Avanceret Programmering (Uge 37)

Christian Gram Kalhauge (CKL)

Get started – exercise!

See `square.py`

Section 1

Aggregation

Loop Invariant

A statement that is true no matter where you are in a loop.

```
def maxof(numbers, ifempty=0):
    maximum = ifempty

    # Loop Inv: maximum contains the max
    # of all numbers seen so far.
    for number in numbers:
        if number > maximum:
            maximum = number

    return maximum
```

Induction

We prove the loop invariant using induction:

- **0**: *What happens if the list is empty?*
- **n + 1** *Given that you have succeeded so far, so how do we procceed. . .*

Induction: Sum

```
def sumof(numbers):
    # given that the list of numbers is empty
    total = 0

    for number in numbers:
        # Given that the total is the sum of all
        # numbers seen so far, then:
        total = total + number

    return total
```

Induction: Product – Exercise

Write `prodof(numbers)`:

```
.>> prodof([1,2,3])
6
```

1. What is the loop-invariant: *What should be true for every iteration of the loop?*
2. What is the base-case: *What should we return if the list is empty?*
3. What is the IH: *Given that the loop-invariant is true, what should we do to maintain it*

Built-in Aggregators

```
>>> ', '.join(['Hello', 'World!', 'and', 'others'])
'Hello, World!, and, others'

>>> sum([1, 2, 3, 4])
10

>>> max(['Some', 'Loooong', 'String'], key=len)
'Loooong'

>>> min(['Some', 'Loooong', 'String'], key=len)
'Some'
```

Recursion

Recursion is reverse induction. We simply assume that we have fixed the problem and then we go from there:

```
def sumof(numbers):
    # if there are no numbers
    if not numbers: return 0

    # otherwise the sumof number is the sumof
    # the rest plus the head
    head, *rest = numbers
    return head + sumof(rest)
```

Recursion : Product – Exercise

Write `count_positives(numbers)`:

```
.>> count_positive([-12,3,2])
2
```

1. What is the recursive-invariant: *What should be true for calls to countPositive(number) ?*
2. What is the base-case: *What should we return if the list is empty?*
3. What is the IH: *Given that the we can solve the problem for the smaller list, then what should we do to solve the bigger problem?*

Section 2

Functions

Scope

```
>>> var = 1
>>> def globalfn():
...     return var
>>> globalfn()
1
>>> def localfn():
...     var = 2
...     return var
>>> localfn()
2
>>> var
1
```

Scope (cont.)

```
>>> var = 1
>>> def globalfn2():
...     global var
...     var = 2
...     return var
>>> globalfn2()
2
>>> var
2
```

Functional Programming

Functions are just objects, with the `__call__` method

```
>>> def say_hello():
...     print('hello')
>>> say_hello
<function say_hello at ...>
>>> say_hello.__call__()
hello
>>> say_hello()
hello
```

Functional Programming

```
>>> def say_hello():
...     print('hello')
>>> def say_yes():
...     print('yes')
>>> for sayer in [say_hello, say_yes]:
...     sayer()
hello
yes
```

Create a Random Greeting – Exercise

See `greeter.py`.

Closures

```
>>> def counter():
...     counts = 0
...     def inner():
...         nonlocal counts
...         counts += 1
...         return counts
...     return inner
>>> count = counter()
>>> count()
1
>>> count()
2
```

Closures – Assignment

Build a logger creater:

```
from datetime import datetime

def logger(file):
    def inner(msg, level="INFO"):
        ...

log = logger(sys.stdout);
log("Hello, World!")
log("Second log", level="DEBUG")
```

```
17:30 - INFO  - Hello, World!
17:31 - DEBUG - Hello, World!
```

Section 3

Objects

Object

```
>>> class Point:
...     static_field = 0
...     def __init__(this, x, y):
...         # dynamic fields
...         this.x = x;
...         this.y = y
...     def __repr__(this):
...         return f'Point(x={this.x}, y={this.y})'
>>> Point(10, 20)
Point(x=10, y=20)
```

(Multiple) Inheritance

```
>>> class Aviator:
...     def fly(this):
...         print(f"{this} can fly!")

>>> class ClassPrinter:
...     def __repr__(this):
...         return f"{this.__class__.__name__}"

>>> class Bird(Aviator, ClassPrinter):
...     pass

>>> Bird().fly()
Bird can fly!
```

Your first decorator: @classmethod

```
>>> class BetterClassPrinter:
...     @classmethod
...     def __repr__(cls):
...         return f"{cls.__name__}"

>>> class Bird(Aviator, BetterClassPrinter):
...     pass

>>> Bird().fly()
Bird can fly!
```

@staticmethod

```
>>> class PointFactory:
...     @staticmethod
...     def makePoint(x, y):
...         return Point(x, y)

>>> PointFactory.makePoint(10, 10)
Point(x=10, y=10)
```

Private variables

You can use `_varname` to indicate that your variable is private.

Getters and Setters

- Never use getters and setters, use `@property`

```
class Point:
    @property
    def x(this):
        return this._x

    @x.setter
    def x(this, x):
        this._x = x
```