Pokročilé techniky programování

Magie

- There should be one and preferably only one obvious way to do it.
- Magie je něco, co funguje, ačkoli tomu nerozumíme.
- optimalizace a použitelnost vs čitelnost a udržovatelnost
- knihovny, frameworky

Dekorátory a lambda funkce

Speciální (Magic) metody

- Kontex manažery
- Iterátory

Generátory

Dekorátory

- funkce vyšších řádů
- dekorují funkce a třídy za účelem přidání nové funkcionality
- syntax:

```
@decorator  # decorated_function = decorator(decorated_function)
def decorated_function():
  ...
  return
```

In [ ]:

```
import time

def time_it(fn):
    @wraps(fn)
    def inner(*args, **kwargs):
        start = time.time()
        result = fn(*args, **kwargs)
        print(f"Function call took {(time.time() - start) * 1000} ms.")
        return result

    return inner


@time_it
def do_something(count):
    something = 1000

    for i in range(count):
        something = something * 0.9999999

    return something
```

Lambda funkce

- anonymní funkce
- jednoduché, jednořádkové
- neobsahují příkazy (return, =, ...)
- syntax:

```
lambda argument_1, argument_2: pass
```

In [ ]:

```
add = lambda x, y: x + y
add(2, 3)
# 5


f = filter(lambda x: x > 0, [10,-2, 24, -5])
list(f)
# [10, 24]


s = [('Anička', 23), ('Honza', 20), ('Matěj', 22)]
sorted(s, key=lambda x: x[1])
# [('Honza', 20), ('Matěj', 22), ('Anička', 23)]
```

Speciální (Magic) metody

- umožňuje měnit "normální" chování klasických pythonovských komponent:
  - operátorů(+,*, ..., ==, >=, ..., =, ...), klíčových slov (del, class, for, ...), volání funkce, převádění typů apod.
- téměř všechno v Pythonu ale jde předefinovat
- velká moc a velká zodpovědnost
- popsány v dokumentaci: https://docs.python.org/3/reference/datamodel.html#special-method-names (https://docs.python.org/3/reference/datamodel.html#special-method-names)

Magic methods

__getattr__
__setattr__
__delattr__
__getattr__ vs __getattribute__

Object oriented programming in python

- Metody pro předefinování aritmetických operátorů: __add__, __sub__, __mul__, __div__, __floordiv__, __pow__, __matmul__, __lshift__, __rshift__, __or__, __xor__ a varianty s r a i (__radd__, __iadd__, atd.); __neg__, __pos__, __abs__, __invert__.
- Metody pro předefinování porovnávání: __eq__, __ne__, __lt__, __gt__, __le__, __ge__, __hash__.
- Metoda pro zavolání objektu jako funkce: __call__.
- Metody pro funkčnost sekvencí a kontejnerů: __len__, __iter__, __next__, __reversed__; __contains__ pro operátor in.
- Metody pro „hranaté závorky“: __getitem__, __setitem__, __delitem__.
- Převádění na řetězce: __repr__, __str__, __format__.
- Převádění na bool (např. i v if): __bool__.
- Vytváření a rušení objektů: __new__ (konstruktor – vytvoří objekt dané třídy), __init__ (inicializuje objekt dané třídy), __del__ (zavoláno před zrušením objektu).
- Předefinování přístupu k atributům: __getattr__ (zavolá se, pokud se atribut nenajde), __getattribute__ (zavolá se pro každý přístup k atributu), __setattr__, __delattr__, __dir__.
- Implementace context manageru (pro with): __enter__, __exit__.

In [ ]:

```python
from functools import total_ordering

@total_ordering
class Wizard:

    def __init__(self, name, attack, defence):
        self.name = name
        self.attack = attack
        self.defence = defence

    @property
    def rank(self):
        return self.attack + self.defence

    def __str__(self):
        return f"I am mighty fighter called {self.name}!"

    def __call__(self):
        print("Chaaarge!")

    def __eq__(self, other):
        if not isinstance(other, self.__class__):
            raise NotImplemented

        damage_dealt = max(other.attack - self.defence, 0)
        damage_received = max(self.attack - other.defence, 0)

        return damage_dealt == damage_received

    def __gt__(self, other):
        if not isinstance(other, self.__class__):
            raise NotImplemented

        damage_dealt = max(other.attack - self.defence, 0)
        damage_received = max(self.attack - other.defence, 0)
```

```
        return damage_dealt > damage_received

    def __add__(self, other):
        return type(self)(f"Baby of {self.name} and {other.name}",
                          max(self.attack, other.attack),
                          max(self.defence, other.defence))
```

Kontext manažery

- třídy implementující metody __enter__ a __exit__
- používají se v situacích, kdy je potřeba vykonat něco před začátkem a po ukončení akce:
    - práce se soubory, řízení spojení s databází apod.
- syntax pomocí klíčového slova with:

```
with open(filename) as f:  # zde se zavolá __enter__ a výsledek se uloží
do proměnné
  result = f.read()
                           # zde se zavolá __exit__
print(result)
```

Iterátory a iterovatelné objekty

- iterátory jsou třídy implementující metodu `__next__`, která vrací následující prvek nebo vyhazuje vyjímku `StopIteration`
- iterovatelné objekty jsou třídy implementující metodu `__iter__`, která vrací iterátor
    - seznamy, slovníky, ...

In [ ]:

```
it = iter([1, 2, 3])
next(it)
# 1
next(it)
# 2
next(it)
# 3
next(it)
# Traceback (most recent call last):
#   ...
# StopIteration
```

Generátory

- speciální druh iterátoru
- vhodný pro iterování obrovských kolekcí (velkých souborů), které se naráz nevejdou do paměti
- umožňuje iterovat přes nekonečně dlouhé kolekce
- definovány pomocí klíčového slova `yield` nahrazující klíčové slovo `return`

In [ ]:

```
def infinite_increase(number):
    while True:
        number += 1
        yield number

generator = infinite_increase(0)
next(generator)
# 1
next(generator)
# 2
```

In [ ]:

```
import contextlib

@contextlib.contextmanager
def ctx_manager():
    print('Entering')
    yield 123
    print('Exiting')


with ctx_manager() as obj:
    print('Inside context, with', obj)

########################################

def read_files(filenames):
    for filename in filenames:
        with open(filename) as f:
            yield from iter(f)
```

Komprehenze

- kompaktní zápis inicializace iterovatelných objektů a generátorů (jednoduché závorky)
- syntax:

```
[výraz for prvek in iterovatelný_objekt if podmínka]
```

In [ ]:

```
leap_years = [year for year in range(1900, 2020) if year % 4 == 0]

matrix = [[x for x in range(10)] for _ in range(10)]

object_generator = (get_object(id) for id in ids )
```