Linnæus University

Grunderna i Python, del 3

Tobias Andersson Gidlund

Agenda

- Lite objektorientering
- Ytterligare datatyper
- I/O
- Felhantering

De sista bitarna

- Allt som ”behövs” har redan behandlats, de sista bitarna är sådant som inte är strikt nödvändigt
- Det är dock fortfarande väldigt värdefullt!
- De olika koncept som tas upp i den här föreläsningen gör program säkrare och mer robusta samt lättare att underhålla
- Till viss del handlar det också om inbyggda saker som underlättar arbetet med att ta fram mjukvara
  - Att inte behöva uppfinna hjulet på nytt

Objektorientering

Klasser och objekt

- En teknik, eller förhållningssätt, till programmering är vad som kallas *objektorientering*
- Här skapar man entiteter av självbestämmande enheter som kommunicerar med varandra
  - Tanken är att det ska påminna om verkligheten
  - En person är ett objekt som man frågar om information
- Objektorientering är egentligen inte bara en egen lektion, utan till och med en egen kurs
  - Så här blir det bara en enkel grund

Inbyggda klasser i Python

- Python innehåller en uppsättning inbyggda klasser som redan har använts
  - Till exempel `random` och `list`
- Det är till och med så att nästan allt i Python faktiskt är klasser – även primitiva typer

```
print(type(42))
print(type('Star Wars'))
```

```
<class 'int'>
<class 'str'>
```

Klasser

- I Python skapar man *klasser* som är en mall för hur data och beteende man vill jobba med ska se ut
  - Data lagras som variabler
  - Beteende som funktioner
- Variablerna kallas också *tillståndsvariabler* eftersom de, när programmet kör, kommer att hålla ett tillstånd
- Meddelande skickas mellan olika delar av programmet via funktionerna

Objekt

- Från klasserna skapar man *objekt* som är de delar i programmet som har tillstånd och kommunicerar
- Det kallas att skapa en *instans* av klassen
- En klass kan användas för att skapa många instanser, där varje instans är unik
- Objekten når man via en variabel (eller som del i en lista eller likande)
- Funktionerna kallas *metoder* i objektorientering

Inkapsling

- Det här innebär att varje objekt ses som en egen entitet, en samlad helhet som är ansvarig för något
- För objektorienterade program innebär det ofta att det är mappat mot ett verkligt föremål
  - En person, en bil, ett bankkonto, en semesterresa
  - Något som ses som en helhet i *verkligheten*
- Det kan, som exemplen visar, vara både fysiska saker, men även abstrakta, ofta substantiv, som finns i *problemdomänen*

Exempel på klass

- Klassen har ett namn samt en metod som heter `__init__(self)`

```
class Song:
  def __init__(self):
    self.name = ''    # Initieras som tom
    self.album = ''   # Som ovan
    self.year = 1900  # Ges värdet 1900 som standard
```

- Metoden är vad som kallas en *konstruktor*, används för att initiera objektet med värden
  - Parametern `self` är obligatorisk och kan heta något annat, men praxis är att kalla den så här

Skapa objekt

- Från klasserna måste man sedan skapa objekt som man refererar till via en variabel

```
time = Song()
shine = Song()
```

- De olika delarna nås sedan via *punktnotation* på objektet

```
time.name = 'Time'
time.album = 'Dark Side of the Moon'
time.year = 1973
```

Fullständigt exempel

```
class Song:
  def __init__(self):
    self.name = ''    # Initieras som tom
    self.album = ''   # Som ovan
    self.year = 1900  # Ges värdet 1900 som standard

# Här börjar programmet
time = Song()

time.name = 'Time'
time.album = 'Dark Side of the Moon'
time.year = 1973

print(time.name)
print(time.album)
print(time.year)
```

```
Time
Dark Side of the Moon
1973
```

Praxis

- Namnet på klassen ska börja med stor bokstav
- De attribut, variablerna, som finns inne i klassen ska skrivas med liten bokstav
- Det kommer att finnas fler metoder och de ska också börja med liten bokstav
- Syftet med metoderna är att *skydda* objektet mot felaktig användning
- Alla objekt är *referenser* så skickas de till en funktion så kan de ändras direkt

Utöka konstruktorn

- Eftersom konstruktorn är en metod så går det att skicka fler parametrar till den
  - Observera att `self` alltid måste finnas med och vara först i listan

```
class Song:
  def __init__(self, name, album, year):
    self.name = name
    self.album = album
    self.year = year
```

- Den måste då också få alla parametrar när objektet skapas

```
shine = Song('Shine on you crazy diamond, part I-V', 'Wish You Were Here', 1975)
```

Fler metoder

- Det vanliga är att det finns en metod för att sätta och en för att hämta varje attribut
  - Kallas även *getters* och *setters*, en del av vad som kallas *inkapsling*
- Utöver det brukar man ha metoder för olika beräkningar och liknande
  - Till exempel för att beräkna area för ett cirkelobjekt och så vidare
- Namnen på metoderna, och vilka parametrar man vill använda, bestämmer man själv, men `self` måste alltid finnas med

Exempel

```
class Song:
  def __init__(self, name, album, year):
    self.name = name
    self.album = album
    self.year = year

  def set_year(self, year):
    if year > 1900 and year < 2100:
      self.year = year

  def get_year(self):
    return self.year

# Här börjar programmet
money = Song('Money', 'Dark Side of the Moon', 1973)
print(money.get_year())
money.set_year(1873)
print(money.get_year())
```

- Körning:

```
1973
1973
```

Mer om objektorientering

- I objektorientering är det vanligt att se till att attributen är *privata*, dvs att man inte kan nå dem annat än med metoder
- För att göra det i Python behöver namnet ändras till att börja med två understrykningssträck
  - Kallas även *name mangling*
  - Exempel: `self.__name = name` i konstruktorn
- Det är också viktigt med *arv*, dvs att en klass *utökar* en annan
- En **svala** *är en* **fågel**, här är **fågel** en superklass och **svala** en subklass
- Superklassen sätts inom parentes i definitionen av subklassen: `class Swallow(Bird)`

Meddelanden

- En viktig del i objektorienteringen är att objekten pratar med varandra
- För att det ska vara möljligt måste en referens skapas till den andra klassen
- Det görs genom att deklarera en variabel av det andra objektet i det första
  - Det kan också vara en lista av objekt
- Genom att anropa metoder på objektet skickas meddelanden

Exempel

```
class Song:
  def __init__(self, name):
    self.name = name

class Album:
  def __init__(self):
    self.__name = ''
    self.__year = 1900
    self.__songs = list()

  def set_name(self, name):
    if not name == '':
        self.__name = name

  def get_name(self):
      return self.name

  def add_song(self, song):
    self.__songs.append(song)

  def show_songs(self):
    for i in self.__songs:
        print(i.name)
```

```
# Här börjar programmet
dsotm = Album()
dsotm.set_name('Dark Side of the Moon')
dsotm.year = 1973

time = Song('Time')

dsotm.add_song(time)
dsotm.add_song(Song('Money'))

dsotm.show_songs()
```

➤ Körning:

```
Time
Money
```

Operatoröverlagring och speciella metoder

- För att klasser ska kunna fungera i alla sammanhang finns det en möjlighet att själv definiera till exempel `<`, `>` och `+`
  - Det görs med speciella metoder som `__add__(self, other)`
- För att göra det enklare att se innehåll i ett objekt, kan man även implementera en metod som heter `__str__`
  - Den ska ge en *strängrepresentation* av objektet
- Även om metoderna kallas ”speciella” så är det bara namnet som är speciellt, de implementeras precis som vanligt

Sammanfattning

- Objektorientering är att kraftfullt sätt att dela upp program i logiska delar
- Delarna är mer eller mindre autonoma och kommunicerar med varandra
- På så sätt kan man skapa stora program som är förhållandevis enkla att förstå och underhålla
- Här har vi bara skrapat på ytan, det kommer att komma mer i de olika delmomenten i kursen

Inbyggda datastrukturer

Python datastrukturer

- Det finns andra inbyggda datastrukturer i Python än listor
- Dessa strukturer skiljer sig i sina egenskaper, såsom förändringsbarhet, ordning och hur de kopplar data
- Varje struktur har sina egna styrkor, vilket gör den särskilt lämpad för vissa uppgifter och problemlösningsmetoder
- Att förstå dessa ytterligare datastrukturer kommer att göra det enklare att lösa fler olika problem
- Tre ytterligare datastrukturer som är inbyggda i Python kommer att täckas: *tupler*, *mängder* och *lexikon*

Tupler

- Tupler liknar listor på så sätt att de är ordnade sekvenser av element
- Skillnaden är att tupler är *oföränderliga*
  - Vilket betyder att när de skapats kan de inte ändras
- På grund av detta kan implementeringen av tupler göras mer minneseffektiv
  - De använder mindre minne än listor
  - Python kan också optimera minnesallokering eftersom de är oföränderliga
- Detta leder i sin tur till att tupler har något bättre prestanda för till exempel iteration

Skapa tupler

- Skapandet liknar listor, men istället för hakparenteser används vanliga parenteser

```
colours = ('red', 'green', 'blue')
print(colours)
```

```
('red', 'green', 'blue')
```

- Tupler kan också skapas från en lista

```
colour_list = ['yellow', 'purple', 'orange']
more_colours = tuple(colour_list)
print(more_colours)
```

```
('yellow', 'purple', 'orange')
```

Åtkomst till element

- Precis som för listor nås element med hjälp av indexoperatorn
  - Observera att detta använder hakparenteser precis som för listor

```
colours = ('red', 'green', 'blue')
print(colours[1])
```

```
green
```

- Det är också möjligt att komma åt element som skivor

```
colours = ('red', 'green', 'blue')
print(colours[1:3])
print(colours[-1])
```

```
('green', 'blue')
blue
```

Iterera över en tupel

- En tupel kan itereras över precis som en lista

```
colours = ('red', 'green', 'blue', 'yellow', 'purple', 'orange')

for c in colours:
    print(f'{c} ', end='')

print()

for cc in range(len(colours)):
    print(f'{colours[cc]} ', end='')
```

```
red green blue yellow purple orange
red green blue yellow purple orange
```

Funktioner för tupler

- De vanliga funktionerna för listor är också tillgängliga för tupler
- Liksom `in` och `not in`

```
colours = ('red', 'green', 'blue', 'yellow', 'purple', 'orange')

print(min(colours)) # För att 'b' är tidigt i alfabetet
print(max(colours))
print('purple' in colours)
print('lilac' in colours)
```

```
blue
yellow
True
False
```

Packning och uppackning

- Något som skiljer sig från listor är möjligheten att automatiskt *packa* och *packa upp* tupler
- Det är möjligt att skapa en tupel från kommaseparerade värden (utan parenteser)

```
colours = 'red', 'green', 'blue'
print(colours)
```

```
('red', 'green', 'blue')
```

- Värdena packas automatiskt till en tupel

Uppackning

- *Uppackning* sker när två (eller fler) variabler finns tillgängliga på vänster sida

```
r, g, b = colours
print(r)
print(g)
print(b)
```

```
red
green
blue
```

- Detta händer också för funktioner som returnerar mer än ett värde
- Det är också möjligt att få värdena returnerade som en tupel

Mängder

- Nästa inbyggda datastruktur att titta på är *mängden*
- I en mängd lagras en *samling* element
  - Element lagras i en sekvens, men ordningen är inte viktig i en mängd
- Dessutom är elementen också *distinkta*, vilket betyder att inga dubbletter tillåts
- Till skillnad från tupler är mängder föränderliga
  - Det är möjligt att lägga till, ta bort eller ändra elementen

Skapa mängder

- För att skapa en mängd, använd klammerparenteser `{` och `}`

```
animals = {'Tauntaun', 'Banta', 'Dewback', 'Rancor', 'Sarlacc'}
print(animals)
```

```
{'Banta', 'Tauntaun', 'Dewback', 'Sarlacc', 'Rancor'}
```

- Observera att ordningen de skrivs ut i inte är samma som den ordning de angavs i
- Detta beror på implementeringen av mängder som använder *hash-värden*
  - Speciellt, beräknat, *värde* för varje sträng
- Detta gör mängder mycket effektiva för sökning och borttagning
- Observera också att en tom mängd måste skapas med hjälp av klassen: `s = set()`

Funktioner för mängder

- De vanliga funktionerna `len()`, `min()`, `max()` och `sum()` fungerar som förväntat

```
animals = {'Varactyl', 'Boga', 'Porg'}
print(f'Number of animals: {len(animals)}')
```

```
Number of animals: 3
```

- Det är också möjligt att iterera genom en mängd

```
animals = {'Fathier', 'Vulptex', 'Orbak'}
for a in animals:
    print(f'{a} ', end='')
```

```
Vulptex Fathier Orbak
```

Manipulera mängder

- Det finns flera metoder tillgängliga för mängder
  - `add()` för att lägga till element
  - `remove()` eller `discard()` för att ta bort element
- Observera att metoden `remove()` kommer att ge ett undantag (mer senare) om inget element hittas, medan `discard()` bara inte gör något

```
animals = {'Tauntaun', 'Banta', 'Dewback', 'Rancor', 'Sarlacc'}
animals.add('Wampa')
animals.discard('Rancor')
animals.discard('Gundark')
print(animals)
```

```
{'Banta', 'Tauntaun', 'Dewback', 'Sarlacc', 'Wampa'}
```

Likhet

- Två mängder är samma om innehållet är detsamma
  - Eftersom ordningen är oviktig, behöver de inte vara i samma ordning (som för listor)
- Som oftast testas likhet med `=` (och olikhet med `!=`)

```
animals = {'Tauntaun', 'Banta', 'Dewback', 'Rancor', 'Sarlacc'}
also_animals = {'Sarlacc', 'Tauntaun', 'Dewback', 'Banta', 'Rancor'}
print('Are they the same?', animals == also_animals)
```

```
Are they the same? True
```

Mängdoperationer

- Mängder i Python fungerar som mängder i matematik (därav namnet...🤯)
- Vanliga operationer för mängder är:
  - union för alla element i båda mängderna
  - snitt för de element som finns i båda mängderna
  - differens för de element som finns i första mängden men inte i den andra
  - symmetrisk differens för de element som finns i endera
- Alla operationer har metoder men också operatorer (`|`, `&`, `-` och `^` respektive)

Exempel med metoder

```
tatooine_animals = {'Bantha', 'Dewback', 'Eopie', 'Sarlacc'}
hoth_animals = {'Tauntaun', 'Wampa'}
naboo_animals = {'Fambaa', 'Kaadu', 'Sarlacc'}

all_animals = tatooine_animals.union(hoth_animals, naboo_animals)
print('All animals:', all_animals)

common_animals = tatooine_animals.intersection(naboo_animals)
print('Common animals between Tatooine and Naboo:', common_animals)

unique_to_tatooine = tatooine_animals.difference(naboo_animals)
print('Animals unique to Tatooine:', unique_to_tatooine)

diff_tatooine_hoth = tatooine_animals.symmetric_difference(hoth_animals)
print('Animals on either Tatooine or Hoth, but not both:', diff_tatooine_hoth)
```

```
All animals: {'Dewback', 'Sarlacc', 'Kaadu', 'Wampa', 'Fambaa', 'Tauntaun',
'Eopie', 'Bantha'}
Common animals between Tatooine and Naboo: {'Sarlacc'}
Animals unique to Tatooine: {'Dewback', 'Eopie', 'Bantha'}
Animals on either Tatooine or Hoth, but not both: {'Dewback', 'Sarlacc',
'Wampa', 'Tauntaun', 'Eopie', 'Bantha'}
```

Lexikon (Dictionaries)

- Den sista inbyggda datastrukturen som vi diskuterar är *lexikon*
- Konceptet för ett lexikon är *nyckel-värde-paret*
  - En nyckel: Detta är en unik identifierare för data
  - Ett värde: Detta är den faktiska data som är associerad med nyckeln
- Varje nyckel mappas till *ett* värde och det kan inte finnas dubbletter
- Detta gör dem mycket lika verkliga lexikon 📘 🤯
  - Även om i dessa kan ett sökord leda till flera förklaringar

Egenskaper hos lexikon

- Snabb uppslagning: Du kan snabbt hitta ett värde om du känner till dess nyckel
- Oordnad: Lexikon upprätthåller oftast inte en specifik ordning av objekt
- Föränderliga: Du kan lägga till, ta bort eller ändra nyckel-värde-par efter att ha skapat lexikonet
- Unika nycklar: Varje nyckel i ett lexikon måste vara unik, men värden kan upprepas
- Flexibla värdetyper: Medan nycklar vanligtvis är strängar eller siffror, kan värden vara av vilken datatyp som helst

Skapa lexikon

- Precis som mängder använder lexikon klammerparenteser ({}) för skapande
- Det är här möjligt att använda bara en tom uppsättning klammerparenteser för att skapa ett tomt lexikon utöver en klass

```
computers = {}
more_computers = dict()
```

- Nyckel-värde-par skapas med ett kolon emellan och kan hårdkodas om nödvändigt

```
computers = {'atarist': 'Atari ST, 16-bit from 1985'}
numbers = {1 : 'The first number after zero'}
```

- Till vänster om kolonet är *nyckeln* och till höger är *värdet*

Lägga till och ändra

- Arbete med lexikonet använder variabeln och nyckeln som index mellan hakparenteser (liknande listor)
- Att lägga till ett nytt nyckel-värde-par görs genom att tilldela en oanvänd nyckel till ett värde:

```
computers['c64'] = 'Commodore 64, 8-bit from 1982'
```

- Nycklarna är *oföränderliga* så om samma nyckel används igen kommer värdet att uppdateras

```
computers['atarist'] = 'Atari ST, 16/32-bit from 1985'
```

Hämta värden

- Värden kan hämtas från lexikonet genom att använda nyckeln som index

```
computers = {'atarist': 'Atari ST, 16/32-bit from 1985'}
print(computers['atarist'])
```

```
Atari ST, 16/32-bit from 1985
```

- Om nyckeln finns kommer det associerade värdet att returneras
- Om nyckeln inte finns kommer ett `KeyError` att kastas
  - Hantering av detta kommer att täckas senare

Ta bort värden

- Att ta bort en nyckel, och därmed värdet, görs med en något märklig syntax, med nyckelordet `del`

```
computers = {
    'atarist': 'Atari ST, 16/32-bit from 1985',
    'amiga': 'Commodore Amiga, 16/32-bit from 1986'
}
print(computers)
del computers['amiga']
print(computers)
```

```
{'atarist': 'Atari ST, 16/32-bit from 1985', 'amiga': 'Commodore Amiga,
16/32-bit from 1986'}
{'atarist': 'Atari ST, 16/32-bit from 1985'}
```

- Om nyckeln inte finns kommer Python återigen att kasta ett `KeyError`

Iterera över lexikon

- `for` kan användas för att gå igenom lexikonet där nyckeln används för att iterera över lexikonet

```
computers = {
    'atarist': 'Atari ST, 16/32-bit from 1985',
    'c64': 'Commodore 64, 8-bit from 1982',
    'zx80': 'Sinclair ZX80, 8-bit from 1960',
    'a3010': 'Acorn A3010, 32-bit from 1992'
}

for key in computers:
    print(f'{computers[key]} ', end='')
```

```
Atari ST, 16/32-bit from 1985 Commodore 64, 8-bit from 1982 Sinclair ZX80, 8-
bit from 1960 Acorn A3010, 32-bit from 1992
```

Andra funktioner

- Funktionerna `len()`, `min()`, `max()` fungerar för *nyckeln*

```
computers = {
    'atarist': 'Atari ST, 16/32-bit from 1985',
    'c64': 'Commodore 64, 8-bit from 1982',
    'zx80': 'Sinclair ZX80, 8-bit from 1960',
    'a3010': 'Acorn A3010, 32-bit from 1992'
}

print(len(computers))
print(min(computers))
print(max(computers))
```

```
4
a3010
zx80
```

Hitta nycklar

- Det är möjligt att använda iteration för att se om en specifik nyckel finns i ett lexikon
- Operatorerna `in` och `not in` kan också användas
  - Det finns ingen egentlig hastighetsfördel med att använda `in`, men koden blir kortare

```
computers = {
    'atarist': 'Atari ST, 16/32-bit from 1985',
    'c64': 'Commodore 64, 8-bit from 1982',
    'zx80': 'Sinclair ZX80, 8-bit from 1960',
    'a3010': 'Acorn A3010, 32-bit from 1992'
}

print('Is the C64 here?:', 'c64' in computers)
```

```
Is the C64 here?: True
```

Metoder för lexikon

- Eftersom lexikon är objekt kommer de från en klass med metoder
- Några av metoderna är:
  - `get()`: Returnerar värdet av en angiven nyckel
  - `items()`: Returnerar en lista med lexikonets (nyckel, värde) par
  - `values()`: Returnerar en lista med alla värden i lexikonet
  - `keys()`: Returnerar en lista med lexikonets nycklar
  - `pop()`: Tar bort elementet med den angivna nyckeln
- En fördel med att använda vissa av dessa är att de returnerar `None` snarare än ett undantag

Exempel på metoder

```
computers = {
    'atarist': 'Atari ST, 16/32-bit from 1985',
    'c64': 'Commodore 64, 8-bit from 1982',
    'zx80': 'Sinclair ZX80, 8-bit from 1960',
    'a3010': 'Acorn A3010, 32-bit from 1992'
}

print('Can I get an Amiga?:', computers.get('amiga'))
keys = list(computers.keys())
print(keys)
values = list(computers.values())
print(values)
computers.pop('zx80')
print(computers)
```

```
Can I get an Amiga?: None
['atarist', 'c64', 'zx80', 'a3010']
['Atari ST, 16/32-bit from 1985', 'Commodore 64, 8-bit from 1982', 'Sinclair
ZX80, 8-bit from 1960', 'Acorn A3010, 32-bit from 1992']
{'atarist': 'Atari ST, 16/32-bit from 1985', 'c64': 'Commodore 64, 8-bit from
1982', 'a3010': 'Acorn A3010, 32-bit from 1992'}
```

Sammanfattning av de inbyggda datastrukturerna

- Att hantera data är ofta en del av ett program, så att känna till de inbyggda datastrukturerna är viktigt
- Det är också viktigt att känna till för- och nackdelarna med dem
  - Så att rätt struktur används
- De inbyggda datastrukturerna lista, tupel, mängd och lexikon är alla väl integrerade i Python och effektivt implementerade

Fil I/O

Arbeta med filer

- Ganska ofta räcker det inte att bara hantera data i ett körande program, utan data behöver även lagras permanent
- Logiskt sett finns det två typer av filer:
  - Textfiler som bara innehåller tecken som är direkt läsbara för användaren
  - Binära filer som innehåller sekvenser av bitar som representerar till exempel bilder, musik eller video
- För datorn är alla filer binära, men med Python är det enkelt att läsa och skriva textfiler
- Denna föreläsning kommer endast att täcka textfiler

Sökvägar

- En sökväg är en sekvens av mappar som används för att navigera till filen som ska användas
- Dessa sökvägar kan vara *absoluta* eller *relativa*
- Den absoluta sökvägen är från roten av en enhet till filen
  - På Windows, till exempel: `C:\tmp\test\myfile.txt`
  - På andra plattformar: `/home/tanmsi/Filer/Source/myfile.txt`
- Relativa sökvägar utgår alltid från *nuvarande position*
  - Om du befinner dig i `/home/tanmsi/`, då är den relativa sökvägen till filen ovan `Filer/Source/myfile.txt`

Aktuell arbetskatalog

- Genom att använda biblioteket `os` är det möjligt att ta reda på den aktuella arbetskatalogen

```
import os

current_directory = os.getcwd()
print("Current working directory:", current_directory)
```

```
Current working directory:
/home/tanmsi/Filer/Kurser/2025/VT/4DV121/lectures/lecture3
```

- Observera att detta kommer att returnera en strängrepresentation av sökvägen

Första titten på `os.path`

- Undermodulen `os.path` innehåller flera användbara funktioner för att arbeta med sökvägar och filer
- Dessa arbetar med eller returnerar strängar, precis som den föregående

```
import os.path

absolute_path = os.path.abspath('lecture3.qmd')
relative_path = os.path.relpath('lecture3.qmd')
print(os.getcwd())
print(absolute_path)
print(relative_path)
```

```
/home/tanmsi/Filer/Kurser/2025/VT/4DV121/lectures/lecture3
/home/tanmsi/Filer/Kurser/2025/VT/4DV121/lectures/lecture3/lecture3.qmd
lecture3.qmd
```

Textströmmar

- Textfiler lagras som en sekvens av teckenkodningar
- Vanligtvis används UTF-8 som kodningsformat, vilket innebär att upp till fyra byte kan användas för varje tecken
- Python använder *strömmar* för att läsa och skriva data
- Detta kan visualiseras som en vattenström genom en slang
  - Vatten kommer in från ena sidan, går ut genom den andra
- Ditt program är i mitten, läser in strömmar och skriver ut strömmar
- Detta kontrolleras i Python med hjälp av ett fil-*objekt* som är anslutet till strömmen

Läsa filer

- Läsning (och även skrivning) använder funktionen `open()` som ansluter strömmen till programmet
- Denna funktion tar flera parametrar varav två eller tre parametrar kommer att användas här och returnerar ett filobjekt
- Den första parametern är filnamnet (inklusive sökväg om det behövs) som en sträng
- Den andra parametern är *läget* som filen öppnas i
  - Läsning, skrivning, tillägg eller läsning och skrivning
- Exemplet på nästa bild förutsätter att det finns en fil som heter `shine.txt` i samma katalog som källfilen

Läsa en fil

```
file = open('shine.txt', 'r')
for line in file:
    print(line, end='')
file.close()
```

```
Remember when you were young, you shone like the sun.
Shine on you crazy diamond.
Now there's a look in your eyes, like black holes in the sky.
Shine on you crazy diamond.
You were caught on the crossfire of childhood and stardom,
Blown on the steel breeze.
Come on you target for faraway laughter,
Come on you stranger, you legend, you martyr, and shine!
You reached for the secret too soon, you cried for the moon.
Shine on you crazy diamond.
Threatened by shadows at night, and exposed in the light.
Shine on you crazy diamond.
Well you wore out your welcome with random precision,
Rode on the steel breeze.
Come on you raver, you seer of visions,
```

Ytterligare kommentarer

- Alla öppnade filer bör *stängas* när arbetet är klart
  - Kommer att göras automatiskt när programmet stängs, men i större program kan problem uppstå om filer inte stängs ordentligt
- Windows har ibland svårt att förstå att filer är i UTF-8
- Därför, om du använder Windows, lägg till `encoding='utf8'` till funktionen `open()`

```
file = open('shine.txt', 'r', encoding='utf-8')
```

Mer om läsning

- I exemplet lästes varje rad i textfilen en i taget
- En ”rad” definieras som alla tecken fram till ett nyradstecken
  - I Linux och macOS är tecknet `\n`
  - I Windows är det oftast `\r\n`
- Det är oftast en bra idé att läsa en rad i taget för att bevara minnet
- Det är dock möjligt att läsa hela texten på en gång med hjälp av `readlines()`
  - Detta returnerar en *lista* med alla rader som element

Exempel på att läsa alla rader

- Det här exemplet förutsätter återigen att filen `shine.txt` finns i samma katalog

```
shine = open('shine.txt', 'r')
lyrics = shine.readlines()
print('Lines in the lyrics:', len(lyrics))
print('Line 5:', lyrics[5])
shine.close()
```

```
Lines in the lyrics: 16
Line 5: Blown on the steel breeze.
```

- Denna fil var ganska liten, men hade den varit en större fil är det inte omöjligt att fylla minnet

Exempel på att läsa rad för rad

- Detta exempel läser romanen ”The Fall of the House of Usher” och räknar både tecken och alla gånger ordet ”Usher” finns med

```
usher = open('TheFalloftheHouseofUsher.txt', 'r')
line = usher.readline()
num_of_char = 0
num_of_usher = 0

while line != '': # End of File (not even newline)
    num_of_char += len(line) # add number of characters on line
    if 'Usher' in line: # only count if Usher is in the line
        num_of_usher += line.count('Usher')
    line = usher.readline()
usher.close()

print('Number of characters:', num_of_char)
print('Mentions of "Usher":', num_of_usher)
```

```
Number of characters: 62093
Mentions of "Usher": 26
```

Mer om `os.path`

- Det tidigare nämnda biblioteket `os.path` har flera användbara funktioner
  - `os.path.exists()`: Kontrollera om en sökväg finns
  - `os.path.isfile()`: Kontrollera om en sökväg är en fil
  - `os.path.isdir()`: Kontrollera om en sökväg är en katalog
  - `os.path.getsize()`: Hämta storleken på en fil
- Använd dessa för att alltid kontrollera om en fil finns innan du öppnar den
  - Detsamma gäller för kataloger
- Det finns naturligtvis flera andra funktioner att titta på

Kontrollera filexistens

- Följande kontrollerar om en fil existerar innan den försöker öppna den

```
import os.path

if os.path.isfile('TheFalloftheHouseofUsher.txt'):
    usher = open('TheFalloftheHouseofUsher.txt', 'r')
    first_line = usher.readline()
    print(first_line)
    size = os.path.getsize('TheFalloftheHouseofUsher.txt')
    print(f'Size: {size} bytes')
    usher.close()
else:
    print('The file does not exist.')
```

```
The Project Gutenberg eBook of The Fall of the House of Usher

Size: 63469 bytes
```

Skriva filer

- Det är naturligtvis också möjligt att skriva till filer såväl som att läsa dem
- För skrivning är det enklaste sättet att använda metoden `write()` för filobjektet
- Parametern till `write()` är en sträng eftersom textfiler är just det, rader av strängar
- Det är också viktigt att använda parametern `w` när filen öppnas
  - Eftersom detta öppnar filen för skrivning
- Det är inte garanterat att filen faktiskt skrivs till förrän den är stängd

Enkel skrivning

- Metoden `write()` lägger inte till en nyrad i slutet av varje skrivning
  - Därför viktigt att göra det om rader önskas
  - Enklast att göra genom att avsluta varje skrivning med en `\n`
- Det är också viktigt att notera att om filen redan finns kommer den att skrivas över
  - Så, om innehållet inte får skrivas över, använd först `os.path.isfile()` för att kontrollera
- Ingenting visas på skärmen när man skriver, så glöm inte att berätta för användaren vad som händer

Enkelt exempel på skrivning

- Detta exempel kommer att skriva en fil till disk som innehåller de tre bästa Pink Floyd-låtarna

```
pinkfloyd = open('pinkfloyd.txt', 'w')

print('Writing file...')
pinkfloyd.write('Shine on you crazy diamond, parts 1-5\n')
pinkfloyd.write('Comfortably Numb\n')
pinkfloyd.write('Time\n')

pinkfloyd.close()
print('Done writing file')
```

```
Writing file...
Done writing file
```

Använda `with` för skrivning

- Eftersom det ibland är lite svårt att komma ihåg att alltid stänga filer finns det ytterligare ett sätt att göra det på
- Genom att skapa ett block med `with()` kommer filer att stängas automatiskt när slutet av blocket nås
- Det är möjligt att definiera flera `open()`-anrop i samma `with()` och alla kommer att stängas korrekt
- Användning av `as` efter `open()` definierar variabeln som pekar på den filen

Skriva fem rader med fem smileys

- Programmet kommer att skapa en fil som heter `smilies.txt` med hjälp av `with()`

```
import random

# Randomly select smilies and write them to a file, 5 per row
with open('smilies.txt', 'w') as f:
    for _ in range(5):  # 5 rows
        row = ''
        for _ in range(5): # five smilies per row
            # Range for smilies in Unicode
            smiley_code_point = random.randint(0x1F600, 0x1F64F)
            smiley = chr(smiley_code_point)
            row += smiley
        f.write(f'{row}\n')
```

Läsa smileys med `with`

- Detta program använder också `read()` för att läsa hela filen till en sträng
  - Använd detta med försiktighet eftersom om filen är stor kommer mycket minne att användas

```
with open('smilies.txt', 'r') as f:
    smilies = f.read() # Reads the entire file

for s in smilies:
    print(s, end='')
```

Lägga till data i filer

- Hittills kommer skrivning till en fil att skriva över allt som redan finns där
- Det är möjligt att öppna en fil med a som parameter vilket öppnar den för *tillägg* (appending)
- Om filen finns kommer ytterligare skrivning nu att göras *i slutet* av den öppnade filen
- Dessutom kommer läsning och skrivning av numeriska data också att visas
  - Eftersom textfiler innehåller just text måste alla numeriska data konverteras till text
  - Detta gäller både för skrivning *och* läsning (om du behöver använda det som ett tal)

Exempel på tillägg

```
import random
# Append random temperatures to file
with open('temps.txt', 'a') as file:
    for _ in range(3):
        # random.uniform(a, b) generates a random
        # float between a and b
        temp = round(random.uniform(0, 40), 1)
        file.write(f'{temp}\n')

# Read temperatures and calculate average
total, count = 0, 0
with open('temps.txt', 'r') as file:
    for line in file:
        total += float(line.strip()) # To remove whitespace
        count += 1

avg_temp = round(total / count, 1)
print(f'Average temperature: {avg_temp}°C')
```

```
Average temperature: 19.7°C
```

Sammanfattning

- Fokus för den här delen har varit att arbeta med data
  - Antingen i minnet eller i en fil
- De flesta större program behöver göra båda
  - Spara data mellan körningar av programmet
  - Hantera samma data i minnet under körningen

Felhantering

Hantering av fel

- När ditt program körs och något händer som inte är normalt, kallas detta för en *exceptionell* händelse
- Algoritmen är ofta ganska ”ren” och om all felhantering skulle finnas i den, skulle den bli alldeles för rörig
- I Python finns det sätt att hantera undantag:
  - Kasta (raise) ett undantag i funktionen som har utfört det
  - Funktionen som anropar funktionen kan fånga det
- Det viktigaste är att programmet aldrig bara ska ”dö”

Att inte hantera fel

- När man kör ett program som ”kraschar” får man ett felmeddelande
  - I Jupyter visas det precis under kodcellen i rosa-rött
  - I andra miljöer kommer det upp i terminalen
- En sträng i rött berättar vad felet är
- Om en sträng ges som inmatning när ett heltal förväntas, kommer det att säga `ValueError`
- Detta innebär också att ett undantag har *kastats* och att det är möjligt att göra något åt det

Klockfunktion

- Följande är en funktion för att konvertera mellan 24-timmarsformat och 12-timmarsformat med AM/PM

```
def ampm(hour):
    if 0 < hour < 13:
        return f'{hour} AM'
    else:
        return f'{hour - 12} PM'

def main():
    h = 13
    print(f'In Amerikat they would say {ampm(h)}')

main()
```

```
In Amerikat they would say 1 PM
```

Ansvar för fel

- Hittills har fel hanterats genom att inte skicka fel information till metoden

```
if h > 0 and h < 25:
    print(f'In Amerikat they would say {ampm(h)}')
else:
    print('Error in specifying time!')
```

- Detta leder till mycket att tänka på och det i sin tur gör det skört
- Ett alternativ skulle vara att returnera en sträng från funktionen för att säga att inmatningen var fel
  - Men detta fungerar bara för funktioner som returnerar strängar

Att kasta ett undantag

- En bättre lösning är att se till att funktionen är ansvarig för *sina* delar, och kastar ett undantag

```
def ampm(hour):
    if 0 < hour < 13:
        return f'{hour} AM'
    elif 12 < hour < 25:
        return f'{hour - 12} PM'
    else:
        raise ValueError('Error in specifying time.')
```

- Det är nu möjligt för funktionen att ha full kontroll över vilka intervall den accepterar

Körning

- Om programmet körs nu, och något fel skickas till det, kommer det att visa ett felmeddelande

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
Cell In[2], line 12
      9 def main():
## Several lines removed!
----> 7     raise ValueError('Error in specifying time.')

ValueError: Error in specifying time.
```

- I slutet är det möjligt att läsa felmeddelandet som definierades i funktionen
- Det är fortfarande endast tillgängligt för utvecklaren, men det är bättre än tidigare

Att fånga fel

- Tyvärr kastar vi fortfarande bara felet, så programmet kraschar
- I Python är det möjligt att hantera fel med ett `try - except` block

```
try:
    # kod att kontrollera
except <ExceptionType>
    # kod för att hantera felet
```

- Eftersom undantaget fångas, avslutas inte programmet abrupt utan kan snarare fortsätta efteråt

Exempel på att fånga felet

```
def ampm(hour):
    if 0 < hour < 13:
        return f'{hour} AM'
    elif 12 < hour < 25:
        return f'{hour - 12} PM'
    else:
        raise ValueError('Error in specifying time.')

def main():
    try:
        print(ampm(14))
        print(ampm(9))
        print(ampm(25))
    except ValueError as e:
        print(e)

main()
```

```
2 PM
9 AM
Error in specifying time.
```

Fortsätta köra

- Det är nu möjligt att låta programmet fortsätta köra
- I exemplet avslutades programmet efter den sista utskriften
- Om man istället skapar en iteration runt inmatningen, är det möjligt att bara fortsätta
  - När ett fel upptäcks, skrivs felmeddelandet ut
  - Men körningen fortsätter med nästa instruktion

Exempel med `while`

```
def ampm(hour):
    if 0 < hour < 13:
        return f'{hour} AM'
    elif 12 < hour < 25:
        return f'{hour - 12} PM'
    else:
        raise ValueError('Error in specifying time.')

def main():
    time = -1

    while time != 0:
        time = int(input('What time is it? '))
        if time != 0:
            try:
                print(f'In Amerikat they would say {ampm(time)}')
            except ValueError as e:
                print(e)

main()
```

Utdata

- Observera hur programmet frågar efter inmatning igen efter -12

```
What time is it?  14
In Amerikat they would say 2 PM
What time is it?  9
In Amerikat they would say 9 AM
What time is it?  -12
Error in specifying time.
What time is it?  0
```

Hantering av flera undantag

- Hittills har endast ett undantag hanterats
- Det är möjligt att hantera flera undantag med fler `except`-block
  - Varje block hanterar en specifik typ av fel
- Dessutom är det möjligt att fånga ett allmänt undantag, det vill säga alla andra undantag genom att inte ange vilket undantag som ska fångas
- Undantagen hanteras i den ordning de definieras i `except`-listan, därför placera det allmänna `except` sist

Uppdaterad `main`

- Gör det till en vana att alltid använda en tom `except` i slutet

```
def main():
    time = -1

    while time != 0:
        try:
            time = int(input('What time is it? '))
            if time != 0:
                print(f'In Amerikat they would say {ampm(time)}')
        except ValueError as e:
            print('Value error with message:', e)
        except:
            print('General error')

main()
```

Körning av exemplet

- Observera att i detta fall är det `int` för `input()` som kastar felet
  - Vilket är varför felmeddelandet inte är detsamma som tidigare

```
What time is it?  12
In Amerikat they would say 12 AM
What time is it?  14
In Amerikat they would say 2 PM
What time is it?  twelve
Value error with message: invalid literal for int() with base 10: 'twelve'
What time is it?  9
In Amerikat they would say 9 AM
What time is it?  0
```

Mer om undantag

- `try - except`-blocket har två till alternativ tillgängliga
- Först är det möjligt att använda en `else` i slutet
  - Detta är kod som kommer att köras om *inget* undantag kastas
- Det andra alternativet är att använda `finally` i slutet
  - Kod här kommer *alltid* att köras
  - Ett exempel kan vara att stänga en fil som öppnades i `try`
- Det är också möjligt att *skapa dina egna* undantag
  - Men detta är mer uppenbart när objektorientering har behandlats, så det hoppas över här

Räkna antalet ”a”:n i en fil

```
num_of_a = 0
try:
    usher = open('TheFalloftheHouseofUsher.txt', 'r')
except IOError as e:
    print('Could not open file')
    print('Error message: ', e)
except:
    print('General error')
else:
    line = usher.readline()

    while line != '':
        line = line.lower()
        num_of_a += line.count('a')
        line = usher.readline()
finally:
    usher.close()
print(f'Number of "a":s in the text: {num_of_a}')
```

```
Number of "a":s in the text: 3610
```

Sammanfattning

- Den här föreläsningen har behandlat många, ganska olika, saker
  - Från datastrukturerer till felhantering
- Gemensamt är att det är ganska viktiga saker
- Men, det innebär också att grunderna i Python nu är visade
- Resten handlar mycket om att *använda* Python som en verktygslåda
  - Där vi som programmerare måste veta vilka verktyg som finns där och hur man använder dem