Lambda

Wyrażenie:

```
f=lambda x: x+3
```

jest równoważne zdefiniowaniu funkcji:

```
def f(x):
    return x+3
```

W obu przypadkach tworzymy obiekt typu 'function' (można sprawdzić type(f)).

W obu przypadkach wywołanie funkcji wygląda dokładnie w taki sam sposób, czyli np. f(7), a że funkcja zwraca wartość to pewnie raczej a=f(7).

W wyrażeniu lambda po dwukropku wpisuje się wyrażenie, którego wartość jest przez funkcję zwracana. Nie ma możliwości tworzenia bloku kodu.

Lambda

Wszystko co było mówione odnośnie sposobu przekazywania parametrów do funkcji jest prawdą także w przypadku wyrażeń lambda. Poprawne są zatem, np.:

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

```
f=lambda *x: sum(x)
f(*range(10))                   #45
```

```
f=lambda *x,y=1: y+sum(x)
f(*range(10))                   #46
```

Funkcje wbudowane

```
import builtins
dir(builtins)
```

- print
- dir, help
- range
- type, isinstance, id
- len
- ascii i repr
- tuple, list, dict, set, frozenset
- abs, pow, sum, max, min, round, divmod
- enumerate
- input
- eval
- exit i quit - należy z nich korzystać TYLKO, jeśli korzystamy bezpośrednio z interpretera. W innym przypadku do wymuszenia zakończenia programu należy korzystać z funkcji sys.exit.

Funkcje

- int, float, str
- bin

```
bin(13)                 #'0b1101'
int('0b1101',2)         #13
```

- bool

```
bool([])                #False
bool([1])               #True
```

- chr i ord

```
ord('a')                #97
chr(97)                 #'a'
```

- complex

```
complex(13)             #(13+0j)
complex(13,3)           #(13+3j)
```

- oct i hex

```
oct(13)                 #'0o15'
int('0o15',8)           #13
hex(13)                 #'0xd'
int('0xd',16)           #13
```

Funkcje wbudowane można nadpisać!

Czyli jakkolwiek można napisać:

```
print=6
```

to jednak nic już wtedy w programie nie wypiszemy na ekran:

```
print('ala')                #TypeError: 'int' object is not callable
```

Skoro o 'callable' mowa, to jest taka funkcja wbudowana:

```
callable(print)             #False, jeśli zrobiliśmy to co powyżej
```

Możemy zawsze usunąć obiekt z przestrzeni lokalnej:

```
del print
```

i wtedy odzyskamy funkcję print:

```
callable(print)             #True
print('ala')                #działa!
```

Funkcje

Elementy zawarte w lokalnej i globalnej przestrzeni nazw można pobrać w postaci słownika odpowiednio funkcjami wbudowanymi locals i globals:

```
x=2
def fun(a,b):
    c=a+b
    print(locals())                                  #{'a': 4, 'b': 5, 'c': 9}
    print(globals())
                #{'x': 2, 'fun': <function fun at 0x7f95c24d85f0>}*

fun(4,5)
```

* tak, tu trochę elementów usunęłam ...

Funkcje

Dowolny obiekt iterowalny można przekazać jako parametr do funkcji `sorted`, `reversed` czy `iter`, w ostatnim przypadku na wyjście otrzymujemy iterator, po którym możemy jak sama nazwa wskazuje iterować, ale też kolejne elementy można pobrać korzystając z funkcji `next`.

```
ob=random.sample(string.ascii_uppercase, 10)
print(ob)      #['W', 'B', 'S', 'N', 'E', 'O', 'G', 'Z', 'F', 'A']

sob=[el for el in sorted(ob)]
print(ob)      #['W', 'B', 'S', 'N', 'E', 'O', 'G', 'Z', 'F', 'A']
print(sob)     #['A', 'B', 'E', 'F', 'G', 'N', 'O', 'S', 'W', 'Z']

srob=[el for el in sorted(ob, reverse=True)]
print(ob)      #['W', 'B', 'S', 'N', 'E', 'O', 'G', 'Z', 'F', 'A']
print(srob)    #['Z', 'W', 'S', 'O', 'N', 'G', 'F', 'E', 'B', 'A']

rob=[el for el in reversed(ob)]
print(ob)      #['W', 'B', 'S', 'N', 'E', 'O', 'G', 'Z', 'F', 'A']
print(rob)     #['A', 'F', 'Z', 'G', 'O', 'E', 'N', 'S', 'B', 'W']
```

Funkcje

Przydatnymi funkcjami są także: all i any. Obie jako parametr przyjmują obiekt iterowalny i zwracają wartość logiczną.

- all zwraca wartość prawdy, jeśli wszystkie elementy obiektu mają wartość logiczną prawdy
- any, jeśli którykolwiek

```
all(range(10))        #False
all(range(2,10))      #True
any(range(10))        #True
```

map

`map`(func, *iterables) - odwzorowanie funkcji na sekwencję

- func przyjmuje tyle parametrów ile przekazaliśmy obiektów iterowalnych,
- map pobiera iteracyjnie elementy o tych samych indeksach z przekazanych parametrów i przekazuje je jako parametry funkcji func,
- zwracany wynik zapisywany jest w obiekcie map

```
map(lambda x: x**2, range(5))
#0, 1, 4, 9, 16

map(lambda x,y: x+y, range(5), range(5,10))
#5, 7, 9, 11, 13
```

Jeśli przekazane sekwencje są różnej długości wynikowy obiekt zawiera tyle elementów ile wynosi długości najkrótszej z nich.

zip

`zip`(*iterables) - łączy w krotki elementy o tych samych indeksach, zatrzymuje się na najkrótszej sekwencji, np:

```
zip(range(5), range(5,7), range(0,10,2))
#(0, 5, 0), (1, 6, 2)
```

filter

`filter`(function or None, iterable) - przechowuje wartości, dla których funkcja zwróciła wartość prawdy, a jeśli zamiast funkcji użyte zostało None - elementy, które mają wartość logiczną prawdy, np.:

```
filter(lambda x: x%2, range(10))          #1, 3, 5, 7, 9

filter(None, range(5))                    #1, 2, 3, 4
```

- Funkcje `map`, `zip` i `filter` zwracają obiekty własnych typów będące iteratorami.
- Można po nich iterować lub uzyskać kolejny element korzystając z funkcji `next`.
- Funkcja `next` zgłosza wyjątek `StopIteration` po wyczerpaniu dostępnych obiektów.
- Wyjątek `StopIteration` jest obsługiwany w pętli i powoduje jej zakończenie.

```
zipOb=zip(range(5), range(5,7), range(0,10,2))

next(zipOb)     #(0, 5, 0)
next(zipOb)     #(1, 6, 2)
next(zipOb)     #StopIteration


for el in zipOb:
  print(el)     #(0, 5, 0)
                #(1, 6, 2)
```

Po dojściu do końca iteratora jesteśmy na jego końcu, co oznacza, że:

- korzystamy z next przeważnie w przypadku obiektów nieskończonych
- tworzymy obiekt na potrzeby danej iteracji, np.:

```
for el in zip(range(5), range(5,7), range(0,10,2)):
    pass
```

- wymuszamy stworzenie listy

```
list(map(lambda x,y: x+y, range(5), range(5,10)))
```

reduce

```
import functools
functools.reduce(function, sequence[, initial]) # value
```

reduce

```
import functools
functools.reduce(function, sequence[, initial]) # value
```

```
functools.reduce(lambda x, y: x+y, [1, 2, 3, 4, 5])      # 15
                                                  #((((1+2)+3)+4)+5)
```

reduce

```
import functools
functools.reduce(function, sequence[, initial]) # value
```

```
functools.reduce(lambda x, y: x+y, [1, 2, 3, 4, 5])      # 15
                                                  #((((1+2)+3)+4)+5)
```

Zwracana wartość może być potencjalnie obiektem dowolnego typu!

Funkcje

Kilka pozostałych funkcji wbudowanych dla osób zainteresowanych pozostawiam do samodzielnego przestudiowania.

Przy okazji proszę też rzucić okiem na moduł itertools, może się Państwu kiedyś przydać.