Программирование в лингвистике

Функции. Рекурсия

Еще две парадигмы программирования

(способы организации кода)

Функциональное программирование	Объектно-ориентированное программирование (ООП)
«Глагольный» подход: мир состоит из списка действий, и субъекты и объекты этих действий не важны.	«Существительный» подход: мир состоит из объектов, у которых есть свои атрибуты и характеристики.
Например, в нашем мире есть функция "приготовить обед", а кто будет в ней участвовать – не важно, хоть котик. Разница будет только в результате.	Например, объект "котик" умеет мурлыкать и мяукать, это его методы. А еще у котика есть уши и хвост – это его атрибуты.

Функции в Python

```
def <имя функции>(*args, **kwargs):
    <тело функции>
    return <возвращаемый объект>
```

Функция – вызываемый объект, содержащий **заранее собранный код**.

Функции часто служат для оптимизации внешнего вида кода. Если нам нужно **повторить одни и те же действия** в разных частях программы, имеет смысл использовать **функции**.

Args (arguments) – это **параметры** функции, аргументы которых необходимо дать в **определенной последовательности**.

Kwargs (keyword arguments) – это **ключевые параметры** функции. Передача аргументов в такие параметры необязательна (рассмотрим это далее).

Не важно, в каком порядке передаются аргументы, если присутствуют **ключи**. Если ключей не присутствует, то последовательность **должна соблюдаться**.

```
def divide(x, y, rounded=False):
  result = x / y
  if rounded:
    result = round(result)
  return result

print(divide(5, 2, rounded=True))
```

```
2
```

По конвенциям определять функции следует в самом начале кода.

Пример функции

```
def divide(x, y, rounded=False):
  result = x / y
  if rounded:
    result = round(result)
  return result

print(divide(5, 2, rounded=True))
```

```
2
```

Эта функция делит первый аргумент (x) на второй (y). Если ключевой аргумент при вызове функции устанавливается и является True, то результат деления округляется.

- def – элемент синтаксиса Python, говорящий ему, что мы создаем функцию.
- divide – наше название функции.
- x, y – args. Здесь это будут числа, и их важно дать в нужном порядке, чтобы наша функция поняла, где делимое, а где делитель.
- rounded=False – kwarg. Ключевой параметр. При передаче аргумент может отсутствовать. Здесь значение rounded по дефолту является False.
- все, что находится внутри функции (с отступом) – ее тело.
- return – элемент синтаксиса Python, предполагающий, что то, что идет за ним через пробел будет результатом выполнения функции (возвращением). Может отсутствовать.
- divide(5, 2, rounded=True) – вызов функции. Здесь x это 5, y это 2, а ключевой аргумент rounded является True.

Параметры args и kwargs

Args – обязательные параметры функции. У них нет дефолтного значения, поэтому они требуют передачи аргумента.
Здесь это number1.

Kwargs – необязательные параметры функции. При создании функции дефолтные значения передаются через знак присвоения.
Здесь это number2, по дефолту равный 4.

```
def print_numbers(number1, number2=4):
  print(number1, number2)

print_numbers(10, 5)
print_numbers(number2=5, number1=10)
print_numbers(10)
```

```
10 5
10 5
10 4
```

Первый раз мы вызываем эту функцию, передав туда оба числа. Number2 оказывается равным 5.
Во второй раз мы делаем то же самое, только дав аргументы вразнобой, указав их ключи.
В третий раз мы передаем в эту функцию только число 10. Number2 остается равным 4.

Аннотирование типов

```
def repeat(s: str, n: int=5) -> str:
  return s * n

print(repeat("a"))
```

```
aaaaa
```

Здесь IDE подскажет Вам, что параметр **s** должен получить **строку**, а параметр **n** – **целое число**. А также напомнит, что **функция** должна возвратить **строку**.

Однако, никакой **ошибки не возникнет**, если что-то из этого пойдет не так.

Можно сделать описание функции **более подробным**. IDE будет давать Вам **подсказки**, основываясь на ее **описании**.

Еще есть оператор **pass**, который ничего не выполняет. Может использоваться для временного заполнения тела функции.

```
def ill_think_later():
  pass
```

Глобальные и локальные переменные

```
x = 5
def create_a():
  a = "a"

print(x)
print(a)
```

```
5

NameError
```

Неймспейс – пространство имен. Это как будто словарь из нашего кода, в котором содержатся все наши объекты под уникальными названиями.

Так, здесь есть следующие элементы неймспейса:
create_a – функция
a – локальная переменная функции
x – глобальная переменная

Глобальные переменные создаются вне функций или классов. Они доступны практически из любого уголка программы, но лишь после их объявления.

Локальные переменные создаются внутри класса или функции, и могут использоваться только на том же уровне вложения.

Здесь попытка напечатать локальную переменную вне функции приводит к ошибке.

Глобальные и локальные переменные

```
def print_1(x):
  x = 1
  print(x)

x = 5
print_1(x)
print('Global x is', x)
```

```
1
Global x is 5
```

Здесь **глобальная** и **локальная** переменные имеют **одно имя**.
Однако, переменная **x** все же равна **5**, даже после того, как в функции мы **изменили** ее значение на **1**. Как так?

Во время работы функции, когда мы **переопределяем** переменную **x**, переменная в глобальном неймспейсе становится **недоступной**. Но когда функция завершает свою работу, **локальная** переменная **x** **перестает существовать**, и на сцену опять выходит **глобальная переменная** со значением **5**.

Рекурсия

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

fact(4)
```

```
24
```

Рекурсия – вызов **функцией самой себя**. При использовании рекурсии, так же как и при использовании цикла **while**, важно предусмотреть **условие выхода**.

В отличие от бесконечного цикла, у рекурсии Python'ом установлена **максимальная глубина рекурсии** (которое можно поменять), поэтому попасть в бесконечную рекурсию не так страшно.

Данная функция вычисляет **факториал**. Как она это делает? Рассмотрим на следующих слайдах.

Вычисление факториала

```python
def fact(n):
  print(f"Функция вызвана со значением n = {n}")
  if n == 0:
    print("Рекурсия начинает возвращать значения!")
    return 1
  factorial = fact(n - 1)
  print(f'Возвращаемое значение: {factorial}')
  return n * factorial

print("Сейчас будет первый вызов функции")
print(f"Итоговое возвращаемое значение: {fact(4)}")
```

```
Сейчас будет первый вызов функции
Функция вызвана со значением n = 4
Функция вызвана со значением n = 3
Функция вызвана со значением n = 2
Функция вызвана со значением n = 1
Функция вызвана со значением n = 0
Рекурсия начинает возвращать значения!
Возвращаемое значение: 1
Возвращаемое значение: 1
Возвращаемое значение: 2
Возвращаемое значение: 6
Итоговое возвращаемое значение: 24
```

Как мы можем видеть, функция рекурсивно «набирает» кучу вызовов самой себя, лишь с постоянно уменьшающимся на единичку n. Как только рекурсия доходит до очевидного return'а (return 1), она начинает сворачиваться, накапливая в n результат выполнения функций, начиная с последней, где, благодаря проверке n == 0, мы получили 1.

Это напоминает следующее выражение:

$4! = 4 \times (3 \times (2 \times (1 \times 1)))$

Вычисление факториала

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

fact(4)
```

```
24
```

> def fact(n) – наша функция, которая принимает параметр n.
> If n == 0 – проверка, условие выхода. Как только постоянно уменьшающееся n дойдет до 0, мы превратим его в 1, чтобы начать сворачивание рекурсии.
> return n * fact(n - 1) – возвращение функцией результата выполнения кода, который вызывает рекурсию.

Мы вызываем эту функцию 5 раз. Первый раз – когда печатаем fact(4), а все остальные разы - в последнем return'е. Каждый раз, когда происходит вызов функции, Python говорит нам: n я знаю, а вот fact(n - 1) мне еще придется вычислить. Тогда он переходит на следующий уровень рекурсии. Это будет продолжаться до тех пор, пока Python не увидит очевидную единичку и не начнет возвращать результаты с самого конца.

Лямбда-функции

Вычисление кубов с помощью def

```
def microfunc(x):
    return x ** 3
```

Вычисление кубов с помощью lambda

```
microfunc = lambda x: x ** 3
```

Когда нам нужно написать **функцию-крошку**, нам необязательно прибегать к классическому синтаксису описания функций.

Вместо этого мы можем создать **лямбда-функцию**, которая позволит записать нужную нам операцию в одну строку.

Лямбда-функции

```
microfunc = lambda x: x ** 3
```

Интерпретация

Что сделать с **аргументом**: возвести **аргумент** в третью степень.

> **microfunc** – название нашей лямбда-функции. Обычно лямбда-функции используются внутри какого-то метода или функции, поэтому объявлять их отдельно необязательно.

> **lambda** – элемент синтаксиса Python, сообщающий ему, что мы начинаем лямбда-функцию.

> **x** – переменная лямбда-функции. Далее с помощью нее мы скажем Python, что делать с аргументом функции.

> **x ** 3** – тело лямбда-функции. То, что мы будем делать с аргументом, переданным в microfunc.

Лямбда-функции

```
words = ["orb", "era", "inc"]
words.sort(key=lambda x: x[-1])
print(*words)
```

```
era orb inc
```

Особенно лямбда-функции полезны в качестве ключей для сортировки.

Здесь x[-1] означает, что каждый элемент будет возвращать последнюю букву, по которой и будет сортироваться список слов.

Спасибо за внимание!

admvereshchagina@gmail.com