdam
New Caledonia
New Zealand  Wellington
Nicaragua  Managua
```

```
Niger  Niamey
Nigeria  Abuja
Niue
Norfolk Island
Norway  Oslo
```

Når du kjører dette programmet vil du få en liste over land og hovedsteder. La oss skjønne hva som skjer!

- Vi kan sjekke om en linje begynner med tegnet `#` ved hjelp av `line.startswith`. Dersom den gjør det går vi videre til neste linje med `continue`, siden resten av løkken blir hoppet over.
- Vi deler hver linje i en liste av felter ved hjelp av `split`, og vi forteller `split` at vi vil bruke semikolon, `;`, som skilletegn mellom feltene.
- Etter litt prøving og feiling finner vi at land og hovedstad ligger som felt 4 og 38 (telt fra 0). Vi skriver derfor ut disse feltene.
- Det er et problem med første linje i `cow.txt`-filen. Den ser ut som om den begynner med en `#`, men det ligger et usynlig tegn helt først som gjør at testen vår ikke slår inn. I stedet for å lage en mer avansert test hopper vi bare over første linje ved å si `next(f_in)` rett etter at vi åpner filen.

Om du ser på listen over land og hovedsteder som skrives ut vil du se et par små utfordringer til. Den første linjen som skrives ut er ikke et land, men en forklarende overskrift. Vi vil ikke at denne skal være med i listen vår over hovedsteder. En del land (eller "land") har ingen hovedstad. Disse kan vi heller ikke ta med i quizen. Til slutt er det mellomrom foran navnene på landene og hovedstedene. Disse kan vi fjerne med `strip`. Den neste versjonen av programmet tar også høyde for disse utfordringene, samtidig som dataene leses inn i `dicten` `hovedsteder` i stedet for å skrives til skjerm.

[101]:
```python
import random

capitals = {}

with open("cow.txt") as f_in:
    next(f_in)
    for line in f_in:
        if line.startswith("#"):
            continue
        fields = [f.strip() for f in line.split(";")]
        if len(fields[0]) == 2 and fields[38]:
            capitals[fields[4]] = fields[38]

with open("verdensquiz.txt", mode="w") as f_out:
    countries = random.sample(list(capitals), 100)
    for number, country in enumerate(countries, start=1):
        f_out.write(f"\n\n{number}. What is the capital of {country}?\n   ")
        wrong_answers = random.sample(list(capitals.values()), 3)
        alternatives = sorted(wrong_answers + [capitals[country]])
        f_out.write("   ".join(alternatives))
```

Stort sett er dette programmet satt sammen av ting vi har sett tidligere, men noe nytt har også dukket opp.

- I filen `cow.txt` er det en del mellomrom mellom feltene som vi må ta vekk. Det er nettopp dette `strip` gjør, fjerne mellomrom på begynnelsen og slutten av tekststrenger. Det gjør det for eksempel mulig å teste om et land har angitt en hovedstad ved å teste at hovedstad-feltet (`fields[38]`) har innhold.
- For å unngå overskriftlinjen tester vi også på om felt 0 har lengde nøyaktig 2, siden dette skal være en 2-bokstavers kode som identifiserer landet.
- For å gjøre quizen litt enklere vil vi også gi alternativer. Dette gjør vi igjen ved hjelp av `random`-pakken. Funksjonen `random.sample` plukker et gitt antall ting tilfeldig fra en liste. Vi lager alternativene våre ved å sette det riktige svaret sammen tre feil svar (her er det en liten *bug* i programmet vårt, vi er ikke helt sikre på at alle byene i `feil_svar` faktisk er feil). Ved å sortere svaralternativene skjuler vi hvilket svar som er det riktige.
- Funksjonen `join` slår sammen en liste av tekststrenger til en lengre tekststreng.

Programmet skriver quizen til en tekstfil, `verdensquiz.txt`. Et eksempel på hvordan denne kan se ut (rekkefølgen på spørsmålene og alternativene er tilfeldig) er som følger:

[102]:
```
!python cat.py verdensquiz.txt
```

```
1. What is the capital of Antigua and Barbuda?
   Berne   Conakry   San Marino   St. John's

2. What is the capital of Denmark?
   Copenhagen   St George's   St. John's   Suva

3. What is the capital of Iran, Islamic Republic of?
   Ouagadougou   Tehran   Vienna   Vilnius

4. What is the capital of Russian Federation?
   Baghdad   Moscow   Roseau   Vatican City State

5. What is the capital of Brunei Darussalam?
   Baghdad   Bandar Seri Begawan   Beirut   Nairobi

6. What is the capital of Bahamas?
   Caracas   Nassau   Vienna   Vientiane

7. What is the capital of Comoros?
   Bandar Seri Begawan   Bishkek   Manila   Moroni

8. What is the capital of Liechtenstein?
   Accra   Freetown   San Salvador   Vaduz

9. What is the capital of Mauritius?
   Amman   Male'   Port Louis   Praia

10. What is the capital of Benin?
```

```
Brazzaville    Hanoi    Porto-Novo    Tallinn
```

Dette siste eksempelet kan virke litt komplisert, men samtidig har vi sett hvordan det er bygd opp av mange små biter som hver for seg er ganske forståelige. Slik er det med det meste av ekte data. De er rotete og må ryddes i. Heldigvis kan vi som regel rydde i en ting av gangen som vi gjorde her.

4.5 Manipulering av filsystemet

Modulene `pathlib` og `shutil` støtter de fleste operasjonene på filsystemet.

For eksempel kan `.exists()`, `.is_file()` og `.is_dir()` brukes for å sjekke om en fil eller katalog eksisterer. Metoden `.glob()` brukes for å gjøre mønstersøk i filsystemet, for eksempel vil `.glob('*.log')` returnere en liste over alle `.log`-filer. For å kopiere eller flytte filer kan man bruke `shutil.copy()` eller `shutil.move()`.

[103]:
```
pathlib.Path("verdensquiz.txt").exists()
```

[103]:
```
True
```

[104]:
```
pathlib.Path("hello.py").is_file()
```

[104]:
```
True
```

[105]:
```
pathlib.Path("hello.py").is_dir()
```

[105]:
```
False
```

[106]:
```
list(pathlib.Path.cwd().glob("c*.py"))
```

[106]:
```
[PosixPath('/home/gahjelle/Dropbox/presentasjoner/20220922_python_nve/intro/cat.
py')]
```

[107]:
```
import shutil

shutil.copy("hello.py", "new_hello.py")
```

[107]:
```
'new_hello.py'
```

5 Datoer og kalendere

Python har støtte for å regne med datoer og klokkeslett gjennom pakkene `datetime` og `calendar`.

5.1 Datoer med `datetime`

Pakken `datetime` inneholder modulene `date`, `time` og `datetime` for å jobbe med henholdsvis datoer, klokkeslett og datoer med klokkeslett.

[108]:
```
from datetime import date

now = date.today()
now
```

[108]:
```
datetime.date(2022, 9, 22)
```

[109]:
```
now.isocalendar()
```

[109]:
```
datetime.IsoCalendarDate(year=2022, week=38, weekday=4)
```

[110]:
```
help(now.isocalendar)
```

```
Help on built-in function isocalendar:

isocalendar(…) method of datetime.date instance
    Return a named tuple containing ISO year, week number, and weekday.
```

I eksempelet over ser vi at 22. september 2022 er en torsdag (ukedag nummer 4) i uke 38. Vi kan også regne ut forskjellen mellom to datoer:

[111]:
```
christmas = now.replace(month=12, day=24)
christmas
```

[111]:
```
datetime.date(2022, 12, 24)
```

[112]:
```
christmas - now
```

[112]:
```
datetime.timedelta(days=93)
```

[113]:
```
(christmas - now).days
```

[113]:
```
93
```

[114]:
```
(christmas - now).months
```

```
---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)
Cell In [114], line 1
----> 1 (christmas - now).months

AttributeError: 'datetime.timedelta' object has no attribute 'months'
```

Modulene `time` og `datetime` fungerer tilsvarende. Dessverre er `datetime`-pakken litt begrenset i forhold til hvilken informasjon den gir om forskjellen på to datoer. Om du trenger mer avansert datohåndtering kan du bruke `dateutil`.

5.2 Mer avansert datohåndtering med `dateutil`

Pakken `dateutil` er ikke en del av standardbiblioteket, og må derfor installeres eksplisitt. Informasjon om pakken finnes på dateutil.readthedocs.org, og den kan installeres gjennom `conda` eller `pip`:

[ ]:
```
!conda install python-dateutil -y

# !pip install python-dateutil
```

Spesielt er den bedre utbygd en `datetime` i forhold til å beregne forskjellen mellom to datoer, eller til å beregne en dato relativt til en annen dato.

[115]:
```
from datetime import date
from dateutil import relativedelta

now = date.today()
christmas = now.replace(month=12, day=24)

relativedelta.relativedelta(christmas, now)
```

[115]:
```
relativedelta(months=+3, days=+2)
```

Vi kan også finne ut når den 5. fredagen fra i dag faller, eller den siste søndagen denne måneden:

[116]:
```
now + relativedelta.relativedelta(weekday=relativedelta.FR(5))
```

[116]:
```
datetime.date(2022, 10, 21)
```

[117]:
```
now + relativedelta.relativedelta(day=31, weekday=relativedelta.SU(-1))
```

[117]:
```
datetime.date(2022, 9, 25)
```

Et annet eksempel er utregning av bursdager:

[118]:
```
from datetime import datetime
from dateutil.relativedelta import relativedelta

bday_str = input("Når er du født (yyyy-mm-dd)? ")
birthday = datetime.strptime(bday_str, "%Y-%m-%d")
age = relativedelta(datetime.today(), birthday)
print(f"Du er {age.years} år gammel")
```

```
Når er du født (yyyy-mm-dd)?  1997-12-08
```

```
Du er 24 år gammel
```

Pakken `dateutil` kan også hjelpe med utregning av påsken, eller andre helligdager som beveger seg med påsken. Eksempelene under regner ut påskeaften og kristi himmelfart 2023. Modulen `dateutil.easter` finner 1. påskedag så vi må flytte svaret fra denne en dag.

[119]:
```
from dateutil.easter import easter
from dateutil.relativedelta import relativedelta

p_aften = easter(2023) + relativedelta(days=-1)
p_aften
```

[119]:
```
datetime.date(2023, 4, 8)
```

[120]:
```
p_aften + relativedelta(days=40)
```

[120]:
```
datetime.date(2023, 5, 18)
```

5.3 `calendar`-pakken

For enkle kalendere er `calendar`-pakken nyttig. For eksempel kan vi skrive ut en månedskalender på denne måten:

[121]:
```
import calendar

print(calendar.month(2022, 9))
```

```
   September 2022
Mo Tu We Th Fr Sa Su
          1  2  3  4
 5  6  7  8  9 10 11
12 13 14 15 16 17 18
19 20 21 22 23 24 25
26 27 28 29 30
```

En full årskalender kan skrives ut ved hjelp av `print(calendar.calendar(2022))`. Navnene på ukene og månedene er også tilgjengelige. For eksempel:

[122]:
```
list(calendar.day_name)
```

[122]:
```
['Monday', 'Tuesday', 'Wednesday', 'Thursday', 'Friday', 'Saturday', 'Sunday']
```

[123]:
```
list(calendar.month_abbr)
```

[123]:
```
['',
 'Jan',
 'Feb',
 'Mar',
 'Apr',
 'May',
 'Jun',
 'Jul',
 'Aug',
```

```
 'Sep',
 'Oct',
 'Nov',
 'Dec']
```

Disse er lokaliserte, slik at på en datamaskin med norsk oppsett vil dagene og månedene være oversatt til norsk.

6 Lesbar kode, dokumentasjon og testing

I Python har lesbarhet alltid vært sterkt vektlagt, og effektivitet for utviklerene er noe Python alltid scorer høyt på. Det finnes også et eget dokumentasjonssystem bygd inn i språket.

6.1 Lesbar kode

Pythonkode er som regel ganske lesbar. Høynivå operasjonene og datastrukturene, sammen med den tvungne kodeformatteringen, gjør at Python av og til blir omtalt som **kjørbar pseudokode**.

Det følgende er noen tommelfingerregler som også hjelper på lesbarheten:

- Bruk mellomrom for å øke lesbarheten: Bruk et mellomrom rundt operatorer, et mellomrom etter komma i forskjellige typer lister, to blanke linjer mellom funksjoner og så videre.
- Ikke skriv for lange linjer. En vanlig konvensjon er at hver kodelinje skal være maksimalt 79 tegn. Se under for hvordan du kan dele opp lange linjer.
- Bruk konsistente og meningsfylte navn på variabler, funksjoner, klasser og så videre.

6.2 Del opp lange kodelinjer

Kodelinjer i Python kan deles opp inne i forskjellige typer paranteser. Vi så et eksempel på dette i definisjonen av `OPERATIONS` tidligere.

I eksempelet nedenfor ser vi hvordan vi kan legge inn et linjeskift i `def`-linja og funksjonskallet nederst uten problemer, fordi disse allerede inneholder paranteser. I `if`-testen og `return`-kallet har vi lagt til en ekstra parantes rundt uttrykket for å kunne dele det på to linjer. I det øverste `print`-uttrykket har vi i tillegg benyttet oss av implisitt streng-sammensetting.

[124]:
```python
def en_funksjon_med_et_altfor_langt_navn(en_variabel, en_variabel_til):
    print(
        "En lang tekststreng kan deles opp som "
        "dette hvis den er plassert inne i paranteser!"
    )

    if en_variabel > 10 and en_variabel * en_variabel_til < en_variabel**2:
        return en_variabel**2 - en_variabel * en_variabel_til
    else:
        return en_variabel


print(en_funksjon_med_et_altfor_langt_navn(2**10 - 1000, 10**2 - 9**2))
```

```
En lang tekststreng kan deles opp som dette hvis den er plassert inne i
paranteser!
120
```

Lange tekststrenger som inneholder linjeskift skrives enklest ved å bruke tre fnutter i Python.

[125]:
```
jan_erik_vold = """Det er
ingenting. Og det
blir
ingen

ting.
Og innimellom
myldrer
det."""

print(jan_erik_vold)
```

```
Det er
ingenting. Og det
blir
ingen

ting.
Og innimellom
myldrer
det.
```

Tekststrenger som defineres på denne måten bevarer også linjeskift og indentering.

6.3 Automatisk formatering

Programmet Black kan brukes til å automatisk formattere kode. Det er **veldig** nyttig! Black kan settes opp slik at det for eksempel kjøres hver gang du lagrer en fil i PyCharm.

Black har i praksis ingen opsjoner. Det betyr at all Black-formatert kode ser veldig konsistent ut.

6.4 Programstruktur

Som nevnt tidligere, leses en Python-fil linje for linje. En vanlig programstruktur er derfor at man definerer alle funksjoner først, og starter selve programmet nederst i Python-fila.

[126]:
```
# 1: Importer moduler
import math

# 2: Definer globale konstanter
EXPONENT = 3


# 3: Definer hovedskriptet
```

```
def main():
    print(func2("14.2"))


# 4: Definer hjelpefunksjoner
def func1(number):
    return number**EXPONENT


def func2(number_string):
    return func1(float(number_string)) + math.pi


# 5: Ikke start hovedskriptet hvis filen importeres
if __name__ == "__main__":
    main()
```

```
2866.429592653589
```

6.5 Kommentarer

Kommentarer i Python startes med `#`. En `#` hvor som helst på en kodelinje (unntatt inne i en tekststreng) vil starte en kommentar, og resten av linjen ignoreres av Python-tolkeren. Gode kommentarer er kommentarer som gjør koden mer lesbar og forståelig.

En effektiv måte å kommentere ut deler av koden mens man utvikler er å bruke tre fnutter, `"""`. Dette gjør om koden til en tekststreng, som så ignoreres av Python.

[127]:
```
number = 14

"""
if number < 20:
    print('Tallet er mindre enn 20')
"""

if number % 2 == 0:
    print("Tallet er et partall")
```

```
Tallet er et partall
```

6.6 Dokumentasjon

Python har et innebygd dokumentasjonssystem hvor spesielt plasserte tekststrenger (kalt **doc-strings**) legges inn i selve objektet, og kan hentes ut som dokumentasjon på forskjellige måter. For eksempel, for å dokumentere en funksjon skriver man docstringen rett etter `def`-uttrykket.

[128]:
```
%%writefile documented.py
def is_selfsquared(number):
    """Sjekker om et gitt tall er selvkvadratisk.
```

```
    Vi sier et tall er selvkvadratisk dersom tallet kan
    skrives AB slik at (A + B) * (A + B) er lik AB.

    Eksempel:
    >>> is_selfsquared(60481729)
    (6048 + 1729)^2 = 60481729
    True
    >>> is_selfsquared(42)
    False
    """
    number_str = str(number)
    for idx in range(1, len(number_str)):
        number_a = int(number_str[:idx])
        number_b = int(number_str[idx:])
        if (number_a + number_b) ** 2 == number:
            print('({} + {})^2 = {}'.format(number_a, number_b, number))
            return True
    return False

if __name__ == '__main__':
    for number in range(100_000):
        is_selfsquared(number)
```

```
Writing documented.py
```

[129]:
```
!python documented.py
```

```
(8 + 1)^2 = 81
(10 + 0)^2 = 100
(20 + 25)^2 = 2025
(30 + 25)^2 = 3025
(98 + 1)^2 = 9801
(100 + 0)^2 = 10000
(88 + 209)^2 = 88209
```

Denne dokumentasjonen kan for eksempel hentes ut interaktivt.

[130]:
```
import documented

help(documented.is_selfsquared)
```

```
Help on function is_selfsquared in module documented:

is_selfsquared(number)
    Sjekker om et gitt tall er selvkvadratisk.

    Vi sier et tall er selvkvadratisk dersom tallet kan
```

```
    skrives AB slik at (A + B) * (A + B) er lik AB.

    Eksempel:
    >>> is_selfsquared(60481729)
    (6048 + 1729)^2 = 60481729
    True
    >>> is_selfsquared(42)
    False
```

Dokumentasjonsverktøy som for eksempel Sphinx og MkDocs bruker også disse tekststrengene for å automatisk bygge dokumentasjon for kode. For eksempel er deler av dokumentasjonen på docs.python.org hentet fra docstrenger.

6.7 Enkel testing med `doctest`

I eksempelet ovenfor inkluderte vi noen eksempler på hvordan funksjonen kan brukes, og med forventede returverdier. Python kan automatisk kjøre disse eksemplene som tester for oss, ved å bruke modulen `doctest` som er en del av standardbiblioteket.

[131]:
```
!python -m doctest documented.py
```

Vanligvis er `doctest` helt stille dersom alle testene passerte. Ved å legge til opsjonen `-v` vil `doctest` bli mer pratsom og fortelle om alle testene den gjør.

[132]:
```
!python -m doctest -v documented.py
```

```
Trying:
    is_selfsquared(60481729)
Expecting:
    (6048 + 1729)^2 = 60481729
    True
ok
Trying:
    is_selfsquared(42)
Expecting:
    False
ok
1 items had no tests:
    documented
1 items passed all tests:
   2 tests in documented.is_selfsquared
2 tests in 2 items.
2 passed and 0 failed.
Test passed.
```

6.8 Mer avansert testing med `pytest`

`pytest` er det vanligste verktøyet for testing i Python. Det har spesielt god støtte for enhetstesting.

7 Python-verden

Python-verdenen vokser stadig. De siste årene har Python blitt stadig mer populært for eksempel i akademia, i takt med at gode pakker for forskjellige fagfelt blir utviklet. Samtidig har Python klart å bevare noe av de små quirksene som gjør også communityen rundt Python litt spesiell.

7.1 Monty Python

Bare for å ha det sagt. Python er **ikke** kalt opp etter en slange. Språket fikk navnet sitt fordi Guido van Rossum var en stor fan av Monty Python: "*I chose Python as a working title for the project, being in a slightly irreverent mood (and a big fan of Monty Python's Flying Circus).*" Om man følger med vil man se små referanser til Monty Python rundt omkring i Python-verdenen.

7.2 PyPI - The Python Package Index

Det finnes veldig mange pakker og moduler utviklet til Python. De fleste av disse er tilgjengelige på PyPI. Det er nå omtrent 400 000 pakker tilgjengelig innenfor alle mulige fagfelt og bruksområder på PyPI. Disse pakkene installeres vanligvis ved hjelp av et program som heter `pip`.

7.3 PEP-8

En **PEP** er en **Python Enhancement Proposal**. Dette er dokumenter som brukes for å styre den videre utviklingen av språket. Hvem som helst kan skrive en PEP for eksempel for å foreslå en ny feature i språket. Denne PEP'en blir så debattert før det blir tatt en avgjørelse på om dem skal implementeres. For et nylig eksempel ble PEP-536 implementert i Python 3.10. Denne PEPen innfører et nytt `match case` uttrykk som kan brukes til kraftig behandling av datastrukturer.

PEP-8 definerer en **Style Guide for Python Code**. Dette dokumentet inneholder mange små, konkrete tips om hvordan man skriver både lesbar og god Python-kode og er vel verdt en titt.

7.4 `import this` - The Zen of Python

I PEP-20 har Tim Peters skrevet et dikt som beskriver mye av filosofien bak Python. Dette diktet følger også med alle Python-installasjoner som et slags *easter egg*.

[133]:
```
import this
```

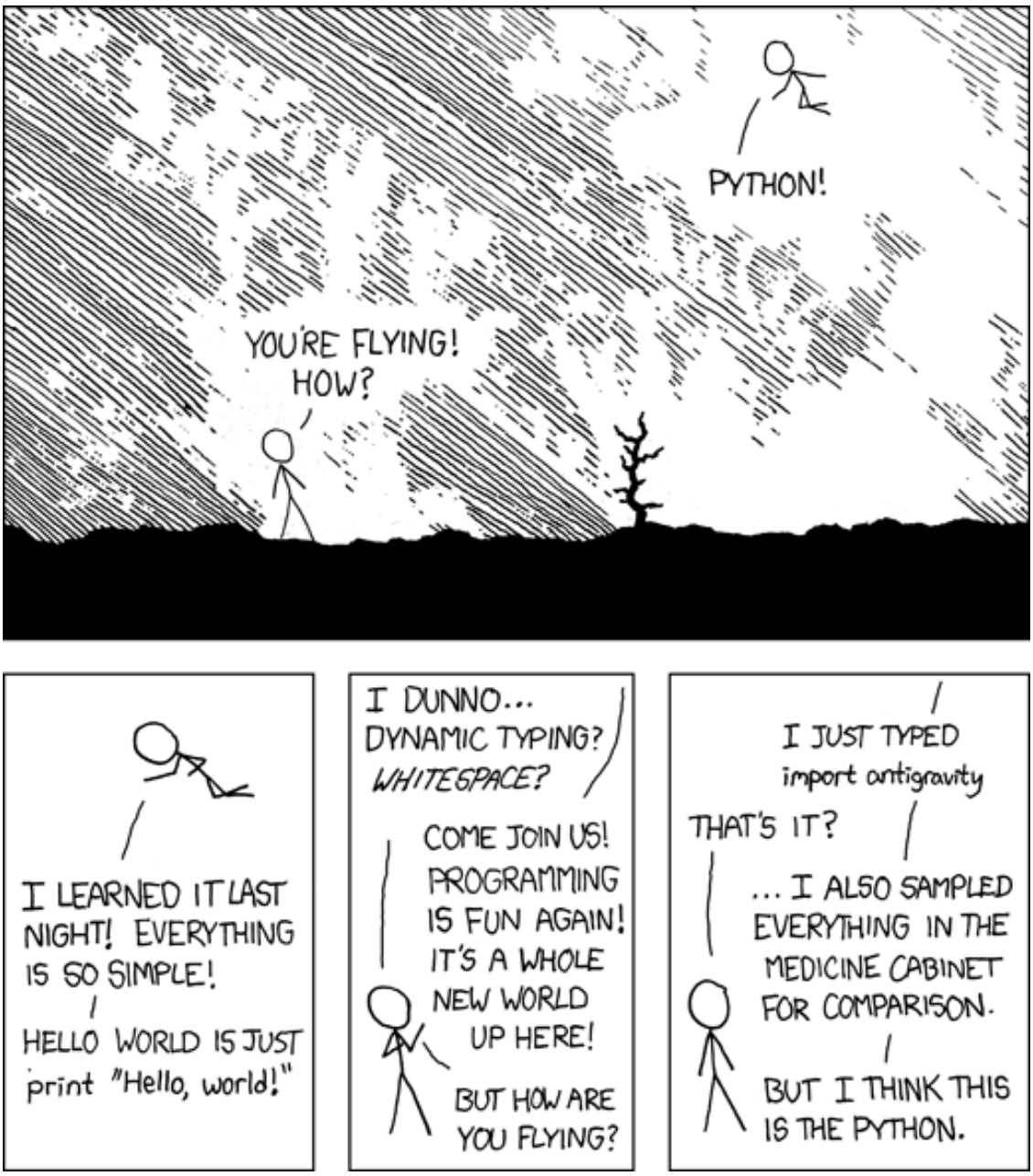
```
The Zen of Python, by Tim Peters

Beautiful is better than ugly.
Explicit is better than implicit.
Simple is better than complex.
Complex is better than complicated.
Flat is better than nested.
Sparse is better than dense.
Readability counts.
Special cases aren't special enough to break the rules.
Although practicality beats purity.
Errors should never pass silently.
Unless explicitly silenced.
```

```
In the face of ambiguity, refuse the temptation to guess.
There should be one-- and preferably only one --obvious way to do it.
Although that way may not be obvious at first unless you're Dutch.
Now is better than never.
Although never is often better than *right* now.
If the implementation is hard to explain, it's a bad idea.
If the implementation is easy to explain, it may be a good idea.
Namespaces are one honking great idea -- let's do more of those!
```

7.5 `import antigravity`

Tegneserien XKCD har et kjærlig forhold til Python. Dette forholdet er gjengjeldt, dels gjennom easter egget `import antigravity`.

[ ]:
```
import antigravity
```



7.6 docs.python.org

Den offisielle dokumentasjonen til Python finnes på docs.python.org. Denne er (ihvertfall tidvis) informativ og nyttig. Se også litt på de forskjellige *tutorials* og *how-tos* som er tilgjengelig på denne siden.

8 Veien videre

Dette korte kurset dekker veldig vidt. Det betyr at mange detaljer er utelatt fra dette heftet. Disse detaljene finner du stort sett godt forklart i Python-dokumentasjonen eller andre steder på nettet. Nedenfor nevnes også noen bøker som kan være nyttige.

Selv om vi har rukket over mye på kort tid er det også store hull i vår oversikt over Python. Noen av de viktigere hullene nevnes til slutt. Se mer dokumentasjon på nettet eller i boklisten for mer informasjon om disse temaene.

8.1 Bokliste

Det finnes selvsagt altfor mange bøker om Python. Her er en kort liste til noen bøker som kan være nyttige:

- **Introducing Python (Bill Lubanovic)** Denne dekker mye av det samme territoriet som dette kurset (og litt til), men i mer detalj. Boken bruker mye eksempler for å vise frem hvordan ting fungerer.
- **Automate the Boring Stuff with Python (Albert Sweigart)** Denne boken kan fåes gratis på nettet og viser hvordan man gjør nyttige administrasjonsoppgaver som filbehandling, organisering av filer, behandling av regneark, PDF- og Word-dokumenter, i tillegg til hvordan man jobber med CSV- og JSON-data.
- **Python Cookbook (David Beazley og Brian K. Jones)** Denne boken er skrevet som en kokebok, og inneholder mange nyttige oppskrifter på hvordan man gjør konkrete oppgaver.
- **Fluent Python (Luciano Ramalho)** Denne går i dybden og er veldig nyttig for de som virkelig vil lære *the inns and outs* av Python. Ikke for nybegynner, dog.

8.2 Nettsteder

- **Stack Overflow** Programmering er jo stort sett en øvelse i klipp og lim fra Stack Overflow :blush: Stack Overflow har svarene på de fleste spørsmålene du klarer å stille om Python.
- **Real Python** publiserer dybdeartikler og -videoer om Python hver uke. Disclaimer: Jeg jobber for Real Python
- **PyVideo** Det arrangeres mange konferanser om Python. Om du har muligheten anbefaler jeg absolutt å delta på en av dem. Heldigvis blir mange av foredragene filmet og lagt ut på nett. På PyVideo finner du en flott oversikt.

8.3 Klasser og objekter

Vi har såvidt nevnt objekter. Python er et fullverdig objektorientert språk, og har en veldig ryddig og eksplisitt modell for hvordan egendefinerte klasser og typer implementeres.

8.4 Feilhåndtering

Python har et **exception**-system som ligner det som finnes i mange andre programmeringsspråk. Spesielt kan man bruke `try` og `except`for å ta mer kontroll over hvordan feil håndteres i spesielle deler av koden.

I tillegg kommer Python med en innebygd debugger som kan hjelpe med feilsøking i koden. For å bruke debuggeren kan for eksempel linjen `breakpoint()` legges inn i kildekoden der du ønsker debuggeren skal starte fra. Debuggeren har sine egne kommandoer du kan bruke. For å se en liste over disse kan du skrive `h` eller `help` etter at debuggeren har startet opp.

8.5 Indre og anonyme funksjoner

Det er mulig å definere indre funksjoner i Python. Det vil si funksjoner definert inne i andre funksjoner. Dette kan være nyttig i forhold til informasjonsutvekslingen mellom funksjonene. Python har også enkel støtte for anonyme funksjoner gjennom uttrykket `lambda`. Anonyme funksjoner er ikke-navngitte funksjoner som ofte er definert for å gjøre en liten, konkret oppgave. For eksempel, kan en `lambda`-funksjon brukes for å definere en spesiell måte å sortere en liste på.

8.6 Generatorer

Generatorer er spesielle funksjoner med en slags iterator-oppførsel. I stedet for å gjøre ferdig en hel beregning slik en funksjon typisk gjør, kan en generator vente med utregninger til den blir bedt om dem. Det høres litt mystisk og akademisk ut, men er spesielt nyttig i minnetunge applikasjoner hvor en generator-implementasjon ofte gjør at man unngår å måtte ha hele datastrukturer i minnet samtidig.

8.7 Regulære uttrykk: Mer avansert teksthåndtering

Regulære uttrykk (*regular expressions*) er et eget minispråk for avansert teksthåndtering. Regulære uttrykk har sitt utspring fra matematikk og teoretisk computer science. Gjennom 1970-tallet ble de viktige komponenter i flere programmer og språk i Unix-verden. Regulære uttrykk kan være litt kompliserte å bruke, men de er utrolig kraftige innen behandling av tekst. De fleste språk har implementert regulære uttrykk i dag. Python støtter dem gjennom modulen `re`.