**1. Data Structures & functions**

1. Introduktion af datastrukturere I python : List, Tuple, Set og Dictionary
2. Comprehensions: hvad er det? Og hvorfor bruge det? (Det er færre linjer og er nemmere at parse, memory til hele listen bliver allokeret i starten, fremfor at løbende skulle udvides i runtime, og så slipper man også for at kalde ”.append” mange gange.)
3. Funktioner er ”first class” obejkter der kan passeres som argumenter og returneres
4. Hvis der er tid kan jeg også komme ind på ”multiple assignment”, hvordan man gør det og hvad man kan bruge det til.

Abstract list comprehension for filtrering:

new\_list = [expression for member in iterable (if conditional)]

Abstract list comprehension for ændring af værdier:

new\_list = [expression (if conditional) for member in iterable]

Eksempel på en comprehension med en condition:

l = [i for i in range(1, 10) if i%2 == 0]

eksempel på dict comprehension:

squares = {i: i \* i for i in range(10)}

Hvornår skal man ikke bruge comprehensions? Ved meget store datasæt er det problematisk, da hele listen bliver læst ind i memory. Der skal man i stedet bruge generators.

eksempel på function med inner functions

def parent(num):

def first\_child():

return "Hi, I am Emma"

def second\_child():

return "Call me Liam"

if num == 1:

return first\_child

else:

return second\_child

**2. IO & Context Managers**

Plan for jupyter:

Start med at introducere

1. IO er python modulet der indeholder redskaberne til at læse fra og skrive til filer. (måske kan man snakke lidt om moduler her. Import statement)

2. Når man arbejder med filer er det vigtigt at man at man lukker dem efter man er færdig med at bruge dem for at undgå ressource lækager.

3. Context managers kræver implementering af ”\_\_enter\_\_” og ”\_\_exit\_\_” metoderne. ”\_\_enter\_\_” har til opgave at returnere den ressource man vil arbejde med, hvor ”\_\_exit\_\_” metoden har til ansvar ”at rydde op” efter enter metoden. I tilfældet med filer vil det betyde at lukke filen. Når disse to metoder er implementerede, kan man instantiere context manageren med ” with x(context manager klassen, der implementere ”\_\_enter\_\_” og ”\_\_exit\_\_” protokolerne) as y(navnet man vil give den ressource som manageren returnere).

4. kan alternativt implementeres vha. at importere ”contextmanager” fra biblioteket ”contextlib”, lave en metode i der indeholder en try-finally blok, hvor det man ønsker i ”\_\_enter\_\_” metoden skrives i try delen, og det man ønsker i ”\_\_exit\_\_” metoden skrives i finally delen, og derefter dekorere den metoden med ”context manager”. Decoratoren returnere så en context manager baseret på den try-fianlly blok.

Eksempel på åbning af fil:

with open('testfiles/hrdata.csv', 'r') as f:

Fun fact: man kan have mere end et “open” statement til en enkelt ”with” statement. Man komma separerer dem.

A high-level explanation of the context management protocol is:

The expression is evaluated and should result in an object called a “context manager”. The context manager must have \_\_enter\_\_() and \_\_exit\_\_() methods.

The context manager’s \_\_enter\_\_() method is called. The value returned is assigned to VAR. If no as VAR clause is present, the value is simply discarded.

The code in BLOCK is executed.

If BLOCK raises an exception, the context manager’s \_\_exit\_\_() method is called with three arguments, the exception details (type, value, traceback, the same values returned by sys.exc\_info(), which can also be None if no exception occurred). The method’s return value controls whether an exception is re-raised: any false value re-raises the exception, and True will result in suppressing it. You’ll only rarely want to suppress the exception, because if you do the author of the code containing the ‘with’ statement will never realize anything went wrong.

If BLOCK didn’t raise an exception, the \_\_exit\_\_() method is still called, but type, value, and traceback are all None.

**3. Pythonic OOP**

In my presentation i will focus on encapsulation and how this is achieved through the use of properties. I will give some pratical examples of first public variables and how to get and set them and then how we without making changes in the user code can add properties in our classes and achive encapsulation through these.

Noter:

Overloading af constructor: Virker ikke som normalt grundet måden python interpreteren læser python filen. En ekstra constructor deklaration vil overskrive den første. Måden man overloader på vil være at man putter logik ind i constructoren, som tager højde for at constructoren kan modtage fllere forskellige argumenter af forskellig type.

Inheritance: man nedarver fra en anden klasse ved at sætte typen ind i en parentes i klasse deklarationen, og så skal man selvfølgelig huske at kalde superklassen konstruktør. Det er muligt at nedarve fra mere end en klasse ad gangen, i modsætning til f.eks. java. Funfact: før python 3 skulle man deklarerer ”object” som superklasse, hvis man ville lave et objekt.

Eksemepl på constructor med multiple inheritance:

class FooBar(Foo, Bar):

def \_\_init\_\_(self, bar='bar'):

Foo.\_\_init\_\_(self) # explicit calls without super

Bar.\_\_init\_\_(self, bar)

Attributes: traditionelle getters og setters bliver ikke brugt i python. Fieldsne kan tilgås vha. af normal .notation og kan ændres intiutivt ved ”*objekt*.*attribut= 42”*. Nye attributer kan også tilføjes instanser af objekter, ”on the fly” ved at ”objekt.nytAttribut=43”.

Encapsulation: Der er teknisk set ikke en måde at gøre ting ”private” som i java, men der er en måde at skjule bestemte attributter, ved at skrive \_\_ foran attribut navnet. Dette ”skjuler” navnet, men det er stadig tilgængeligt, men er lidt mere besværligt at kalde. Hvis man gerne vil kontrollere værdierne kan man gøre brug af @property og @*attributnavn.setter*. kontrollen man ønsker at have på sit attribut kan man så putte i *setter* blokken. Denne er den foretrukne måde at opnå implementere en setter, da dette bevare den pythoniske måde at tilgå og ændre variabler.

**\_\_ Protocols \_\_**

1. Protokoller er de metoder der starter med ”\_\_x\_\_” . Deres funktion er typisk er dels at muliggøre at bruge operatorere som ”+”, ”-” eller ”==” på objekter, som muliggør en mere uniform eller intiutiv, fremfor hvordan man normalt ville lave en sammenligning med en ”.equals()” metode, i stedet for for at sammenligne vha. operatoren ”==”, og dels at det muliggøre nogle under the hood ting, som f.eks. ”\_\_enter\_\_” og ”\_\_exit\_\_” der når de implementeres muliggør konstuktionen af en context manager.

**5. Generators**

1. En generator er en function der returnerer et objekt af typen ”lazy itereator”, der kan iterereres igennem som en liste. Fordelen ved ”lazy iteratoren” fremfor en normal liste er at den gør brug af ”call by need” evalution, der muliggør behandling af langt større datasæt, da hele listen ikke bliver indlæst i memory som en normal liste.

Generators vs. Lists: hvilken en skal man bruge? Det kommer an på hvad flaskehalsen er. Hvis memory er flaskehalsen, så bør man bruge en generator, hvis hastighed er vigtigere, så er listen er optimal, da den er hutigere.

Eksempel på generator expression : csv\_gen = (row for row in open(file\_name))

Generator function: # A simple generator function

def my\_gen():

n = 1

print('This is printed first')

# Generator function contains yield statements

yield n

n += 1

print('This is printed second')

yield n

n += 1

print('This is printed at last')

yield n

**6. Decorators**

I Python er funktioner ”first class” objekter, dvs. at de kan returneres og bruges som argumenter. Dette gør at man kan lave decorators der kan ”wrappes” udenom en funktion, for at tilføje funktionalitet til en funktion, uden at ændre på den originale.

For at deklerere en:

def my\_decorator(func):

def wrapper():

print("Something is happening before the function is called.")

func()

print("Something is happening after the function is called.")

return wrapper

Når wrapperen er defineret kan man dekorere funktioner med den ved at skrive @my\_decorator over den funktion som man ønsker dekorereret, så vil det automatisk wrappes rundt funktionen når den bliver kaldt.

Eksempel på decorator der tager højde for funktioners argumenter, introspektion og returværdien fra den indre funktion.

import functools

def do\_twice(func):

@functools.wraps(func)

def wrapper\_do\_twice(\*args, \*\*kwargs):

func(\*args, \*\*kwargs)

return func(\*args, \*\*kwargs)

return wrapper\_do\_twice

Eksempel på plug-in struktur:

import random

PLUGINS = dict()

def register(func):

"""Register a function as a plug-in"""

PLUGINS[func.\_\_name\_\_] = func

return func

@register

def say\_hello(name):

return f"Hello {name}"

@register

def be\_awesome(name):

return f"Yo {name}, together we are the awesomest!"

def randomly\_greet(name):

greeter, greeter\_func = random.choice(list(PLUGINS.items()))

print(f"Using {greeter!r}")

return greeter\_func(name)

I mit eget eksempel kan jeg erstatte random greeting med random band, og greetings med band navvne og sangtekster.