Avanceret Programmering (Uge 38)

Christian Gram Kalhauge (CKL)

Today

- Wrapping up Python
- Next Time

Section 1

Wrapping up Python

Wrapping up

- Closures
- Exceptions
- Duck Typing
- Decorators
- Generators and Co-routines
- Context Managers
- The Zen of Python

Closures

An inner function can close (access) the variables in scope.

Closures – Assignment

Build a logger creator:

```
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!
```

Exceptions - Raising

```
>>> raise "Hello"
Traceback (most recent call last):
...
TypeError: exceptions must derive from BaseException

>>> raise Exception("Something bad happend")
Traceback (most recent call last):
...
Exception: Something bad happend
```

Exception - Excepting

```
>>> try:
...     print(1 / 0)
... except ZeroDivisionError as e:
...     print(e)
... else:
...     print("If not raised")
... finally:
...     print("Always")
division by zero
Always
```

Exceptions

It's better to ask for forgiveness than for permission.

```
>>> x = [1, 2]
```

Exceptions – Don't

```
>>> if len(x) > 3:
...     print(x[3])
... else:
...     print("List too short")
List too short
```

Exceptions – Do

```
>>> try:
...     print(x[3])
... except IndexError:
...     print("List too short")
List too short
```

Duck typing

If it walks like a duck and it quacks like a duck, then it must be a duck.

```
>>> class Pikachu:
...     def use_thunderbolt(self):
...         print("ZAAAP!")
>>> class Charmander:
...     def use_fireball(self):
...         print("WHUSSH!")
>>> class Magnemite:
...     def use_thunderbolt(self):
...         print("ZAAAP!")
```

Ducktyping – Don'ts

```
>>> def attack(pokemon):
...     if isinstance(pokemon, Pikachu):
...         pokemon.use_thunderbolt()
>>> attack(Pikachu())
ZAAAP!
>>> attack(Magnemite())
>>> attack(Charmander())
```

Ducktyping – Do

```
>>> def attack(pikachu):
...     pikachu.use_thunderbolt()
>>> attack(Pikachu())
ZAAAP!
>>> attack(Magnemite())
ZAAAP!
>>> attack(Charmander())
Traceback (most recent call last):
...
AttributeError: 'Charmander' object has no attribute 'use_t
```

Decorators

Any function can be turned into a decorator, by using the `@operator`

```
>>> def add_one(fn):
...     def inner(*args, **kwargs):
...         return fn(*args, **kwargs) + 1
...     return inner

>>> @add_one
... def const(number):
...     return number
>>> const(10)
11
```

Decorators – Wrapper

```
>>> const
<function add_one.<locals>.inner at ...>
>>> from functools import wraps
>>> def add_one(fn):
...     @wraps(fn)
...     def inner(*args, **kwargs):
...         return fn(*args, **kwargs) + 1
...     return inner

>>> @add_one
... def const(number):
...     return number
>>> const
<function const at ...>
```

Decorators – Exercise

The Fibonacci sequence is the list of numbers: [0, 1, 1, 2, 3, 5, 8, 13 . . .]

We can compute it recursively like this:

```
def fib(n):
    if n < 2:
        return n
    else:
        return fib(n - 1) + fib(n - 2)
```

Make it efficient without changing the implementation, using a technique called dynamic programming.

Generators

A generator is an iterable that is not put in a container

```
>>> (x for x in [0, 1])
<generator object <genexpr> at ...>
>>> for y in (x * 2 for x in [0, 1]):
...     print(y)
0
2
```

Co-routines

A Co-routine is a function that `yield` s values instead of returning them.

```
>>> def number(n):
...     print("before")
...     yield n
...     print("after")
>>> number(10)
<generator object number at ...>
>>> for i in number(10):
...     print(i)
before
10
after
```

Co-routines – Example

Look at `primes.py`

Context Managers

Something that can be used in a `with` statement

```
>>> class BeforeAndAfter(object):
...     def __enter__(self):
...         print("before")
...         return "return value"
...     def __exit__(self, type, value, traceback):
...         print("after", type, value, traceback)

>>> with BeforeAndAfter() as x:
...     print("x =", x)
before
x = return value
after None None None

>>> try:
```

Context Managers – Exercise

Write a timer `timer.py`

Formatting: PEP-8

https://www.python.org/dev/peps/pep-0008/

The Zen of Python

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.

The Zen of Python

Readability counts.

Special cases aren't special enough to break the rules.

Although practicality beats purity.

The Zen of Python

Errors should never pass silently.

Unless explicitly silenced.

The Zen of Python

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.

The Zen of Python

Now is better than never.

Although never is often better than *right* now.

The Zen of Python

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.

The Zen of Python

Namespaces are one honking great idea – let's do more of those!

Section 2

Projects