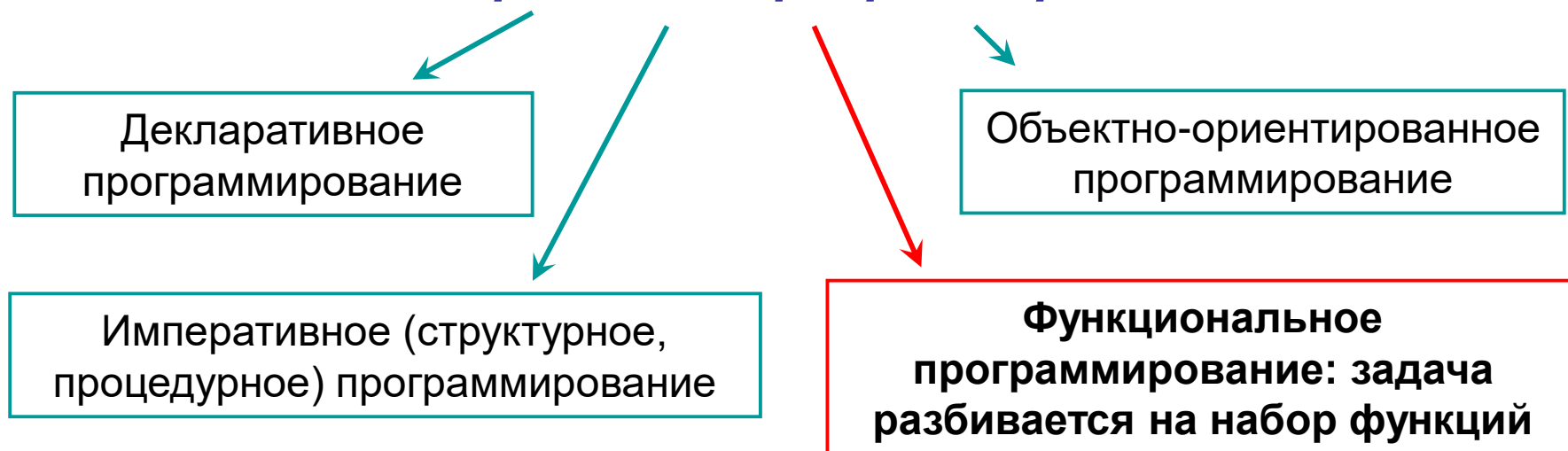


Лекция 3

Функциональное программирование

1 Понятие функционального программирования

Парадигмы программирования



Python поддерживают различные парадигмы программирования:


разные части программы можно писать в разных парадигмах, например, использовать функциональный стиль для обработки больших данных, объектно-ориентированный подход для реализации интерфейса и императивное программирование для промежуточной логики

2 Функция lambda

Синтаксис:

lambda список_аргументов: выражение

```
1 f = lambda x,y: x*y
2
3 f1 = f(2,8)
4
5 print( f1 )
6 print( f(3,5) )
```



```
16
15
```

f – функциональный объект первого класса

Сравнение способов описания функций:

```
1  n = 10
2
3  # 1-е определение функции
4  def pow1( k ):
5      return k * k * k
6  print( pow1(n) )
7
8  # 2-е определение функции
9  pow2 = lambda k: k * k * k
10 print (pow2(n))
```



```
1000
1000
```

Можно выполнять манипуляции с функциональными объектами как с объектами данных, например:

- динамически изменять;
- встраивать в более сложные структуры данных (коллекции);
- передавать в качестве параметров и возвращаемых значений.

```
1 #динамическое изменение объекта pow1
2 pow1=6
3 print ('Строка 3 = ', pow1)
4
5 pow1 = lambda k: k * k * k
6 print ('Строка 6 = ', pow1(5))
7
8 def pow1( k ):
9     return k * k
10 print('Строка 10 = ', pow1(5) )
11
12 #встраивание в более сложные структуры данных (коллекции)
13 l = lambda k: k * k * k
14 list = [1,2,l(5),8]
15 print('Строка 15 = ', list)
16
17 # передача функционального объекта
18 # в качестве параметра функции
19 def f(k):
20     return k+k
21
22 o = lambda d: d * d
23 print('Строка 23 = ', f(o(3)) )
24 print('Строка 24 = ', o(3) )
```

```
Строка 3 = 6
Строка 6 = 125
Строка 10 = 25
Строка 15 = [1, 2, 125, 8]
Строка 23 = 18
Строка 24 = 9
```

3 Функции для работы с последовательностями

ITERABLE (ИТЕРИРУЕМЫЙ) - объект, предоставляющий возможность поочерёдного прохода по своим элементам.

Примеры типов, поддерживающих итерирование по своим элементам :
список, строка, кортеж, словарь, файл.

Ниже приведены функции, которые принимают в качестве параметра `iterable`:

1) **sum** - находит сумму всех элементов последовательности `iterable`

2) **min**, **max** - находит минимум и максимум в последовательности `iterable`

3) Функция map

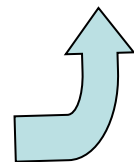
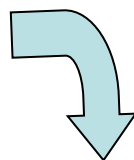
Применение функции к каждому элементу последовательности:

map(функция, последовательность)

```
1 old_list = ['1', '2', '3', '4', '5', '6', '7']
2 print ('old_list = ', old_list)
3
4 new_list = []
5 for item in old_list:
6     new_list.append(int(item))
7
8 print ("new_list = ", new_list)
```

```
old_list = ['1', '2', '3', '4', '5', '6', '7']
new_list = [1, 2, 3, 4, 5, 6, 7]
```

```
1 old_list = ['1', '2', '3', '4', '5', '6', '7']
2 print ('old_list = ', old_list)
3
4 new_list = list(map(int, old_list))
5 print ("new_list = ", new_list)
```



Преимущества:

- 1) меньше строк кода;
- 2) код читабельнее;
- 3) код быстрее выполняется.

Функция **map** работает с функциями, созданными пользователем:

```
1 def miles_to_kilometers(num_miles):  
2     return num_miles * 1.6  
3  
4 mile_distances = [1.0, 6.5, 17.4, 2.4, 9]  
5 kilometer_distances = list(map(miles_to_kilometers, mile_distances))  
6 print (kilometer_distances)
```



```
[1.6, 10.4, 27.84, 3.84, 14.4]
```

Функция **map** может использовать **lambda** выражение:

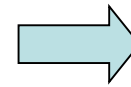
```
1 mile_distances = [1.0, 6.5, 17.4, 2.4, 9]  
2 kilometer_distances = list(map(lambda x: x * 1.6, mile_distances))  
3  
4 print (kilometer_distances)
```



```
[1.6, 10.4, 27.84, 3.84, 14.4]
```


Функция **map** для нескольких списков:

```
1 l1 = [1,2,3]
2 l2 = [4,5,6]
3
4 new_list = list(map(lambda x,y: x + y, l1, l2))
5 print (new_list)
```



[5, 7, 9]

```
1 l1 = [1,2,3]
2 l2 = [4,5]
3
4 new_list = list(map(lambda x,y: x + y, l1, l2))
5 print (new_list)
```



[5, 7]

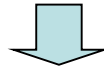
4) Функция filter

Фильтрация элементов последовательности:

filter(функция, последовательность)

функция должна возвращать значение True / False

```
1 mixed = ['кот', 'собака', 'кот', 'кот', 'лиса', \
2 'Кот', 'собака', 'собака', 'кот']
3 list_cat = list(filter(lambda x: x == 'кот', mixed))
4
5 print (list_cat)
```



```
['кот', 'кот', 'кот', 'кот']
```

Как будет выглядеть код без
использования функции lambda?

5) Функция reduce

Получение единственного значения из последовательности:

reduce(функция, последовательность[, начальное_знач])

Функция от 2ух параметров:

- 1) аккумулярованное ранее значение,
- 2) следующий элемент последовательности.

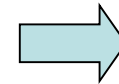
Последовательность, элементы которой требуется свести к единственному значению

Значение, с которого требуется начать отсчёт

Функция **reduce** из модуля `functools`:

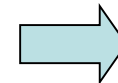
***from** functools **import** reduce*

```
1 from functools import reduce
2 items = [1,2,3,4,5]
3 sum_all = reduce(lambda x,y: x + y, items)
4
5 print (sum_all)
```



15

```
1 from functools import reduce
2 items = [1,2,3,4,5]
3 sum_all = reduce(lambda x,y: x + y, items, 10)
4
5 print (sum_all)
```



25

Как будет выглядеть код без использования функции reduce?

6) enumerate - возвращает кортежи из номера элемента (при нумерации с нуля) и значения очередного элемента

7) any, all - возвращают истину, если хотя бы один или все элементы iterable истинны соответственно

8) zip(iterA, iterB, ...) - конструирует кортежи из элементов (iterA[0], iterB[0], ...), (iterA[1], iterB[1], ...), ...

4 Библиотеки itertools и functools

Библиотека itertools:

- `itertools.combinations(iterable, size)` - генерирует все подмножества множества `iterable` размером `size` в виде кортежей;
- `itertools.permutations(iterable)` - генерирует все перестановки `iterable`;
- `itertools.combinations_with_replacement(iterable, size)` - генерирует все подмножества `iterable` размером `size` с повторениями, т.е. одно и то же число может входить в подмножество несколько раз.

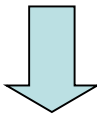
Библиотека functools: `partial`, `reduce`, `accumulate`.

5 Рекурсия

```
1  n = 5
2
3  # вычисление циклом
4  fact_1 = 1
5  while n > 1:
6      fact_1 *= n
7      n -= 1
8  print(fact_1)
9
10 n = 5
11
12 # вычисление рекурсией
13 def fact_2(n):
14     if n == 1:
15         return 1
16     return fact_2(n-1) * n
17 print(fact_2(n))
18
19 # возможности функционального программирования
20 fact_3 = lambda x: ( ( x == 1 ) and 1 ) or x * fact_3( x - 1 )
21 print (fact_3(n))
```

120
120
120

```
1 | if <условие>:  
2 |     <выражение 1>  
3 | else:  
4 |     <выражение 2>
```



```
1 | # функция без параметров:  
2 | lambda: ( <условие> and <выражение 1> ) or ( <выражение 2> )
```

```
1 | n=int(input("N = "))  
2 |  
3 | if n==1:  
4 |     k1=20  
5 | else:  
6 |     k1=30  
7 | print (k1)  
8 |  
9 | k2 = lambda: ( (n==1) and 20 ) or ( 30 )  
10 | print(k2())
```



```
N = 1  
20  
20
```


6 Карринг

Карринг — это преобразование функции от многих аргументов в набор функций, каждая из которых является функцией от одного аргумента


```
1  # простая функция для приветствия
2  def greet(greeting, name):
3      print(greeting + ', ' + name)
4  greet('Hello', 'Alex')
5
6  # карринг
7  def greet_curried(greeting):
8      def greet(name):
9          print(greeting + ', ' + name)
10         return greet
11
12  greet_hello = greet_curried('Hello')
13
14  greet_hello('German')
15  greet_hello('Ivan')
16
17  # вызов greet_curried напрямую
18  greet_curried('Hi')('Roma')
```



```
Hello, Alex
Hello, German
Hello, Ivan
Hi, Roma
```


Карринг можно делать с любым количеством аргументов:

```
1 def greet_deeply_curried(greeting):
2     def w_separator(separator):
3         def w_emphasis(emphasis):
4             def w_name(name):
5                 print(greeting + separator + name + emphasis)
6                 return w_name
7             return w_emphasis
8     return w_separator
9
10 greet = greet_deeply_curried("Hello")(" , ")(" ! ")
11 greet('German')
12 greet('Ivan')
13
14 greet1 = greet_deeply_curried("Hi")(" ... ")
15 greet2 = greet1(" . ")
16 greet2('Alex')
```



Hello , German !
Hello , Ivan !
Hi ... Alex .

Карринг с lambda:



```
1 greet_deeply_curried = lambda greeting: lambda separator: lambda emphasis: lambda name: \
2     print(greeting + separator + name + emphasis)
3
4 greet = greet_deeply_curried("Hello")(" , ")(" ! ")
5 greet('German')
6 greet('Ivan')
7
8 greet1 = greet_deeply_curried("Hi")(" ... ")
9 greet2 = greet1(" . ")
10 greet2('Alex')
```