Технологии программирования

Functional programing elements

Pakhomova K.I.,
Kononova N.V.

Code documentation

```
def Function():
    """
      There is an empty function
    """
print(Function.__doc__)
```

```
class Empty:
    "there is an empty class"
    pass
print(Empty.__doc__)
```

```
print(int.__doc__)
```

Function scope, global variables

```
X = 88 # Global X

def func():
    global X
    X = 99 # Global X: outside def

func()
print(X) # 99
```

Function key arguments

```
def func(a, b, c=2):
    return a + b + c
```

```
func(1, 2)# a = 1, b = 2, c = 2
func(1, 2, 3)  # a = 1, b = 2, c = 3
func(a=1, b=3)  # a = 1, b = 3, c = 2
func(a=3, c=6)  # a = 3, c = 6, b is undefined
```

Function arguments

```
>>> def func(*args):
...     return args
... func(1, 2, 3, 'abc')
>>> func(1, 2, 3, 'abc')
(1, 2, 3, 'abc')
>>> func()
()
>>> func(2)
(2,)
```

Function key arguments

```
>>> def func(**kwargs):
...     return kwargs
...
>>> func(a=1, b=2, c=3, d='abc')
{'a': 1, 'b': 2, 'c': 3, 'd': 'abc'}
>>> func()
{}
>>> func(a=1)
{'a': 1}
```

Programming style and paradigm

-**Imperative style** - the programmer instructs the machine how to change its state:

-- imperative,

-- structural,

-- procedural which groups instructions into procedures,

-- object-oriented which groups instructions together with the part of the state they operate on,

-**Declarative style –** the programmer merely declares properties of the desired result, but not how to compute it:

-- functional: the desired result is declared as the value of a series of function applications,

--logic: the desired result is declared as the answer to a question about a system of facts and rules,

--mathematical: the desired result is declared as the solution of an optimization problem

Programming paradigm

-Procedural: programs are lists of instructions that tell the computer what to do with the program's input.

-*Declarative style: a specification that describes the problem to be solved, and the language implementation figures out how to perform the computation efficiently*.

- Object-oriented: manipulate collections of objects.

-Functional: decomposes a problem into a set of functions. Ideally, functions only take inputs and produce outputs, and don't have any internal state that affects the output produced for a given input.

Functional programming

-Input flows through a set of functions.

- Each function operates on its input and produces some output.

-Functions return value

-*Purely functional*: every function’s output must only depend on its input. (have no side effects)

-Functions are first-class objects

-Higher-Order functions

- *Immutable data*

- No variables

- Closures (Замыкания)

- Currying (Каррирование), https://habr.com/ru/post/335866/

- Composition

First-class function

In Python everything is an object, including functions.

Functions in Python are **first-class objects, it:**

-have types

-can be sent as arguments to another function

-can be used in expression

-can become part of various data structures like dictionaries

First-class function

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

def g(function, x):
    return function(x) * function(x)

print(g(f, 7)) # 100
```

First-class function

```
def add(x, y):
    return x + y


def sub(x, y):
    return x - y


def mul(x, y):
    return x * y


def div(x, y):
    return x / y


operations = {
    '+': add,
    '-': sub,
    '*': mul,
    '/': div,
    '^': pow
}


try:
    first = float(input('First number: '))
    operation = input('Operation: ')
    second = float(input('Second number: '))
    result = first / second
```

Closures

A closure is a nested function which has access to a free variable from an enclosing function that has finished its execution.

It is returned from the enclosing function

Python closures help avoiding the usage of global values and provide some form of data hiding. They are used in Python decorators.

https://devpractice.ru/closures-in-python/

Closures

```
def add(n):
    def do_add(m):
        return m+n
    return do_add

sum = add(2)
print(sum)
result = sum(10)
print(result)
```

```
<function add.<locals>.do_add at 0x0000008B08DE3620>
12
```

Functional programming advantages and disadvantages

+

- Formal provability.
- Modularity.
- Caching.
- Parallelism.
- Ease of debugging and testing.

\-

- In practice we need the input and random data

Elements of the functional programming

In Python :

- lambda functions.
- filter() function.
- map() function.
- reduce() function.
- zip() function.
- Decorators
- Iterators and generators

Lambda-functions

Creating of the unnamed functions

```
def add(x, y):
    return x+y
print(add(3,4))

add = lambda x, y: x+y
print(add(3,4))
```

Lambda-functions

```
f = lambda x: x + 1
print(f(1)) # 2

f = lambda a, b: a - b
print(f(3, 2)) # 1
```

Lambda-functions

```
>>> func = lambda x, y: x + y
...
>>> func(1,2)
3
>>> func('a', 'b')
'ab'
```

```
>>> (lambda x, y: x + y)(1, 2)
3
>>> (lambda x, y: x + y)('a', 'b')
'ab'
```

```
>>> func = lambda *args: args
>>> func(2, 3, 4, 5)
(2, 3, 4, 5)
```

Lambda-functions and Sorted

```
l = [['a', 2], ['c', 1], ['b', 7]]
print(sorted(l))
print(sorted(l, key=lambda x: x[1]))
print(sorted(l, key=lambda x: x[1], reverse=True))
d = {'a': 1, 'b': 7, 'c': 5}
for k, v in sorted(d.items(), key = lambda x: x[1]):
    print(k, v)
```

Output:

```
[['a', 2], ['b', 7], ['c', 1]]
[['c', 1], ['a', 2], ['b', 7]]
[['b', 7], ['a', 2], ['c', 1]]
a 1
c 5
b 7
```

Filter function

- filter(func, iterable) – filter out items based on a test function

```
res = list(filter((lambda x: x >= 0), [0, -1, 3, -6]))
print(res)
```

Output:

```
[0, 3]
```

```
r = list(filter((lambda x: x.find('W') >= 0), ['Hi', 'Hello', 'World']))

print(r)   # ['World']
```

Filter function

```
people = [{ 'name': "Ann", 'age': 26},
          {'name': "Kaio", 'age': 10},
          {'name': "Kazumi", 'age': 30}]

# filter
p = filter(lambda x: x['age'] > 18, people)
print(*p)
```

```
{'name': 'Ann', 'age': 26} {'name': 'Kazumi', 'age': 30}
```

Map function

map(func, iterable)

- applies a passed-in function to each item in an iterable object
- returns a list containing all the function call results.

```
(list(map(str, [1, 4, 6])))

res = list(map(lambda x: x+1, [1, 4, 6]))
print(res)
```

Output:

```
['1', '4', '6']
[2, 5, 7]
```

Map function

```
people = [{ 'name': "Ann", 'age': 26}, {'name': "Kaio", 'age': 10},
{'name': "Kazumi", 'age': 30}]

p = map(lambda x: print(x['name'] + " is " + str(x['age'])), people)
print(*p)
```

```
Ann is 26
Kaio is 10
Kazumi is 30
```

Reduce function

reduce(func, iterable) – apply functions to pairs of items and running results

```
from functools import reduce  # Import in 3.X, not in 2.X
reduce((lambda x, y: x + y), [1, 2, 3, 4])  # 10
reduce((lambda x, y: x * y), [1, 2, 3, 4])  # 24
```

```
from functools import reduce
items = [11, 2, -7, 14, 3, 62, 1]
_max = reduce(lambda a, b: a if (a > b) else b, items)

print(_max)  # 62
```

Zip function

```
l1 = [1, 2, 3]
l2 = [1, 2, 3]
for a, b in zip(l1, l2):
    print(a, b)
```

Output

```
1 1
2 2
3 3
```

Built-in functions for sequense work

- sum(), min(), max(), sorted(), enumerate(), range()
- *any(), all()*

```
print(any((True, False, True))) # True

print(all((True, False, True))) # False
```

Decorator

Decorators wrap a function, modifying its behavior

```
def decorator_name(func):
    def decorated_fn(*args, **kwargs):
        print("Something is happening before the function is called.") func()
        print("Something is happening after the function is called.") return
    decorated_fn
```

```
def say_hello(): print("Hello!")

say_hello = decorator_name(say_hello)
say_hello()
```

Decorator

```
@decorator
def say_hello():
    print("Hello!")

say_hello()
```

```
Something is happening before the function is called.
Hello!
Something is happening after the function is called.
```

Decorator example

```
import time

def timer(func):
    def wrapper_timer(*args, **kwargs):
        start_time = time.perf_counter()    # 1
        value = func(*args, **kwargs)
        end_time = time.perf_counter()      # 2
        run_time = end_time - start_time    # 3
        print(f"Finished {func.__name__!r} in {run_time:.4f} secs")
        return value
    return wrapper_timer

@timer
def some_function(num_times):
    for _ in range(num_times):
        sum([i**2 for i in range(10000)])

some_function(100)
```

Generator expressions and list comprehensions

Two common operations on an iterator's output are

1) performing some operation for every element,
2) selecting a subset of elements that meet some condition.

```
line_list = [' line 1\n', 'line 2  \n', ...]

# Generator expression -- returns iterator
splitted_iter = (line.split(\n) for line in line_list)

# List comprehension -- returns list splitted_list
= [line.split(\n) for line in line_list]
```

List comprehensions

```
sq = []
for i in range(10):
    sq.append(i**2)
print(sq)

sq = [x**2 for x in range(10)]
print(sq)
```

Output:

```
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

List comprehension

```
l = [x**2 for x in range(1,10)]
```

Generator expressions and list comprehensions

```
for x in range(5, 10):
    if x % 2 == 0:
        x *=2
    else:
        x += 1

a = [x * 2 if x % 2 == 0 else x + 1 for x in range(5, 10)]
print(a)
```

```
[6, 12, 8, 16, 10]
```

Complex list comprehensions

```python
print([(x, y) for x in [1, 2, 3]
       for y in [1, 4, 3] if x!=y])

combs = []
for x in [1, 2, 3]:
    for y in [1, 4, 3]:
        if x != y:
            combs.append((x, y))
```

Complex list expressions

```python
print([(x, y) for x in [1, 2, 3]
       for y in [1, 4, 3] if x != y])

combs = []
for x in [1, 2, 3]:
    for y in [1, 4, 3]:
        if x != y:
            combs.append((x, y))
print(combs)
```

Output:

```
[(1, 4), (1, 3), (2, 1), (2, 4), (2, 3), (3, 1), (3, 4)]
[(1, 4), (1, 3), (2, 1), (2, 4), (2, 3), (3, 1), (3, 4)]
```

Complex list comprehensions

```
matrix = [[1, 3, 4, 6],
          [6, 3, 2, 8],
          [10, 1, 1, 4]]

print([[row[i] for row in matrix]
        for i in range(4)])
```

Output:

```
[[1, 6, 10], [3, 3, 1], [4, 2, 1], [6, 8, 4]]
```

Iterable objects

An object is called **iterable** if one can get an iterator for it.

Data types support iteration: list, tuple, string and dictionaries.

```
>>> for x in range(5):
...     print(x)
...
0
1
2
3
4
```

Iterators

– object representing a stream of data; returns the data one element at a time.

It support a method __next__() that takes no arguments and always returns the next element of the stream.

The **iter()** function takes an object and tries to return an iterator that will return the object’s contents or elements.

Iterators

```
for i in iter(obj):
    print(i)
```

Is equal to

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

Iterators

```
>>> x = [1, 2, 3]
>>> iter(x)
<list_iterator object at 0x0000001B22C3B278>
>>> it = iter(x)
>>> next(it)
1
>>> next(it)
2
>>> next(it)
3
>>> next(it)
Traceback (most recent call last):
  File "<input>", line 1, in <module>
StopIteration
```

Generator

Generator functions allow to declare a function that behaves like an iterator, i.e. it can be used in a for loop.

Regular functions compute a value and return it, but generators return an iterator that returns a stream of values.

Any function containing a yield keyword is a generator function.

It returns a generator object that supports the iterator protocol.

The difference between yield and a return – on reaching a yield the generator's state of execution is suspended and local variables are preserved. On the next call to the generator's __next__() method, the function will resume executing.

Generator expressions

```
a = (i for i in range(1000000000))
print(a)
print(next(a))
print(next(a))
```

```
<generator object <genexpr> at 0x0000006542386D68>
0
1
```

Generator functions

```
def squares(x):
    for i in range(x):
        yield i ** 2

print(squares(5))
print(list(squares(5)))
```

```
<generator object squares at 0x00000087616C7C78>
[0, 1, 4, 9, 16]
```

Generator functions

```python
def gf():
    s = 5
    for i in [3, 6, 8]:
        yield i
        print(s)
        s = s + 5
```

```python
g = gf()
for j in g:
    print(j)
    print("--------")
```

Output

```
3
--------
5
6
--------
10
8
--------
15
```

```python
g = gf()
print(next(g))
print(next(g))
```

```
3
s = 5
6
```

Itertools module

Itertools – functions creating iterators for efficient looping

from itertools **import** *

Infinite iterators:

Iterator	Arguments	Results	Example
`count()`	start, [step]	start, start+step, start+2*step, …	`count(10) --> 10 11 12 13 14 ...`
`cycle()`	p	p0, p1, … plast, p0, p1, …	`cycle('ABCD') --> A B C D A B C D ...`
`repeat()`	elem [,n]	elem, elem, elem, … endlessly or up to n times	`repeat(10, 3) --> 10 10 10`

https://docs.python.org/3/library/itertools.html

Iterators terminating on the shortest input sequence:

Iterator	Arguments	Results	Example
`accumulate()`	p [,func]	p0, p0+p1, p0+p1+p2, …	`accumulate([1,2,3,4,5]) --> 1 3 6 10 15`
`chain()`	p, q, …	p0, p1, … plast, q0, q1, …	`chain('ABC', 'DEF') --> A B C D E F`
`chain.from_iterable()`	iterable	p0, p1, … plast, q0, q1, …	`chain.from_iterable(['ABC', 'DEF']) --> A B C D E F`
`compress()`	data, selectors	(d[0] if s[0]), (d[1] if s[1]), …	`compress('ABCDEF', [1,0,1,0,1,1]) --> A C E F`
`dropwhile()`	pred, seq	seq[n], seq[n+1], starting when pred fails	`dropwhile(lambda x: x<5, [1,4,6,4,1]) --> 6 4 1`
`filterfalse()`	pred, seq	elements of seq where pred(elem) is false	`filterfalse(lambda x: x%2, range(10)) --> 0 2 4 6 8`
`groupby()`	iterable[, key]	sub-iterators grouped by value of key(v)	

Combinatoric iterators

Iterator	Arguments	Results
`product()`	p, q, … [repeat=1]	cartesian product, equivalent to a nested for-loop
`permutations()`	p[, r]	r-length tuples, all possible orderings, no repeated elements
`combinations()`	p, r	r-length tuples, in sorted order, no repeated elements
`combinations_with_replacement()`	p, r	r-length tuples, in sorted order, with repeated elements
`product('ABCD', repeat=2)`		`AA AB AC AD BA BB BC BD CA CB CC CD DA DB DC DD`
`permutations('ABCD', 2)`		`AB AC AD BA BC BD CA CB CD DA DB DC`
`combinations('ABCD', 2)`		`AB AC AD BC BD CD`
`combinations_with_replacement('ABCD', 2)`		`AA AB AC AD BB BC BD CC CD DD`

Iterators example

itertools.count(*start=0*, *step=1*)

Make an iterator that returns evenly spaced values starting with number *start*.

```
>>> import itertools
>>> itertools.count(start=0, step=1)
count(0)
>>> itertools.count(start=0, step=1)
count(0)
>>> for x in itertools.count(start=0, step=1):
...     print(x)
...     if x == 100:
...         break
...
0
1
2
3
4
```

Iterators example

itertools.repeat(*object*[, *times*])

Make an iterator that returns *object* over and over again. Runs indefinitely unless the *times* argument is specified.

```
>>> for i in itertools.repeat('4', 5):
...     print(i)
...
4
4
4
4
4
```

Iterators example

itertools.accumulate(*iterable*[, *func*])

Make an iterator that returns accumulated sums, or accumulated results of other binary functions (specified via the optional *func* argument).

If *func* is supplied, it should be a function of two arguments.

```
s = itertools.accumulate([1, 2, 3, 4, 5])
print(s) # <itertools.accumulate object at 0x000000D59D66AA08>
print(*s) # [1, 3, 6, 10, 15]
print(list(s))
```

Iterators example

```
s = itertools.chain(*([1, 2, 3], [14, 15, 16]))
print(*s) # 1 2 3 14 15 16

s = itertools.combinations('ABCD', 2)
print(s) # <itertools.combinations object at 0x0000000AC98162C8>
print(*s) # ('A', 'B') ('A', 'C') ('A', 'D') ('B', 'C') ('B', 'D') ('C', 'D')

s = itertools.combinations_with_replacement('ABCD', 2)
print(*s) # ('A', 'A') ('A', 'B') ('A', 'C') ('A', 'D') ('B', 'B') ('B', 'C') ('B', 'D') ('C', 'C') ('C', 'D') ('D', 'D'

s = itertools.compress('ABCDEF', [1,0,1,0,1,1])
print(*s) # A C E F

s = itertools.dropwhile(lambda x: x < 5, [1,2,7,3,1])
print(*s) # 7 3 1
```

Iterators example

```
s = itertools.dropwhile(lambda x: x < 5, [1,2,7,3,1])
print(*s) # 7 3 1

s = itertools.filterfalse(lambda x: x < 5, [1,4,6,4,1])
print(*s) # 6


s = itertools.islice(range(100), 23)
print(*s) # 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22

s = itertools.permutations([1, 2, 3], r=None)
print(*s) # (1, 2, 3) (1, 3, 2) (2, 1, 3) (2, 3, 1) (3, 1, 2) (3, 2, 1)

s = itertools.product('ABCD', 'xy')
print(*s)  # ('A', 'x') ('A', 'y') ('B', 'x') ('B', 'y') ('C', 'x') ('C', 'y') ('D', 'x') ('D', 'y')

s = itertools.starmap(pow, [(2,5), (3,2), (10,3)])
print(*s) # 32 9 1000

s = itertools.takewhile(lambda x: x<5, [1,4,6,4,1])
print(*s) # 1 4
```

Functional programming refs:

- https://tproger.ru/translations/functional-programming-concepts/
- https://habr.com/en/post/257903/
- https://docs.python.org/dev/howto/functional.html
- https://www.youtube.com/watch?v=t4AhK0oWd9I&t=423s

Functional programming refs:

- Iterators & generators

https://www.youtube.com/watch?v=F3fspO4EEC8&t=738s