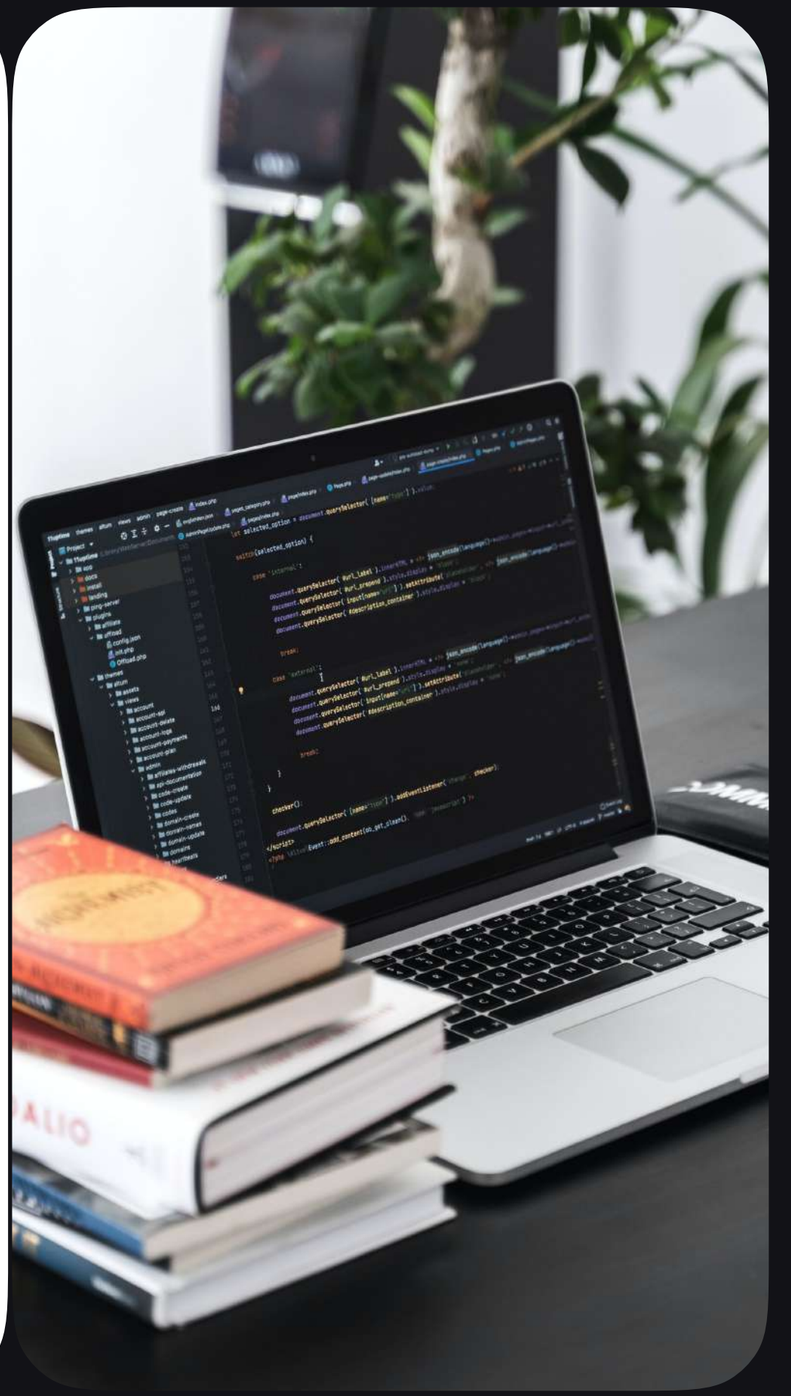
Модуль 2 Занятие 11

Функциональное программирование



Функции

Функциональное программирование

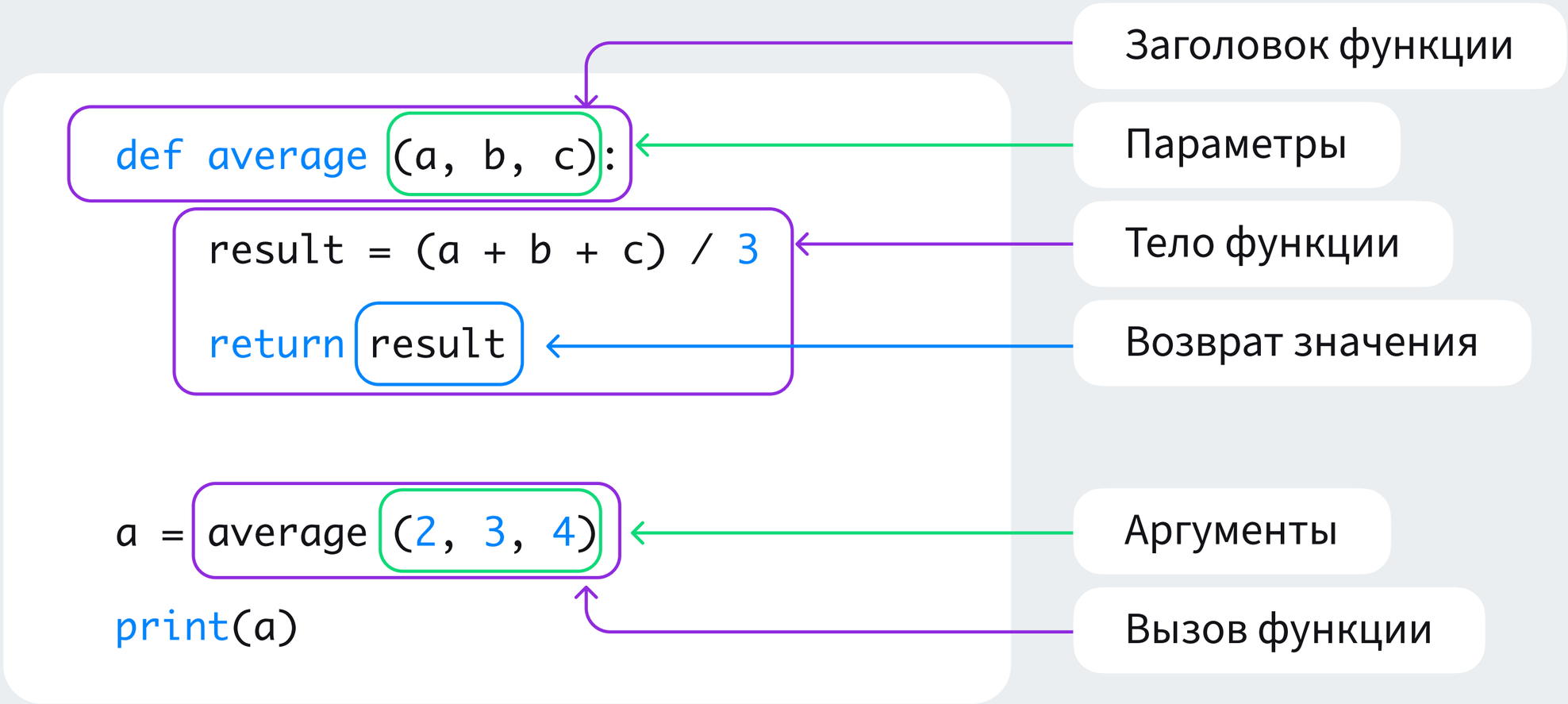
Функции высшего порядка

Анонимные функции

Рекурсия

Функции

Функция — отдельная часть программы, выполняющая определенную задачу.



Позиционные и именованные аргументы

```
def average(a, b, c, precision=0):
    result = (a + b + c) / 3
    return round(result, precision)


print(average(2, 3, 3, precision=3))
print(average(2, 3, 3, 1))
print(average(2, 3, 3))
```

Аргументы, передаваемые без имени, называются позиционными. Аргументы, передаваемые с именами, называются именованными. Можно задать значение параметра по умолчанию, в этом случае аргумент может не передаваться при вызове функции.

Вывод:

```
2.667
2.7
3.0
```

Позиционные и именованные аргументы

```
def average(a, b, c, precision=0):
    result = (a + b + c) / 3
    return round(result, precision)
```

Можно вызвать функцию следующим образом?

```
print(average(c=3, b=2, a=1))
print(average(1, 2, c=3, precision=3))
print(average(a=1, 2, 3, precision=3))
```

Вызывать функции можно комбинируя именованные и позиционные аргументы одновременно. Сначала указываются позиционные, а затем именованные аргументы.

Функции. Повторение

```
def average(*args, **kwargs):
    result = sum(args) / len(args)
    if 'precision' in kwargs:
        return round(result, kwargs['precision'])
return round(result)


print(average(2, 3))
print(average(2, 3, 3))
print(average(2, 3, 3, precision=3))
print(average(2, 3, 3, 5, precision=3, test='test'))
```

При вызове функции можно передавать произвольное число аргументов. Для этого используют переменную со звездочкой *args. При этом в самой переменной args будет кортеж.

Для получения произвольного числа именованных аргументов, в виде словаря, используется переменная **kwargs.

Вывод:

```
2
3
2.667
2.667
```

Парадигмы программирования

Парадигма программирования — это совокупность идей и понятий, определяющих стиль написания компьютерных программ (подход к программированию). Это способ концептуализации, определяющий организацию вычислений и структурирование работы, выполняемой компьютером.

Рассмотрим две парадигмы программирования: **императивную** и **декларативную**.

Парадигмы программирования

Императивная программа похожа на приказы (англ. **imperative** — приказ, повелительное наклонение), выражаемые повелительным наклонением в естественных языках, то есть представляют собой последовательность команд, которые должен выполнить исполнитель.

Императивное программирование отвечает на вопрос «**Как?**».
В рамках этой парадигмы задается последовательность действий, которые нужно выполнить, для того чтобы получить результат.
Результат выполнения сохраняется в памяти, для этого используется оператор присваивания, к которому можно обратиться в последствии.

Парадигмы программирования

Декларативное программирование отвечает на вопрос «**Что?**».
В этом случае описывается задача и ожидаемый результат работы программы, но не определяется, как этот результат будет получен.

Каждая из этих парадигм включает в себя более специфические модели. Например, объектно-ориентированное программирование относится к группе «императивное программирование»,
а функциональное программирование к группе «декларативное программирование».

Язык программирования Python мультипарадигменный,
он не является функциональным языком программирования,
но на нем можно разрабатывать программы в функциональном стиле.

Функциональное программирование

Функциональное программирование — парадигма декларативного программирования, в которой процесс вычисления трактуется как вычисление значений функций в их математическом понимании.

Функциональное программирование

Основные идеи функционального программирования:

- С функциями можно работать, также как и с данными: передавать их в качестве аргументов другим функциям, присваивать переменным, функции могут возвращать функции и так далее.
- Неизменяемые переменные — в функциональных языках не используются переменные (как именованные ячейки памяти), следовательно нет операции присваивания (нельзя изменить значение ячейки).
- **Чистые функции** — про такие функции говорят, что они не имеют побочных эффектов. Чистые функции не меняют глобальных переменных, ничего никуда не присваивают, не печатают. Принимают данные, что-то вычисляют, учитывая только аргументы, и возвращают результа.
- **Функции высшего порядка** — могут принимать другие функции в качестве аргумента или возвращать их.
- **Рекурсия** — является основным подходом для управления вычислениями, а не циклы и условные операторы.
- **Лямбда-выражения** — способ определения анонимных функций.

Функциональное программирование акцентируется на том, **что** должно быть вычислено, а не **как**.

Функциональное программирование

Преимущества функционального программирования:

- надежность
- лаконичность
- удобство тестирования
- оптимизация
- параллельные вычисления

Различные подходы к созданию списка

Описание последовательности действий:

```
lst = []
for i in range(10, 20):
    if i % 2 == 0:
        lst.append(i ** 2)
print(lst)
```

Использование списочного выражения:

```
lst = [i ** 2 for i in range(10, 20) if i % 2 == 0]
print(lst)
```

Использование функций высшего порядка:

```
lst = list(map(lambda x: x ** 2, filter(lambda x: x % 2 == 0, range(10, 20))))
```

Функции, как объекты

Функции в Python — это объекты. Это означает, что с функциями вы можете работать также как и с другими объектами: передавать их в качестве аргументов другим функциям, присваивать переменным, функции могут возвращать в качестве результата другую функцию и так далее.

```
def add_one(a):
    return a + 1


f = add_one
print(type(f))
print(f(5))
```

Вывод:

```
<class 'function'>
6
```

Анонимные функции

Стандартное определение функции:

```
def add_one(x):
    return x + 1
```

Аналогичная функция с использованием lambda:

```
add_one = lambda x: x + 1
```

Такая функция называется анонимная функция или лямбда-функция.

Когда можно использовать лямбда-функции:

- если нужно однократное использование функции
- необходимо передать функцию в качестве аргумента другой функции
- необходимо вернуть функцию в качестве результата другой функции

Пример

Что будет выведено в результате работы программы?

```
students = [('Oleg', [4, 4, 4]), ('Ivan', [3, 4, 5]), ('Lev', [3, 3, 3])]
students.sort(key=lambda value: (sum(value[1]), value[0]))
print(*[value[0] for value in students])
```

- Ivan Oleg Lev
- Lev Ivan Oleg
- Oleg Ivan Lev
- Lev Oleg Ivan

Функции высшего порядка

Функции высшего порядка — это функции, которые могут принимать в качестве аргументов функции и возвращать в качестве результата другие функции.

Функции **map()**, **filter()**, **reduce()**, языка Python, являются функциями высшего порядка.

map()

```
def func(x):
    return x ** 2



res_1 = map(len, ['hello', '123', 'python'])
res_2 = map(str.upper, ['hello', '123', 'python'])
res_3 = map(func, range(1, 5))
res_4 = map(lambda x: x ** 2, range(1, 5))
res_5 = map(lambda x, y: x + y, range(1, 5), range(5, 10))

print(res_1)
print(*res_1)
print(*res_2)
print(*res_3)
print(*res_4)
print(*res_5)
```

map(function, sequence1, ...) — применяет функцию к каждому элементу последовательности (последовательностей) и возвращает последовательность результатов

Вывод:

```
<map object at 0x00000243BC4D6CB0>
5 3 6
HELLO 123 PYTHON
1 4 9 16
1 4 9 16
6 8 10 12
```

filter()

```
res_1 = filter(str.isalpha, ['hello', '123', 'python'])
res_2 = filter(lambda x: x % 2 == 0, range(1, 5))
res_3 = filter(lambda x: x < 0, [10, 4, 2, -1, 6])
res_4 = filter(lambda x: x > 10, map(lambda x, y: x + y, range(1, 5), range(5, 10)))
print(res_1)
print(*res_1)
print(*res_2)
print(*res_3)
print(*res_4)
```

filter(function, sequence) — оставляет только те элементы последовательности для которых функция вернула True

Вывод:

```
<filter object at 0x00000153491D6CB0>
hello python
2 4
-1
12
```

reduce()

```
from functools import reduce

res_1 = reduce(lambda x, y: x + y, range(1, 5)) # ((((1 + 2) + 3) + 4) + 5)
res_2 = reduce(lambda x, y: x + y, range(1, 5), 10)
res_3 = reduce(lambda x, y: [y] + x, [1, 2, 3, 4, 5], [])
print(res_1)
print(res_2)
print(res_3)
```

reduce(function, sequence, init_value) — сворачивает элементы последовательности в один объект, применяя функцию по очереди к последовательным парам элементов. Начальное значение init_value — необязательный параметр.

Вывод:

```
10
20
[5, 4, 3, 2, 1]
```

Пример

Что будет выведено в результате работы программы?

```
lst = [(5, 6), (121, 11), (20, 1, 1), (1, 2, 1)]
result = filter(lambda s: s == s[::-1], map(lambda x: str(sum(x)), lst))
print(*result)
```

- 121 11 1 2 1
- 11 22 4
- (5, 6) (20, 1, 1) (1, 2, 1)
- 11 132 22 4

all(), any()

all(sequence) — возвращает True если все элементы последовательности истины или последовательность пуста.

any(sequence) — возвращает True, если хотя бы один элемент последовательности истинен, если последовательность пуста возвращается False.

```
res_1 = all([1, 2, 3, 0, 1])
res_2 = all(map(str.isalpha, ['hello', '123', 'python']))
res_3 = all(map(lambda x: x % 2 == 0, [1, 2, 3, 4, 5]))
res_4 = any(map(lambda x: x % 2 == 0, [1, 2, 3, 4, 5]))
print(res_1)
print(res_2)
print(res_3)
print(res_4)
```

Вывод:

```
False
False
False
True
```

enumerate()

enumerate(sequence) — возвращает индекс элемента и сам элемент последовательности в качестве кортежа. Также можно указать необязательный параметр start.

```
res_1 = enumerate('hello')
res_2 = enumerate(['a', 'b', 'c'], start=5)

print(res_1)
print(*res_1)
print(*res_2)
```

Вывод:

```
<enumerate object at 0x00000251DE3BBD00>
(0, 'h') (1, 'e') (2, 'l') (3, 'l') (4, 'o')
(5, 'a') (6, 'b') (7, 'c')
```

zip()

zip(sequence1, sequence2) объединяет элементы каждой последовательности в кортежи.

```
res_1 = zip([1, 2, 3], 'abc')
res_2 = zip([1, 2, 3], 'abc')
res_3 = zip(range(5), 'hello')
print(res_1)
print(*res_1)
print(dict(res_2))
print(*res_3)
```

Вывод:

```
<zip object at 0x000001DAF8DFDEC0>
(1, 'a') (2, 'b') (3, 'c')
{1: 'a', 2: 'b', 3: 'c'}
(0, 'h') (1, 'e') (2, 'l') (3, 'l') (4, 'o')
```

Пример

Что будет выведено в результате работы программы?

```
alpha = 'ABCD'
result = dict(map(lambda x: (x[1], x[0]), enumerate(alpha)))
print(result)
```

- {'A': 0, 'B': 1, 'C': 2, 'D': 3}
- A B C D
- {0: 'A', 1: 'B', 2: 'C', 3: 'D'}
- 0 1 2 3

Рекурсия

Строго говоря, в функциональной парадигме программирования нет такого понятия, как цикл. В функциональных языках цикл обычно реализуется в виде рекурсии.

Рекурсивная функция — это функция, которая вызывает сама себя напрямую или через другие функции.

Решение с помощью цикла:

```
n = 10
total = 1
for i in range(1, n + 1):
    total *= i
print(total)
```

Вывод:

```
3628800
```

Решение с помощью рекурсии:

```
def factorial(n):
    if n == 0:
        return 1
    return n * factorial(n - 1)

n = 10
print(factorial(n))
```

Вывод:

```
3628800
```

Рекурсия

Чтобы написать программу с рекурсией, необходимо выполнить 2 условия:

1. Написать функцию, которая вызывает сама себя напрямую или через другие функции.
2. Добавить базовые случаи — условия выхода из рекурсии, когда функция перестает вызывать себя, иначе функция будет вызывать себя бесконечно и программа завершится с ошибкой.

Итоги

- Язык программирования Python мультипарадигменный — он не является функциональным языком программирования, но на нем можно разрабатывать программы в функциональном стиле.
- Функциональное программирование — парадигма декларативного программирования, в которой процесс вычисления трактуется как вычисление значений функций в их математическом понимании.
- Основные идеи функционального программирования: с функциями можно работать, также как и с любыми данными, в функциональном программировании нет переменных и присваивания, в функциональном программировании используются чистые функции без побочных эффектов, функции высшего порядка, рекурсия и лямбда выражения.