Основы Python для Web

Vladimir Maximov

25 сентября 2023 г.

1 Основы Python

1.1 Параметры функций

Символ * позволяет установить, какие параметры будут именнованными - то есть такие параметры, которым можно передать значения только по имени. Все параметры, которые располагаются справа от символа *, получают значения только по имени.

```
1    def print_person(name, *,  age, company):
2       print(f"Name: {name}  Age: {age}  Company: {company}")
3
4    print_person("Bob", age = 41, company ="Microsoft")
5    # Name: Bob  Age: 41  company: Microsoft
```

Если наоборот надо определить параметры, которым можно передавать значения только по позиции, то есть позиционные параметры, то можно использовать символ /: все параметры, которые идут до символа / , являются позиционными и могут получать значения только по позиции.

```
1    def print_person(name, /, age, company="Microsoft"):
2        print(f"Name: {name}  Age: {age}  Company: {company}")
3
4    print_person("Tom", company="JetBrains", age = 24)
5    # Name: Tom  Age: 24  company: JetBrains
6    print_person("Bob", 41)
7    # Name: Bob  Age: 41  company: Microsoft
```

Для одной функции можно определять одновременно позиционные и именнованные параметры.

С помощью символа звездочки можно определить параметр, через который можно передавать неопределенное количество значений. Это может быть полезно, когда мы хотим, чтобы функция получала несколько значений, но мы точно не знаем, сколько именно.

1.2 Преобразование типов

Оба числа в арифметических операциях должны представлять один и тот же тип. Если же два операнда операции представляют разные типы данных, то Python пытается автоматически выполнить преобразования к одному из типов в соответствии со следующми правилами:

- Если один из операндов операции представляет комплексное число (тип complex), то другой операнд также преобразуется к типу complex.
- Иначе, если один из операндов представляет тип float, то второй операнд также преобразуется к типу float.
- Иначе, оба операнда должны представлять тип int, и в этом случае преобазование не требуется.

1.3 Область видимости переменных

Выражение nonlocal прикрепляет идентификатор к переменной из ближайщего окружающего контекста (за исключением глобального контекста). Обычно nonlocal применяется во вложенных функциях, когда надо прикрепить идентификатор за переменной или параметром окружающей внешней функции.

```
1     def outer():  # внешняя функция
2     n = 5
3
4     def inner():    # вложенная функция
5         nonlocal n
6         # указываем, что n - это переменная из окружающей функции
7         n = 25
8         print(n)
9
```

```
10    inner() # 25
11    print(n)
12
13
14   outer() # 25
```

1.4 Замыкания

Замыкание (closure) представляет функцию, которая запоминает свое лексическое окружение даже в том случае, когда она выполняется вне своей области видимости.

Технически замыкание включает три компонента:

- внешняя функция, которая определяет некоторую область видимости и в которой определены некоторые переменные и параметры - лексическое окружение
- переменные и параметры (лексическое окружение), которые определены во внешней функции
- вложенная функция, которая использует переменные и параметры внешней функции

Для определения замыканий в Python применяются локальные функции:

```
1     def outer():         # внешняя функция
2     n = 5           # лексическое окружение - локальная переменная
3
4     def inner():      # локальная функция
5         nonlocal n
6         n += 1        # операции с лексическим окружением
7         print(n)
8
9     return inner
10
11    fn = outer()
12    # fn = inner, так как функция outer возвращает функцию inner
13    # вызываем внутреннюю функцию inner
14    fn()    # 6
15    fn()    # 7
16    fn()    # 8
```

Кроме внешних переменных к лексическому окружению также относятся параметры окружающей функции. Рассмотрим использование параметров:

```
1    def multiply(n):
2        def inner(m): return n * m
3
4    return inner
5
6   fn = multiply(5)
7   print(fn(5))        # 25
8   print(fn(6))        # 30
9   print(fn(7))        # 35
```

1.5 Декараторы

Декораторы в Python представляют функцию, которая в качестве параметра получает функцию и в качестве результата также возвращает функцию. Декораторы позволяют модифицировать выполняемую функцию, значения ее параметров и ее результат без изменения исходного кода этой функции.

```
1     # определение функции декоратора
2     def select(input_func):
3         def output_func():
4             # определяем функцию, выполняющуюся вместо input_func
5             print("******************")
6             # перед выводом оригинальной функции выводим звездочки
7             input_func()
8             # вызов оригинальной функции
9             print("******************")
10            # после вывода оригинальной функции выводим звездочки
11        return output_func     # возвращаем новую функцию
12
13    # определение оригинальной функции
14    @select         # применение декоратора select
15    def hello():
16        print("Hello")
17
18    # вызов оригинальной функции
19    hello()
```

Декоратор может перехватывать передаваемые в функцию аргументы:

```
1     # определение функции декоратора
2     def check(input_func):
3        def output_func(*args):
4             # через *args получаем значения параметров ориг. функции
5             input_func(*args)
6             # вызов оригинальной функции
7         return output_func
8         # возвращаем новую функцию
9
10    # определение оригинальной функции
11    @check
12    def print_person(name, age):
13        print(f"Name: {name}  Age: {age}")
14
15    # вызов оригинальной функции
16    print_person("Tom", 38)
```

Здесь функция print-person() принимает два параметра: name (имя) и age (возраст). К этой функции применяется декоратор check()

В декораторе check возвращается локальная функция output-func(), которая принимает некоторый набор значений в виде параметра *args - это те значения, которые передаются в оригинальную функцию, к которой применяется декоратор. То есть в данном случае *args будет содержать значения параметров name и age.

```
1     # определение функции декоратора
2     def check(input_func):
3         def output_func(*args):
4             name = args[0]
5             age = args[1]
6             # получаем значение второго параметра
7             if age < 0: age = 1
8             # если возраст отрицательный, изменяем его на 1
9             input_func(name, age)
10            # передаем функции значения для параметров
11        return output_func
12
13    # определение оригинальной функции
```

```
14    @check
15    def print_person(name, age):
16        print(f"Name: {name}  Age: {age}")
17
18    # вызов оригинальной функции
19    print_person("Tom", 38)
20    print_person("Bob", -5)
```

1.6 Классы и объекты

Если мы определили в классе инициализатор $\_\_init\_\_$, мы уже не сможем вызвать инициализатор по умолчанию. Теперь нам надо вызывать наш явным образом опреределенный инициализатор $\_\_init\_\_$, в который необходимо передать значение для параметра.

1.6.1 classmethod

```
@classmethod
def f(cls, x, y):
    if x > 0:
        if y > 0:
            return cls(x, y)
```

Класс метод нужен чтобы разделить логику и инициализацию. cls - ссылка на класс. В насследованных классах ссылка меняется на текущий класс.

1.7 Магические методы

Это специальные методы, с помощью которых вы можете добавить в ваши классы «магию». Они всегда обрамлены двумя нижними подчеркиваниями (например, __init__ или __lt__). Используются Python во время работы.

1.8 Инкапсуляция, атрибуты и свойства

У понятия инкапсуляция есть 2 трактовки:

- Объединение в один объект как данных, так и методов работы с ними
- Сокрытие внутренней структуры объекта от внешний воздействий

Касательно инкапсуляции непосредственно в языке программирования Python скрыть атрибуты класса можно сделав их приватными или закрытыми и ограничив доступ к ним через специальные методы, которые еще называются свойствами.

Для создания приватного атрибута в начале его наименования ставится двойной прочерк: $self.\_\_name$. К такому атрибуту мы сможем обратиться только из того же класса. Но не сможем обратиться вне этого класса.

Python имеет также еще один - более элегантный способ определения свойств. Этот способ предполагает использование аннотаций, которые предваряются символом @.

Для создания свойства-геттера над свойством ставится аннотация @property.

Для создания свойства-сеттера над свойством устанавливается аннотация имя_свойства_геттера.setter.

Во-первых, стоит обратить внимание, что свойство-сеттер определяется после свойства-геттера.

Во-вторых, и сеттер, и геттер называются одинаково - age. И поскольку геттер называется age, то над сеттером устанавливается аннотация @age.setter.

После этого, что к геттеру, что к сеттеру, мы обращаемся через выражение tom.age.

1.9 Наследование

Наследование позволяет создавать новый класс на основе уже существующего класса. Ключевыми понятиями наследования являются подкласс и суперкласс. Подкласс наследует от суперкласса все публичные атрибуты и методы. Суперкласс еще называется базовым (base class) или родительским (parent class), а подкласс - производным (derived class) или дочерним (child class).

Синтаксис для наследования классов выглядит следующим образом:

```
1   class подкласс (суперкласс):
2         методы_подкласса
```

Одной из отличительных особенностей языка Python является поддержка множественного наследования, то есть один класс можно унаследовать от нескольких классов:

```
1     #  класс работника
2     class Employee:
3         def work(self):
4             print("Employee works")
5
6     #  класс студента
7     class Student:
8         def study(self):
9             print("Student studies")
10
11    class WorkingStudent(Employee, Student):
12        # Наследование от классов Employee и Student
13        pass
14
15    # класс работающего студента
16    tom = WorkingStudent()
17    tom.work()      # Employee works
18    tom.study()     # Student studies
```

Все классы в Python наследуются от класса object. Это базовый класс языка. И поэтому технически все классы, встроенные или определенные пользователем, являются наследуемыми, а все объекты — экземплярами класса object.

По порядку разрешения методов любой указанный атрибут сначала ищется в объявленном классе. Если его там нет, поиск продолжается в родительских классах на максимальную глубину слева направо без прохода по одному классу дважды.

Такой порядок еще называется линеаризацией класса MultiDerived, а список правил, по которому мы находим такой порядок, называется Method Resolution Order (порядок разрешения методов).

MRO класса можно просмотреть в атрибуте __mro__ или с помощью метода mro(). Вызов атрибута возвращает кортеж, а вызов метода — список.

1.10 Переопределение функционала базового класса

Если в базом классе определен конструктор с помощью метода __init__, и мы хотим в производном классе изменить логику конструктора, то в конструкторе производного класса мы должны вызвать конструктор базового класса.

```
1     class Person:
2         def __init__(self, name):
3             self.__name = name   # имя человека
4
5         @property
6         def name(self):
7             return self.__name
8
9         def display_info(self):
10            print(f"Name: {self.__name}")
11
12    class Employee(Person):
13        def __init__(self, name, company):
14            super().__init__(name)
15            self.company = company
16
17        def display_info(self):
18            super().display_info()
19            print(f"Company: {self.company}")
20
21        def work(self):
22            print(f"{self.name} works")
```

Здесь в классе Employee добавляется новый атрибут - self.company, который хранит компанию работника. Соответственно метод __init__() принимает три параметра: второй для установки имени и третий для установки компании. Но если в базом классе определен конструктор с помощью метода __init__, и мы хотим в производном классе изменить логику конструктора, то в конструкторе производного класса мы должны вызвать конструктор базового класса. То есть в конструкторе Employee надо вызвать конструктор класса Person.

1.11 Проверка типа объекта

При работе с объектами бывает необходимо в зависимости от их типа выполнить те или иные операции. И с помощью встроенной функции isinstance() мы можем проверить тип объекта. Эта функция принимает два параметра:

```
1   isinstance(object, type)
```

1.12 Датакласс

```
@dataclass
class Point:
    x: float
    y: float
```

1.13 Исключения

При обработке исключений также можно использовать необязательный блок finally. Отличительной особенностью этого блока является то, что он выполняется вне зависимости, было ли сгенерировано исключение.

```
1   try:
2       number = int(input("Введите число: "))
3       print("Введенное число:", number)
4   except:
5       print("Преобразование прошло неудачно")
6   finally:
7       print("Блок try завершил выполнение")
```

В Python есть следующие базовые типы исключений:

- BaseException: базовый тип для всех встроенных исключений
- Exception: базовый тип, который обычно применяется для создания своих типов исключений
- ArithmeticError: базовый тип для исключений, связанных с арифметическими операциями (OverflowError, ZeroDivisionError, FloatingPointError).
- BufferError: тип исключения, которое возникает при невозможности выполнить операцию с буффером
- LookupError: базовый тип для исключений, которое возникают при обращении в коллекциях по некорректному ключу или индексу (например, IndexError, KeyError)

И если ситуация такова, что в программе могут быть сгенерированы различные типы исключений, то мы можем их обработать по отдельности, используя дополнительные выражения except. И при возникновении исключения Python будет искать нужный блок except, который обрабатывает данный тип исключения:

```
1     try:
2         number1 = int(input("Введите первое число: "))
3         number2 = int(input("Введите второе число: "))
4         print("Результат деления:", number1/number2)
5     except ValueError:
6         print("Преобразование прошло неудачно")
7     except ZeroDivisionError:
8         print("Попытка деления числа на ноль")
9     except BaseException:
10        print("Общее исключение")
```

Python позволяет в одном блоке except обрабатывать сразу несколько типов исключений. В этом случае все типы исключения передаются в скобках:

```
1    try:
2        number1 = int(input("Введите первое число: "))
3        number2 = int(input("Введите второе число: "))
4        print("Результат деления:", number1/number2)
5    except (ZeroDivisionError, ValueError):
6        #  обработка двух типов исключений -
7        #  ZeroDivisionError и ValueError
8        print("Деление числа на ноль или некорректный ввод")
```

С помощью оператора as мы можем передать всю информацию об исключении в переменную, которую затем можно использовать в блоке except:

```
1    try:
2        number = int(input("Введите число: "))
3        print("Введенное число:", number)
4    except ValueError as e:
5        print("Сведения об исключении", e)
```

Иногда возникает необходимость вручную сгенерировать то или иное исключение. Для этого применяется оператор raise. Например, сгенерируем исключение:

```
1     try:
2         number1 = int(input("Введите первое число: "))
3         number2 = int(input("Введите второе число: "))
4         if number2 == 0:
```

```
5               raise Exception("Второе число не 0")
6           print("Результат деления двух чисел:", number1/number2)
7       except ValueError:
8           print("Введены некорректные данные")
9       except Exception as e:
10          print(e)
```

Оператору raise передается объект BaseException - в данном случае объект Exception. В конструктор этого типа можно ему передать сообщение, которое затем можно вывести пользователю. В итоге, если number2 будет равно 0, то сработает оператор raise, который сгенерирует исключение. В итоге управление программой перейдет к блоку except, который обрабатывает исключения типа Exception.

```
1       class PersonAgeException(Exception):
2           def __init__(self, age, minage, maxage):
3               self.age = age
4               self.minage = minage
5               self.maxage = maxage
6
7           def __str__(self):
8               return f"Недопустимое значение: {self.age}. " \
9                   f"Возраст в диапазоне от {self.minage} до {self.maxage}"
10
11      class Person:
12          def __init__(self, name, age):
13              self.__name = name  # устанавливаем имя
14              minage, maxage = 1, 110
15              if minage < age < maxage:
16                  # устанавливаем возраст, если корректное значение
17                  self.__age = age
18              else:
19                  # иначе генерируем исключение
20                  raise PersonAgeException(age, minage, maxage)
```

В начале здесь определен класс PersonAgeException, который наследуется от класса Exception. Как правило, собственные классы исключений наследуются от класса Exception. Класс PersonAgeException предназначен для исключений, связанных с возрастом пользователя.

1.14 Структуры данных в Python

В питоне нет массива, только списки.

```
1   a = [1, 2, 3] # list []
2   b = (1, 2, 3) # tuple ()
3   c = {1, 2, 3} # set set()
4   d = {1: 1, 2: 2, 3: 3, 4: 4} # dict {}
```

В списки складываем только числа (целые, вещественные).

Когда мы хотим складывать данные с разными типами, мы используем кортеж.

Ключи словаря хранятся во множестве. Отличие словаря от множества в том, что при хранении ключа во множестве также хранится ссылка на объект.

1.15 Работа со списками

Списки нужно использовать для:

- list comprehention
- filter, map, reduce

```
1    xs = [int(input()) for _ in range(15)]
2    xs = {int(input()) for _ in range(15)}
3    xs = tuple(int(input()) for _ in range(15))
4    xs = {i: int(input()) for _ in range(15)}
```

Использование map допускается только для задач, в которых необходимо преобразовать каждый элемент в списке.