КАЛУЖСКИЙ ФИЛИАЛ
ФЕДЕРАЛЬНОГО ГОСУДАРСТВЕННОГО БЮДЖЕТНОГО
ОБРАЗОВАТЕЛЬНОГО УЧРЕЖДЕНИЯ ВЫСШЕГО ОБРАЗОВАНИЯ
«МОСКОВСКИЙ ГОСУДАРСТВЕННЫЙ ТЕХНИЧЕСКИЙ
УНИВЕРСИТЕТ ИМЕНИ Н.Э. БАУМАНА
(национальный исследовательский университет)»

Факультет «Информатика и управление»

Кафедра «Программное обеспечение ЭВМ, информационные технологии»

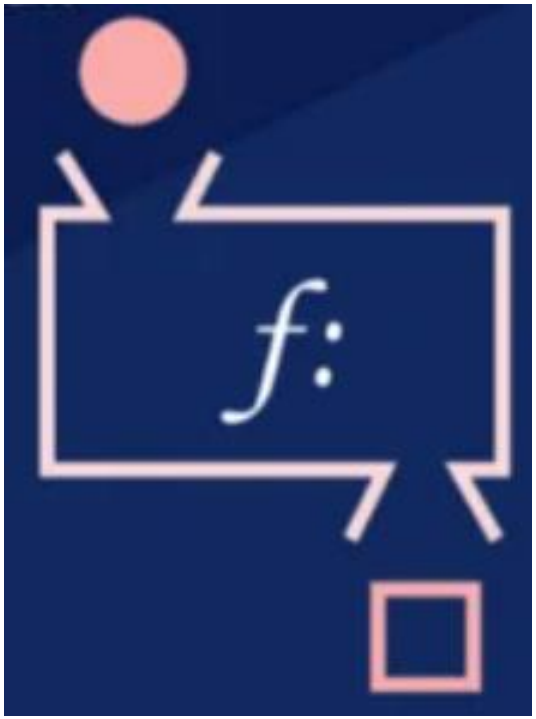
Высокоуровневое программирование

Лекция №10. «Функции в Python»

Калуга - 2020

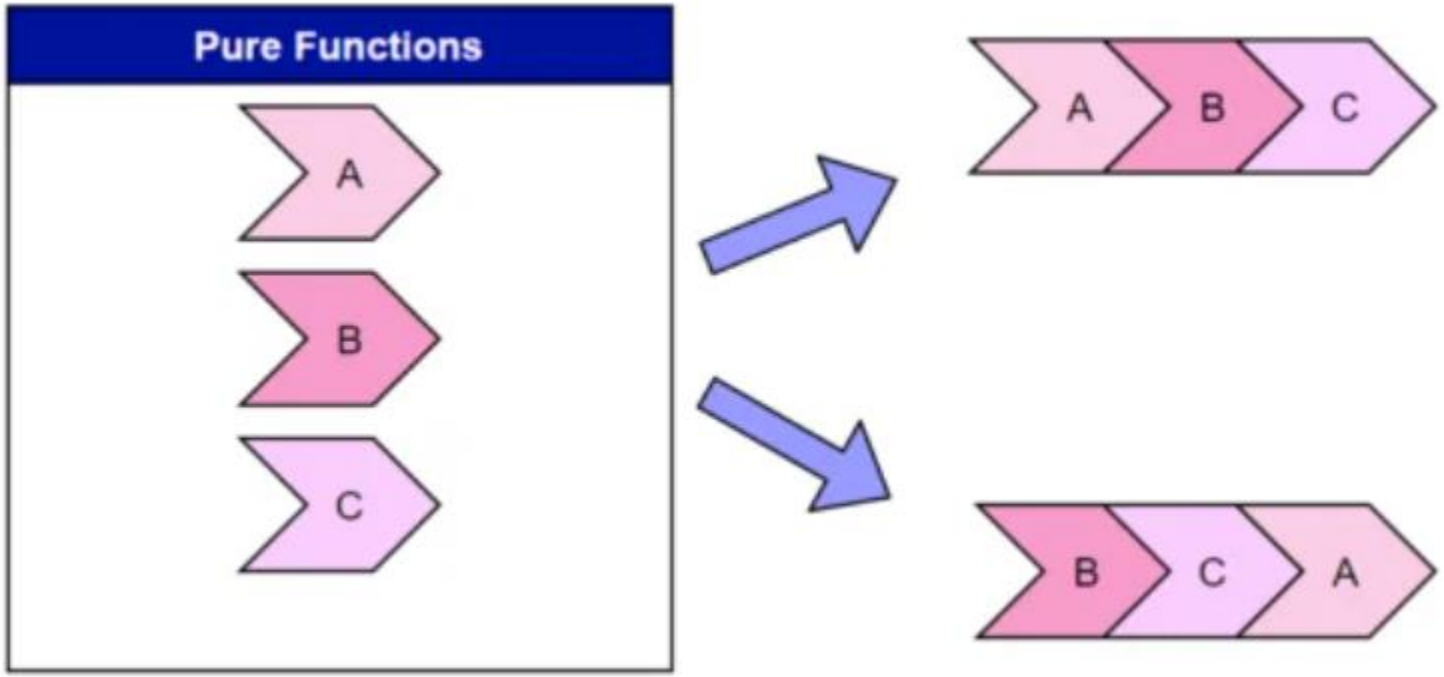
Функциональное программирование

- ***Функциональное программирование*** – это парадигма декларативного программирования, в которой программы создаются путем последовательного применения функций, а не инструкций.
- Каждая из этих функций принимает входное значение и возвращает согласующееся с ним выходное значение, **не изменяясь и не подвергаясь воздействию со стороны состояния программы.**



Функциональное программирование

- *Чистые функции* не производят побочных эффектов и не зависят от глобальных переменных или состояний.



Функциональное программирование

- Для таких функций предусмотрено выполнение только одной операции, если же требуется реализовать сложный процесс, то используется уже композиция функций, связанных последовательно. В процессе ФП мы создаем код, состоящий из множества модулей, поскольку функции в нем могут повторно использоваться в разных частях программы путем вызова, передачи в качестве параметров или возвращения.

Функциональное программирование

- Функциональное программирование (ФП) используется, когда решения легко выражаются с помощью функций и не имеют ощутимой связи с физическим миром.
- В то время как объектно-ориентированные программы моделируют код по образцу реальных объектов, ФП задействует *математические функции, в которых промежуточные или конечные значения не сопоставляются с объектами физического мира*.
- **ФП – реализация формального мира.**

Функциональное программирование

Область применения:

- Проектирование искусственного интеллекта;
- Алгоритмы классификации в МО;
- Финансовые программы;
- Unit-тестирование;
- Многопоточность (Erlang от Ericsson)

Функциональное программирование

Языки ФП:

- Haskell;
- Erlang;
- Clojure;
- F#.

Функциональное программирование

Языки с возможностями ФП:

- Scala;
- JavaScript;
- Python, PHP, C++, Java.

Принципы ФП

Переменные и функции

Ключевыми составляющими функциональной программы являются уже не объекты и методы, а **переменные и функции**.

При этом следует избегать глобальных переменных, потому что изменяемые глобальные переменные усложняют понимание программы и ведут к появлению у функций побочных эффектов.

Принципы ФП

Чистые функции

Для чистых функций характерны два свойства:

- они не создают побочных эффектов;
- они всегда производят одинаковый вывод при получении одинакового ввода, что еще можно называть как ссылочную прозрачность.

Побочные эффекты же возникают, *если функция изменяет состояние программы, переписывает вводную переменную или в общем вносит какие-либо изменения при генерации вывода*. Отсутствие же побочных эффектов снижает риски появления ошибок по вине чистых функций.

Принципы ФП

Неизменяемость и состояния

Неизменяемые данные или состояния не могут изменяться после их определения, что позволяет сохранять постоянство стабильной среды для вывода функций. Лучше всего программировать каждую функцию так, чтобы она выводила один и тот же результат независимо от состояния программы. Если же она зависит от состояния, то это состояние должно быть неизменяемым, чтобы вывод такой функции оставался постоянным.

Принципы ФП

Рекурсия

Одно из серьезных отличий объектно-ориентированного программирования от функционального в том, что программы последнего избегают таких конструкций, как инструкции if else или циклы, которые в разных случаях выполнения могут выдавать разные выводы.

Вместо циклов функциональные программы используют для всех задач по перебору рекурсию.

Принципы ФП

Функции первого класса

Функции в ФП рассматриваются как типы данных и могут использоваться как любое другое значение. Например, мы заполняем функциями массивы, передаем их в качестве параметров или сохраняем их в переменных.

Принципы ФП

Функции высшего порядка

Эти функции могут принимать другие функции в качестве параметров или возвращать функции в качестве вывода. Они делают возможности вызова функций более гибкими и позволяют легче абстрагироваться от действий.

Принципы ФП

Композиция функций

Для выполнения сложных операций функции можно выполнять последовательно. В этом случае результат каждой функции передается следующей функции в виде аргумента. Это позволяет с помощью всего одного вызова функции активировать целую серию их последовательных вызовов.

Функциональное программирование в Python

Самая сложная часть перехода к использованию такого подхода в сокращении числа используемых классов. В Python классы имеют изменяемые атрибуты, что усложняет создание чистых неизменяемых функций.

Попробуйте оформлять весь код на уровне модулей и переключайтесь на классы только по мере необходимости.

Функциональное программирование в Python

- Чистая функция

```
def add_1(x):
    return x + 1
```

Для чистых функций характерны два свойства:

- они не создают побочных эффектов;
- они всегда производят одинаковый вывод при получении одинакового ввода, что еще можно называть как ссылочную прозрачность.

Функциональное программирование в Python

```python
def func(text):
    return text

def shout(text):
    return text.upper()

def greet(f):
    # сохраняем функцию в переменной
    greeting = func("Hi, I am created by a function passed as an argument.")
    print(id(func))
    print(greeting)

greet(shout)
```

```
11298320
Hi, I am created by a function passed as an argument.
```

Декораторы функций

- Декоратор – это функция, которая позволяет обернуть другую функцию для расширения её функциональности без непосредственного изменения её кода.
- Декораторы можно рассматривать как практику функционального программирования, когда программы могут работать с другими программами как со своими данными.

Декораторы функций

```
 1  def A(h):
 2      return lambda x: h(x)*h(7-x)
 3  def B(h):
 4      return lambda x, y: h(x, y)+h(y, x)
 5  def C(h):
 6      return lambda x: h(x, 10-x)
 7
 8  @A
 9  def f(x):
10      return 2*x-1
11  @B
12  def F(x, y):
13      return (8-x)*(y+1)
14  @C
15  def H(x, y):
16      return x*y
17
18  print("f(3) = ", f(3))
19  print("F(5, 7) = ", F(5, 7))
20  print("H(6) = ", H(6))
```

Декораторы функций

```
f(3) =  35
F(5, 7) =  30
H(6) =  24
```

Упаковка и распаковка аргументов

В ряде случаев бывает полезно определить функцию, способную принимать любое число аргументов. Так, например, работает функция print(), которая может принимать на печать различное количество объектов и выводить их на экран.

Достичь такого поведения можно, используя механизм упаковки аргументов, указав при объявлении параметра в функции один из двух символов:

- *: все позиционные аргументы начиная с этой позиции и до конца будут собраны в кортеж;
- **: все ключевые аргументы начиная с этой позиции и до конца будут собраны в словарь.

Упаковка аргументов

```
def print_arguments(* some_tuple, ** some_dict):
    i = 1
    for item in some_tuple:
        print(f'{i} - {item}')
        i += 1


    for key, value in some_dict.items():
        print(f'{key}:  {value}')

print_arguments('Иванов', 'Петров', 'Сидоров', group='ITD-32', year=2020)
```

```
1 - Иванов
2 - Петров
3 - Сидоров
group:  ITD-32
year:  2020
```

Распаковка аргументов

Python также предусматривает и обратный механизм - распаковку аргументов, используя аналогичные обозначения перед аргументом:

- *: кортеж/список распаковывается как отдельные позиционные аргументы и передается в функцию;

- **: словарь распаковывается как набор ключевых аргументов и передается в функцию

Распаковка аргументов

```
def summ(*elements):
    result = 0
    for i in elements:
        result += i
    return result

print(f'sum = {summ(1, 2, 3, 4, 5)}')
print(f'sum = {summ(1, 2, 3, 4, 5, 10)}')
mas = [0, 2, 4, 6]
print(f'sum = {summ(*mas)}')
#такой вызов эквивалентен вызову summ(0, 2, 4, 6)
```

```
sum = 15
sum = 25
sum = 12
```

Что такое *args и **kwargs в Python?

```
1  def print_numbers(a,b,c):
2      print(a, "is stored in a")
3      print(b, "is stored in b")
4      print(c, "is stored in c")
5
6  print_numbers(1,2)
```

```
---------------------------------------
TypeError
<ipython-input-1-46879d054182> in <mod
      3     print(b, "is stored in b")
      4     print(c, "is stored in
----> 5 print_numbers(1,2)

TypeError: print_numbers() missing
```

```
1  def print_numbers(a,b,c=None):
2      print(a, "is stored in a")
3      print(b, "is stored in b")
4      print(c, "is stored in c")
5
6  print_numbers(1,2)
```

```
1 is stored in a
2 is stored in b
None is stored in c
```

```
1  def print_numbers(a=None,b=None,c=None):
2      print(a, "is stored in a")
3      print(b, "is stored in b")
4      print(c, "is stored in c")
5
6  print_numbers(c=3, a=1)
```

```
1 is stored in a
None is stored in b
3 is stored in c
```

Что такое *args и **kwargs в Python?

- Оператор * позволяет «распаковывать» объекты, внутри которых хранятся некие элементы.

```
1  a = [1,2,3]
2  b = [*a,4,5,6]
3  print(b)
```

```
[1, 2, 3, 4, 5, 6]
```

- В этом примере берётся содержимое списка a, распаковывается, и помещается в список b.

Что такое *args и **kwargs в Python?

- ***args** – это сокращение от «arguments» (аргументы)
- ****kwargs** – сокращение от «keyword arguments» (именовнные аргументы)

Каждая из этих конструкций используется для распаковки аргументов соответствующего типа, позволяя вызывать функции со списком аргументов переменной длины.

Что такое *args и **kwargs в Python?

```python
def print_exam_scores(student, *scores):
    print(f"Student Name: {student}")
    for score in scores:
        print(score)

print_exam_scores("Иванов", 100, 95, 88)
print_exam_scores("Петров", 65, 80, 91, 88)
print_exam_scores("Сидоров", 89, 93, 90, 100, 75)
```

```
Student Name: Иванов
100
95
88
Student Name: Петров
65
80
91
88
Student Name: Сидоров
89
93
90
100
75
```

- Нет ли тут ошибки? Ошибки здесь нет.
- Дело в том, что «args» – это всего лишь набор символов, которым принято обозначать аргументы.
- Самое главное тут – это оператор *.

Что такое *args и **kwargs в Python?

```python
def print_faculty_students(department, **students):
    print(f"Name of department: {department}")
    for group, name in students.items():
        print(f"{group}: {name}")

print_faculty_students("ИУК4", group="ИУК4-31Б", names=["Иванов", "Петроа", "Сидоров"])
print_faculty_students("ИУК4", group="ИУК4-32Б", names=["Герасимов", "Федоренко", "Николаев"])
print_faculty_students("ИУК5", group="ИУК5-11Б", names=["Сидоренко", "Антонова", "Носова"])
print_faculty_students("ИУК5", group="ИУК5-12Б", names=["Иванова", "Хохлова", "Ефремов"])
```

```
Name of department: ИУК4
group: ИУК4-31Б
names: ['Иванов', 'Петроа', 'Сидоров']
Name of department: ИУК4
group: ИУК4-32Б
names: ['Герасимов', 'Федоренко', 'Николаев']
Name of department: ИУК5
group: ИУК5-11Б
names: ['Сидоренко', 'Антонова', 'Носова']
Name of department: ИУК5
group: ИУК5-12Б
names: ['Иванова', 'Хохлова', 'Ефремов']
```

Что такое *args и **kwargs в Python? Выводы

- Используйте общепринятые конструкции *args и **kwargs для захвата позиционных и именованных аргументов.
- Конструкцию **kwargs нельзя располагать до *args. Если это сделать – будет выдано сообщение об ошибке.
- Остерегайтесь конфликтов между именованными параметрами и **kwargs, в случаях, когда значение планируется передать как **kwarg-аргумент, но имя ключа этого значения совпадает с именем именованного параметра.
- Оператор *можно использовать не только в объявлениях функций, но и при их вызове.