Zpracování a vizualizace dat v prostředí Python

Pokročilé aspekty jazyka Python

Zdeněk VAŠÍČEK
Fakulta Informačních Technologií, Vysoké Učení Technické v Brně
Brno, Czech Republic
vasicek@fit.vutbr.cz

Obsah

- Uzávěr (Closure) …
  - deklarace, vlastnosti, využití
- Dekorátor (Decorator) …
  - vlastnosti, řetězení, deklarace vlastního dekorátoru pomocí uzávěru a tříd
- Iterátory a generátory (Iterators and Generators) …
  - vlastnosti, řetězení, základní podpora pro práci s iterátory
  - deklarace vlastního iterátoru pomocí tříd, generátorů a generátorových výrazů
  - vlastnosti generátorů a jejich použití

Uzávěr

Closure

Prerekvizity » deklarace a vlastnosti funkcí, rozsah platnosti jmen (scope), **anonymní funkce**

Uzávěr (Closure)

- Funkce první kategorie (first-class function) umožňují vytvářet tzv. uzávěr (Closure) využívající schopnosti definovat zanořené funkce

```
def print_msg():
    # outer enclosing function
    def printer(msg):
        # nested function
        print(msg)
    return printer  # returns the nested function
```

```
f = print_msg()
f("Hello") #Hello

print("not defined" if locals().get('print_msg', globals().get('print_msg')) is None
                    else "defined")
del print_msg
print("not defined" if locals().get('print_msg', globals().get('print_msg')) is None
                    else "defined")

f("Hello") #Hello
```

Uzávěr (Closure)

- Uzávěr je technikou pro implementaci vazby názvů s lexikálním rozsahem
  - Implementačně jde o nějaký záznam ukládající funkci společně s jejím prostředím (rozsahem), tj. stavem přiřazení objektů jménům dostupným funkci (zachycené proměnné, cell objects).
  - Na rozdíl od jednoduché funkce uzávěr umožňuje funkci přístup k zachyceným proměnným, i když je funkce vyvolána mimo jejich rozsah.

- Uzávěr je libovolná funkce, která v sobě obsahuje kontext z jiné funkce, v rámci které byla původně definovaná, i když vykonávání této nadřazené funkce již skončilo.

```
def print_msg(msg):
    # outer enclosing function
    def printer():
        # nested function
        print(msg)  # non-local variable
    return printer  # returns the nested function
```

```
f = print_msg("Hello")
f() #Hello
del print_msg
f() #Hello
```

Vnitřní stav uzávěru

- Pokud je funkce uzávěrem, pak existuje atribut *__closure__*, který vrací n-tici obsahující hodnoty proměnných zachycených proměnných reprezentující vnitřní stav uzávěru (tzv. cell objects).

```
def pretty_print(fmt, n):
    def printer(x):
        print(fmt % (x/(10**n)))
    return printer

pp = pretty_print("%.1f k", 3)
pp(1234)
```

```
pp.__code__.co_freevars # ('fmt', 'n')
pp.__closure__ #(<cell at 0x0000026E6EA1B880: str object at 0x00007FFD72A706E0>,
                 <cell at 0x0000026E6EA1B250: int object at 0x0000026E6EB766F0>)
pp.__closure__[0].cell_contents #'%.1f k'
pp.__closure__[1].cell_contents #3
```

Vnitřní stav uzávěru není neměnný

- Vnitřní stav uzávěru je sice uložen v n-tici avšak je možné jej v kódu aktualizovat
  - aktualizuje se prvek n-tice __closure_, který je instancí typu cell

```
def autoincrement(step):
    n = -step
    def incr():
        nonlocal n
        n += step
        return n
    return incr

c = autoincrement(10)
print(c.__closure__)
print(c())
print(c.__closure__)
print(c())
print(c.__closure__)
print(c())
```

? Lze změnit vnitřní stav uzávěru i mimo uzávěr?

```
type(c.__closure__[0])
```

Využití uzávěru

- Vlastnosti uzávěru / typické použití
  - dovolí se vyhnout použití globálních hodnot
  - poskytuje mechanismus pro skrytí dat (data hiding)
  - umožňuje OOP (alternativa malým třídám s několika málo atributy, případně metodami) – viz Javascript

- Příklad: Realizace Factory (viz návrhové vzory) pomocí uzávěru

```
def make_multiplier_of(n):
    def multiplier(x):
        return x * n
    return multiplier


# Multiplier of 3
times3 = make_multiplier_of(3)

# Multiplier of 5
times5 = make_multiplier_of(5)
```

Uzávěr nebo třída?

- Příklad: nahrazení třídy s jednou metodou uzávěrem
  - třída

```
class SourceTemplate:
    def __init__(self, url):
        self.url = url
    def load(self, **kwargs):
        return requests.get(self.url.format_map(kwargs))
```

  - OOP pomocí uzávěru

```
def sourcetemplete(url):
    def load(**kwargs):
        return requests.get(url.format_map(kwargs))
    return load
```

- Kdy použít uzávěr místo třídy?
  - potřebujeme uložit (pamatovat si) nějaký kontext
  - třída by obsahovala kromě inicializeru __init__ pouze jednu metodu

Problém s cykly a lambda funkcemi (nahrazení uzávěru běžnou funkcí)

- Proměnná cyklu je vytvořena nikoliv v rozsahu platnosti for cyklu, ale v okolním rozsahu (typické nejen pro Python) a je zapotřebí dávat pozor, pokud proměnnou používáme v anonymní funkci
  - hodnota `i` z konkrétní iterace není svázána s anonymní funkcí (`i` je reference na proměnnou cyklu nikoliv na hodnotu)

```
powers = [lambda x: x**i for i in range(10)]
```

=

```
powers = []
for i in range(10):
    def func(x):
        return x**i
    powers.append(func)
```

  - hodnotu je zapotřebí uchovat v proměnné, která je lokální lambda funkci (tradičně přes uzávěr, nebo alternativně přes výchozí hodnotu, která se určuje v době deklarace funkce)

```
for i in range(10):
    def make_func(j):
        def func(x):
            return x**j
        return func
    powers.append(make_func(i))
```

=

```
for i in range(10):
    def func(x, j=i):
        return x**j
    powers.append(func)
```

Dekorátor

Decorator

Prerekvizity » deklarace a vlastnosti funkcí, rozsah platnosti jmen (scope), anonymní funkce, volání funkcí s proměnným počtem parametrů, třídy

Dekorátor funkcí (decorator)

- Dekorátor ≡ funkce vyššího řádu umožňující modifikovat funkcionalitu jiné funkce
  - dekorátor bere funkci v argumentu, dodává nějakou novou funkcionalitu a vrací ji (je uzávěrem)
- Příklad

```
def decorate(func):
    def inner(x):
        print("> I got decorated")
        y = func(x)
        print("> Leave")
        return y
    return inner
```

```
def myfun():
    print("function")
pretty = decorate(myfun)
pretty()
```

? K čemu lze dekorátor využít?

- Syntaxe zjednodušující deklaraci dekorátorů (tzv. syntax suggar)

```
def myfun():
    print("function")
myfun = decorate(myfun)
```

=

```
@decorate
def myfun():
    print("function")
```

Dekorátor funkcí s argumenty

- Při tvorbě dekorátorů je dobré pamatovat na to, do jaké míry by měly být univerzální.
  - Na jedné straně můžeme dělat specializované dekorátory funkcí s fixním počtem argumentů

```
def decorate(func):
    def inner(arg):
        func(arg)
    return inner
```

  - Na druhé straně můžeme předávat všechny argumenty pomocí *args a **kwargs

```
def decorate(func):
    def inner(*args, **kwargs):
        func(*args, **kwargs)
    return inner
```

- Příklad univerzálního dekorátoru

```
@deprecated("use another function")
def mac(a,b,c):
    return a*b + c
```

? Jak by se takový dekorátor definoval?

Dekorátor s argumentem

- Deklarátor s argumentem je možné definovat na základě následující znalosti
  - Deklarace

```
@decorator
def func(*args, **kwargs):
    pass
```

je pouze zkratkou k

```
func = decorator(func)
```

  - Tudíž deklarace

```
@decorator_with_args(arg)
def func(*args, **kwargs):
    pass
```

je zkratkou k

```
func = decorator_with_args(arg)(func)
```

? Jak nadefinovat dekorátor multiply(n), který vynásobí výsledek dekorované funkce?

Řetězení dekorátorů

- Dekorátory je možné libovolně řetězit, neboť
  - výstupem dekorátoru je dekorovaná funkce
  - vstupem dekorátoru je funkce

```
@decor1
@decor2(arg)
def myfun():
    print("function")
```

? Jaké bude pořadí vyhodnocování pokud funkci **zavoláme**?

- Příklad
  - Protože záleží na pořadí, dáváme první funkce umožňující eliminovat další volání …

```
@app.route("/api/resource")
@login_required
def resource_handler():
    pass
```

vs.

```
@login_required
@app.route("/api/resource")
def resource_handler():
    pass
```

Dekorátor pomocí třídy využívající vlastnosti callable

- Využijeme-li faktu, že dekorátor je pouze funkce vyššího řádu (tj. tzv. callable), můžete dekorátor deklarovat také s využitím tříd

```
class MyDeco(object):
    def __init__(self,arg):
        self.arg = arg

    def __call__(self, original_func):
        def inner(*args, **kwargs):
            original_func(*(self.arg, *args),**kwargs)
        return inner
```

```
@MyDeco(123)
def funcarg(darg,x):
    print(darg,x)

funcarg(1)
```

```
@MyDeco(123)
def func(darg):
    print(darg)

func()
```

Dokumentační řetězce

- Dokumentační řetězec (docstring)
  - řetězcová konstanta, uvedená na místě prvního příkazu funkce / metody / modulu
  - spojuje programový kód s jeho komentářem

Sphinx syntax

```
def function_with_types_in_docstring(param1, param2):
    """Example function with types documented in the docstring.

    Args:
        param1 (int): The first parameter.
        param2 (str): The second parameter.

    Returns:
        bool: The return value. True for success, False otherwise.
    """
```

- Použití
  - automatické generování dokumentace – viz např. rozšířený Sphinx
  - automatické testování kódu – viz integrovaný modul doctest (ale vhodnější bude unittest nebo jiné)

Dokumentační řetězce

- Příklad využívající možnosti modulu doctest

```
def is_palindrome(s):
    """
    Funkce vracejici True, je-li retezec s palindrom

    >>> is_palindrome('abba')
    True
    >>> is_palindrome('abab')
    False
    >>> is_palindrome('tenet')
    True
    >>> is_palindrome('deed')
    False
    """
    return ...

doctest.run_docstring_examples(is_palindrome, globals())
```

```
Failed example:
     is_palindrome('deed')
Expected: False
Got: True
```

? Jak by vypadala nejjednodušší implementace?

nebo

```
if __name__ == '__main__':
    import doctest
    display(doctest.testmod())
```

Dokumentační řetězce v případě dekorátoru (uzávěru)

- Není-li to ošetřeno, ztratí se nám dokumentační řetězec původní funkce

```
def decorate(func):
    def inner():
        return func()
    return  inner

@decorate
def myfun():
    """My function docstring"""
    return "A"

help(myfun)
```

- Příklad použití

```
@deprecated("use another function")
def mac(a,b,c):
    """multiply and add"""
    return a*b + c
```

- Je možné napravit, pokud využijeme funkci wraps z modulu functools

```
import functools


def decorate(func):
    @functools.wraps(func)
    def inner():
        return func()
    return  inner


@decorate
def myfun():
    """My function docstring"""
    return "A"

help(myfun)
```

Iterátory a generátory

Iterators and generators

Iterovatelné objekty a iterátory

- Iteraci zajišťuje tzv. iterační protokol (iterator protocol) rozlišující dva druhy objektů:
  - iterovatelné objekty (iterables) a
  - iterátory (iterators).

- Iterovatelný objekt (implementuje Iterable protocol)
  - objekt, který implementuje metodu **__iter__** vracející objekt typu iterátor
  - většina vestavěných objektů (list, str, tuple,…) jsou iterovatelné objekty

? Je range iterátor?

```
hasattr(str, '__iter__') # True
```

```
hasattr(tuple, '__iter__') # True
```

- Iterátor (implementuje Iterator protocol)
  - objekt, který implementuje metodu **__next__**, která buď vrací další prvek iterovaného objektu nebo způsobí výjimku **StopIteration** v případě, že už další prvek neexistuje
  - objekt, který zajišťuje iteraci. Typická implementace si pamatuje původní iterovatelný objekt a aktuální pozici (malá paměťová stopa).

- Iterátory představují základní techniku nezbytnou pro podporu funkcionální přístupu k programování

Iterovatelné objekty a iterátory

- Princip
  - iterátor je možné získat voláním funkce **iter(obj)**, během kterého dojde současně k inicializaci iterátoru

```
it = iter(str("X"))      # aka it = str("X").__iter__()

hasattr(str, '__next__') # False
hasattr(it, '__next__')  # True
```

  - první a další prvek je možné získat voláním funkce **next(it)**, tzn. jde o tzv. lazy evaluation techniku

```
next(it) # 'X' aka it = it.__next__()
next(it) # exception StopIteration
```

- Použití

```
iterator = iter(collection)
while True:
    try:
        x = next(iterator)
    except StopIteration:
        break
    …
```

=

```
for x in collection:
    …
```

Odložené vyhodnocování (lazy evaluation)

- Výhody odloženého vyhodnocování:
  - Nízká paměťové náročnost

```
import sys, itertools

lots_of_fours = itertools.repeat(4, times=100_000_000)
sys.getsizeof(lots_of_fours) # 48 B

lots_of_fours = [4] * 100_000_000
sys.getsizeof(lots_of_fours) # 800000056 B
```

  - Možnost procházet (iterovat skrze) sekvence předem neznámé délky

```
for line in open("99.ipynb","rb"):
    print(line)
```

```
for n in itertools.count():
    if n==10: break
    print(n)
```

? Jak zjistit, je-li open iterátor?

Vlastnosti iterátoru

- Iterátor je iterovatelný (iterator is an iterable)

```
iterator = iter([1,2,3])
iterator #<list_iterator at 0x26e6fde35b0>
iter(iterator) #<list_iterator at 0x26e6fde35b0>
```

  - důsledek: lze jej předat funkci for

```
for i in iterator:
    print(i)
```

- Iterátor spotřebovává iterovatelný objekt (iterator is a consumable iterable)
  - Některé objekty jsou stavové a lze je procházej jen jednou (soubory, sokety, …)
  - Iterátory jsou v principu consumables protože je nelze zresetovat
- Iterátor je kontejner (viz container protocol)
  - kontejner je objekt, nad kterým je možné volat operátor **in** (implementuje metodu __contains__)

```
iterator = iter([1,2,3])
1 in iterator
```

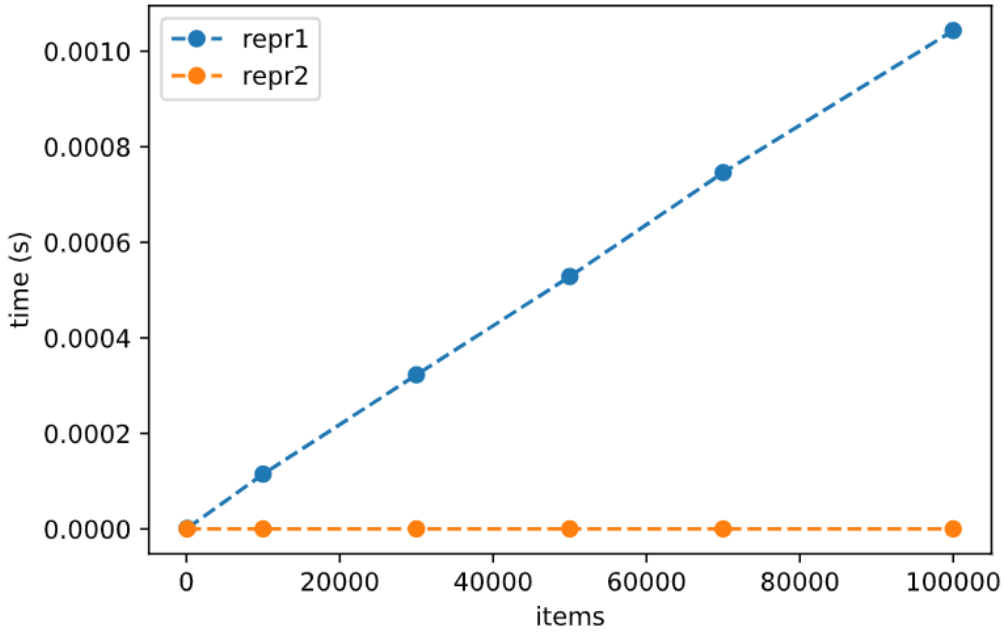
```
"Alex\n" in open("names.txt","r")
```

Pozor na skryté volání iterátoru

- Příklad:

```
def repr1(N):
    return [i for i in range(N)]

def repr2(N):
    return {i for i in range(N)}
```




```
N = 1000000

x = repr1(N)
%timeit N in x
11 ms ± 565 µs per loop (mean ± std. dev. of 7 runs, 100 loops each)

y = repr2(N)
%timeit N in y
67 ns ± 2.7 ns per loop (mean ± std. dev. of 7 runs, 10000000 loops each)
```

– vynucené použití iterátoru

```
%timeit N in iter(y)
13 ms ± 1.1 ms per loop (mean ± std. dev. of 7 runs, 100 loops each)
```

? Jak by se choval slovník?

Řetězení iterátorů (processing pipeline)

- Iterátory lze snadno řetězit
  - poznámka: ne nutně musí každý článek řetězce vracet všechny položky ze vstupu

```
chain = negated(squared(integers()))
```

```
chain1 = integers()  #chain1 ma definovanou metodu __iter__ i __next__
chain2 = squared(chain1) #chain2 ma definovanou metodu __iter__ i __next__
chain = negated(chain2)  #chain ma definovanou metodu __iter__ i __next__
```

- Použití je nativní
  - přímé volání

```
it = iter(chain)
next(it) #-1
next(it) #-4
```

  - for cyklus

```
for i in negated(squared(range(1,10))):
    print(i)
```

Podpora iterátorů v Python

- Vestavěné funkce vracející iterátor
  - map, filter, reduce, zip, …

```
>>> l1, l2 = ['a', 'b'], ['c', 'd']
>>> list(zip(l1,l2))
[('a', 'c'), ('b', 'd')]
```

- Modul itertools implementuje řadu nástrojů pro práci s iterátory
  - chain (řetězení za sebe), compress (filtrování položek), groupby (shlukování), starmap (* map), kombinatorické iterátory product, permutations, combinations, …

```
>>> import itertools
>>> l1, l2, l3 = ['a', 'b'], ['c', 'd', 'e'], "fg"
>>> chained = itertools.chain(l1, l2, l3)
<itertools.chain at 0x27cb4b54190>
>>> list( chained )
['a', 'b', 'c', 'd', 'e', 'f', 'g']
>>> list( itertools.product('ABCD', repeat=2) )
['AA','AB','AC','AD','BA','BB','BC','BD','CA','CB','CC','CD','DA','DB','DC','DD']
```

  - více informací viz dokumentace docs.python.org/3/library/itertools.html

Vestavěné operace pro práci s iterovatelnými datovými typy (iterables)

- Funkce map

```
map(func: Callable[[_T1], _S], iter1: Iterable[_T1], /) -> Iterator[_S]
```

  - Map aplikuje unární funkci na každý element sekvence a vrací novou sekvenci výsledků ve stejném pořadí

```
map(lambda x: x.strip().capitalize(), ['aLex','Bara ',' barbora','emIl','Aneta'])
```

- Funkce filter

```
filter(func: None, iterable: Iterable[Optional[_T]], /) -> Iterator[_T]
```

  - Filter testuje každý element unárním predikátem. Elementy, které vyhovují jsou ponechána a ostatní odstraněny

```
filter(lambda x: x.startswith('A'), ['Alex','Bara','Barbora','Emil','Aneta'])
```

- Funkce reduce

```
functools.reduce(func: Callable[[_T, _S], _T], sequence: Iterable[_S], initial: _T) -> _T
```

  - Reduce kombinuje elementy sekvence dohromady s pomocí binární operace a vrací jednu hodnotu.
  - Bere v potaz inicializační hodnotu, kterou inicializuje redukci nebo vrátí zpět u prázdné sekvence.

```
import operator
functools.reduce(operator.add, [1,2,3,4,5])  # ((((1+2)+3)+4)+5)
```

```
functools.reduce(lambda x, y: 2*x + y, [1,0,0,0])  #bin2hex
```

Využití vestavěných funkcí pro implementaci základních operací nad iterables

- Příklad deklarace řetězeného iterátoru využívající built-in funkci **map**

```
def integers():
    return itertools.count(start=1)
```

count(start=0, step=1) -> Iterator

```
def squared(it):
    return map(lambda x: x**2, it)
```

```
def negated(it):
    return map(lambda x: -x, it)
```

map(func: Callable[[_T1], _S], iter1: Iterable[_T1], /) -> Iterator[_S]

- Použití

```
for i in negated(squared(range(1,10))):
    print(i)
```

Deklarace iterátoru – varianta využívající OOP (třídy)

- Postačující podmínka je definovat metodu __iter__ a __next__

```
class EvenNumbers:
  def __init__(self, maxval=10):
    self.max = maxval

  def __iter__(self):
    self.a = 0
    return self

  def __next__(self):
    if self.a <= self.max:
      x = self.a
      self.a += 2
      return x
    else:
      raise StopIteration
```

```
for x in EvenNumbers(20):
  print(x)
```

Deklarace iterátoru – varianta využívající generátorových funkcí

- Generátorová funkce (generator function) je funkce, která obsahuje klíčové slovo **yield**

```
def generate():
    print('A')
    yield 1
    print('B')
    yield 2
    print('C')
```

  - zavoláním takové funkce dostaneme iterátor

```
>>> it = generate()
<generator object generate at 0x0000026E2FD3CC80>
```

  - voláním next() se funkce provede až po první yield a jako návratová hodnota se vrátí objekt uvedený za klíčovým slovem yield

```
>>> next(it)
1
```

  - další volání next() pokračuje funkce od předchozího bodu přerušení až do dalšího yield, atd.
  - narazí-li se během vykonávání na konec funkce (přirozeně nebo return), pak je generována výjimka StopIteration s hodnotou value identickou jako objekt následovaný za klíčovým slovem return

Deklarace iterátoru s využitím generátorové funkce

- Deklarace iterátoru pomocí generátorové funkce je typicky přímočařejší

```
def squared(seq):
    for i in seq:
        yield i * i

def negated(seq):
    for i in seq:
        yield -i
```

```
def evennumbers(maxval=10):
    i = 0
    while i <= maxval:
      yield i
      i += 2
```

```
def evennumbers(maxval=10):
    if maxval<0: return "Invalid value"
    for i in range(0, maxval+1, 2):
        yield i
```

Deklarace iterátoru – varianta využívajících generátorových výrazů

- PEP-289 zavádí tzv. generátorovou notaci (generator expression)
  - umožňuje zkrátit zápis generátorů pomocí syntaxe podobné list comprehension (… for … in …)
  - výrazově slabší prostředek oproti generátorům
- Příklady:
  - Funkce map

```
(x.strip().capitalize() for x in ['aLex','Bara ',' barbora','emIl','Aneta'])
```

  - Funkce filter

```
( x for x in ['aLex','Bara ',' barbora','emIl','Aneta'] if x.startswith('A'))
```

  - Funkce reduce

```
sum( a for a in [1,2,3,4,5] )
```

```
sum( x*2**i for i, x in enumerate(reversed([1,0,0,0])) )
```

? Je poslední ukázka stejně efektivní jako s využitím reduce?

Generátory lze řetězit stejně jako iterátory

- Příklad implementace tzv. data processing pipeline
  - více jednoduchých obecných funkcí vs. jedna složitá jednoúčelová funkce

```
def readfiles(filenames):
    for f in filenames:
        for line in open(f):
            yield line

def grep(pattern, lines):
    return (line for line in lines if pattern in line)

def printlines(lines):
    for line in lines:
        print(line, end="")

def main_pipeline(pattern, filenames):
    lines = readfiles(filenames)
    lines = grep(pattern, lines)
    printlines(lines)
```

Vlastnosti generátorů

- Generátor má metodu close(), která se volá pokud konzument přestal generátor využívat
  - pokud nás tato situace zajímá, je nutné obsloužit výjimku GeneratorExit (pozor na blokování v případě zachytávání výjimek v generátoru!)
- Pomocí konstrukce **yield from** lze delegovat generátor na jiný generátor
  - po ukončení generátoru na který jsme generování delegovali obsahuje yield from hodnotu vrácenou ve výjimce StopIteration

```
def powerof2(limit):
    count, val = 0, 1
    while val < limit:
        yield val
        count += 1
        val <<= 1
    return count

def powerof2_annotated(n):
    bits = (yield from powerof2(n))
    print('Bits needed: {}'.format(bits))

for value in powerof2_annotated(10):
    print(value)
```

Obousměrná komunikace generátorů

- Generátor umožňuje na rozdíl od iterátoru obousměrnou komunikaci
  - pomocí metody **send**() lze zaslat do generátoru hodnotu, kterou nabývá po návratu **yield**
  - volání send() zapříčiní pokračování vykonávání kódu a jeho návratovou hodnotou je hodnota za yield

```
def generator():
    num = (yield 0)
    yield num * 10
    yield num * 100

>>> it = generator()
>>> next(it)
0
>>> it.send(2)
20
```

? Lze použít it.send() před voláním next(it)?

Obousměrná komunikace generátorů

- Generátor umožňuje na rozdíl od iterátoru obousměrnou komunikaci
  - pomocí metody **throw**() lze do generátoru zaslat výjimku (můžeme ukončit generátor předčasně)

```
def generator():
    try:
        yield
    except Exception as e:
        pass
    yield 123

>>> it = generator()
>>> next(it)
None
>>> it.throw(ValueError())
123
```

? Co by se stalo, pokud bychom místo Exception odchytávali BaseException?

Další využití generátorů

- Vlastnost přerušit provádění funkce lze využít i pro implementaci kontextové manažeru

```
import contextlib

@contextlib.contextmanager
def ctx_manager():
    print('enter')
    yield "obj_value"
    print('exit')

with ctx_manager() as obj:
    pass
```

Další témata ke studiu

- Speciální témata týkající se tříd
  - sloty dovolující zamezit dynamické alokaci atributů, deskriptory umožňující deklaraci počítaných atributů
- Podpora pro asynchronní zpracování dat
  - příkaz async/await, asynchronní iterátory (viz PEP-492), generátory a comprehensions (viz PEP-530)
- Metaprogramování
  - metatřídy (viz type), protokoly, abstraktní bázové třídy (viz abc)