Встроенные функции

Функция как объект

Как мы уже выяснили, функции в python также являются объектами:

```
def debug_with_args(f, *args, **kwargs):
    print(f'Calling function {f.__name__} with args {args} and
{kwargs}')
    return f(*args, **kwargs) # вызвали переданную функцию

def sum(a, b):
    return a + b

# можем сохранить в переменную:
func = sum
# можем передать как аргумент
print(debug_with_args(func, 1, 2))
# Calling function sum with args (1, 2) and {}
# 3
```

Функция как объект

```
def any(values, predicate):
    for v in values:
        if predicate(v):
            return True
    return False

def is_even(num):
    return num % 2 == 0

def is_odd(num):
    return not is_even(num)

print(any([1, 3, 5], is_even)) #?
print(any([1, 3, 5], is_odd)) #?
```

Анонимные функции.

Анонимные функции (или лямбды) позволяют записывать функции короче.

```
lambda arguments: expression
```

эквивалентно

```
def <lambda>(arguments):
    return expression
```

*Всё, сказанное прежде про именованные функции, справедливо и для анонимных. *

Анонимные функции

Код со второго слайда теперь можно переписать короче:

```
def any(values, predicate):
    for v in values:
        if predicate(v):
            return True
    return False

print(any([1, 3, 5], lambda x: x % 2 == 0)) #?
print(any([1, 3, 5], lambda x: x % 2 != 0)) #?
```

iterable

`iterable` — любой объект Python по которому можно проитерироваться.

Уже знакомые нам примеры `iterable` объектов: списки, кортежи, строки и словари — по их содержимому можно проитерироваться с помощью `for`.

Но, естественно, `iterable` объектов в python намного больше и перечисленный набор выше не исчерпывающий.

sorted, max, min

```
bears = [('polar', 450), ('giant panda', 100), ('brown', 80)]

# нахождение максимума/минимума --
# max(iterable, *[, default=obj, key=func]) -> value
# min(--//--) -> value
max(bears, key=lambda x: x[1])
# ('polar', 450)


# сортировка, всегда возвращает список --
# sorted(iterable, /, *, key=None, reverse=False)
sorted(bears, key=lambda x: x[1])
# [('brown', 80), ('giant panda', 100), ('polar', 450)]
```

sum, any, all

```
# просуммировать значения из iterable,
# start -- начальное значение
# подразумевается, что в iterable лежат только числа
# sum(iterable, start=0, /)
sum((1, 2, 3), start=-6)
# 0

# возвращает True, если для всех x из iterable
# bool(x) == True
# all(iterable, /)
all([False, True])
# False

# возвращает True, если хотя бы для одного x из iterable
# bool(x) == True
# any(iterable, /)
any([False, True])
# True
```

map

```
bears = [('polar', 450), ('giant panda', 100), ('brown', 80)]

# возвращает iterable, который получен из исходного
# применением функции к каждому значению:
# map(function, iterable, ...)
list(map(lambda x: x ** 2, [0, 1, 2]))

# обернули в list, т.к. map вернет специальный iterable:
names = map(lambda x: x[0], bears)
# <map at 0x10a439910>
for name in names:
    print(name, end=" ")
# polar giant panda brown

# в map можно передать функцию нескольких переменных и
несколько iterable
list(map(lambda x, y: x + y, [1, 2, 3], (2, 5))) #?
```

filter

```
bears = [('polar', 450), ('giant panda', 100), ('brown', 80)]

# возвращает iterable, который получен из исходного удалением
# всех элементов x для которых predicate(x) == False
list(filter(lambda x: x[1] >= 100, bears))
# [('polar', 450), ('giant panda', 100)]

# map, filter и другие функции можно комбинировать:
names_of_heavy = map(lambda x: x[0], filter(lambda x: x[1] >=
100, bears))
print(list(names_of_heavy))
# ['polar', 'giant panda']
```

zip, enumerate

```
a = ['a', 'b', 'c']
b = [25, 100]
c = [1, 1, 1, 1]

# zip принимает несколько последовательностей и возвращает
# iterable, в котором лежат кортежи;
# кортежи составляются из элементов исходных
последовательностей:
print(list(zip(a, b, c)))
# [('a', 25), ('b', 100)]

# enumerate также возвращает кортежи, но первым элементом в них
# лежит номер элемента
print(list(enumerate(a)))

# понятно, что enumerate выразим через zip и range. Как?
```

reduce

```
# применит функцию двух аргументов кумулятивно к элементам
# послетовательности слева направо
reduce(lambda x, y: x + y, [1, 2, 3, 4, 5])
# будет вычисляться следующим образом:

#     будут переданы в функцию
#     первыми т.е. x = 1, y = 2 на первом шаге
#     /---\
# ((((1 + 2) + 3) + 4) + 5)
#     ^---^    ^
#     \    |  /
#      результат вычисления на первом шаге
#      будет использован как x на втором,
#      т.е. x = 3, y = 3  и т.д.
#
# в результате будет выведено
# 15
```

reduce

```
# обязательно для использования импортировать reduce
from functools import reduce

values = [1, 2, 3, 4, 5]
# reduce очень мощная и общая функция, некоторые функции,
# о которых мы сегодня говорили, могут быть выражены через нее:

# max
reduce(lambda x, y: x if x > y else y, values, 0) # задали
начальное значение 0

# all
reduce(lambda x, y: bool(x) and bool(y), values)

# average
reduce(lambda x, y: x + y / len(values), values, 0)
```

Что еще почитать

Рекомендую посмотреть документацию ко всем функциям, о которых шла речь на лекции:

Посмотреть можно тут