Въведение в Python 2-ра част

Множества (set)

Всеки елемент се съдържа точно един път. Елементите на множествата не са подредени. Създават се със служебната дума set ([списък / комплект]) или {елемент 1 , елемент 2 , ...,елемент n }. Множеството трябва да съдържа поне един елемент.

In [ ]:

```
x = {3,1,2,5,5}
y = set([5,3,1])
z = set()

A={1,2,3}
B={2,1,4}

print("A=", A)
print("B=", B)

# Обединение на множества (union)
C = A.union(B)
# или
C = A | B

print("Union:", A, "|", B, "=", C)

# Сечение на множества (intersection)
C = A.intersection(B)
# или
C = A & B

print("Intersection:", A, "&", B, "=", C)

# Разлика на множества (difference)
C = A.difference(B)
# или
C = A - B

print("Difference:", A, "-", B, "=", C)

# Симетрична разлика на множества (symmetric difference)
C = A.symmetric_difference(B)
# или
C = A ^ B

print("Symmetric difference:", A, "^", B, "=", C)

# Проверка за подмножество

C = {1, 2}

# Множеството C е подмножество на A
print(C, "is a subset of", A, ":", A > C)
```

```
A= {1, 2, 3}
B= {1, 2, 4}
Union: {1, 2, 3} | {1, 2, 4} = {1, 2, 3, 4}
Intersection: {1, 2, 3} & {1, 2, 4} = {1, 2}
Difference: {1, 2, 3} - {1, 2, 4} = {3}
Symmetric difference: {1, 2, 3} ^ {1, 2, 4} = {3, 4}
{1, 2} is a subset of {1, 2, 3} : True
```

Работа с файлове

Отваряне на файл

`open()` - връща файлов обект (object) и най-често се използва с два аргумента: Формат на функцията: `open(име_на_файл, режим)`.

- Първи аргумент – име на файла
- Режими:
  - `r` – файлът се отваря само за четене
  - `w` – създава се нов файл и се отваря само за запис (съществуващ файл със същото име ще бъде изтрит)
  - `a` – файлът се отваря за запис, като добавя съдържанието в края на файла, ако той съществува
  - `t` – текстов режим (по подразбиране)
  - `r +` – файлът се отваря за четене и запис
  - `a +` – файлът се отваря за четене и запис, като добавя съдържанието в края на файла, ако той съществува

Аргументът режим е незадължителен; ако е пропуснат, `t` се подразбира.

Четене от файл

Функция read() - без параметри прочита целия текстов файл (всички символи)

```
file = open("file.txt", "r")
print(file.read())
```

Функция read(char_numbers) - прочита char_numbers символи от файла

```
file = open("file.txt", "r")
print(file.read(100))
```

Метод readlines() - прочита всички редове от файла. За четене на файл ред по ред:

```
file = open("file.txt", "r")
print(file.readlines())
```

или

```
file = open("file.txt", "r")
for line in file:
  print(line)
```

Запис във файл

Метод write(string) - запис на символния низ string във файла:

```
file = open("file.txt","w")
file.write("Line1\n")
file.write("Line2.")
file.close()
```

Затваряне на файл

След като приключите работа с файл, трябва да го затворите с метод `close()`

Достъп до файл с оператор `with`

Операторът `with` предоставя по-ясен синтаксис за обработване на файл. Негово предимство е, че автоматично затваря всеки отворен файл след приключване на файловите операции.

Формат:

with open(“filename”) as file:

```
with open("file.txt") as f:
  for line in f:
    print(line)
```

За писане във файл:

```
with open("out.txt","w") as f:
  f.write("hello\n")
  f.write("123456789\n")
  f.write("abcdef\n")
```

Функции

Функцията е блок от код, който се изпълнява само когато е извикан. Можете да предавате данни (параметри) към функцията и функцията може да върне данни като резултат.

Създаване на функция

В Python функция се дефинира с помощта на ключовата дума `def`:

In [ ]:
```
def func():
  print("Hello!")

func()
```

```
Hello!
```

Аргументи

Информацията може да се предава към функциите като аргументи.

Аргументите се посочват в скобите след името на функцията. Може да се добавят колкото искате аргументи, разделени със запетая.

Следващия пример показва функция с два аргумента `first_name` и `last_name` и отпечатва пълното име:

In [ ]:
```
def print_name(first_name, last_name):
  print(str(first_name) + " " + last_name)

print_name("John", "Smith")
```

```
John Smith
```

Произволни аргументи, *args

Ако не знаете колко аргументи ще бъдат предадени във вашата функция, добавете * преди името на параметъра в дефиницията на функцията.

По този начин функцията ще получи набор / комплект от аргументи:

In [ ]:
```
def print_names(*names):
  for name in names:
    print(name)
  print(names)

print_names("John Smith", "Catherine Johnson", "David Johnson")
```

```
John Smith
Catherine Johnson
David Johnson
('John Smith', 'Catherine Johnson', 'David Johnson')
```

Аргументи с ключови думи

Можете също да изпращате аргументи със `key = value` синтаксис.

По този начин редът на аргументите няма значение.

In [ ]:
```
def print_students(student1, student2, student3):
  print("Student1:", student1)
  print("Student2:", student2)
  print("Student3:", student3)

print_students("James", student3="John", student2="David")
```

```
Student1: James
Student2: David
Student3: John
```

Произволни аргументи с ключови думи, kwargs**

Ако не знаете колко аргумента с ключови думи ще бъдат предадени във вашата функция, добавете две звездички: ** преди името на параметъра в дефиницията на функцията.

По този начин функцията ще получи речник (dict) с аргументи:

In [ ]:
```
def print_person(**person):
  print(person)
  print("First name:",person["first_name"])
  print("Last name:",person["last_name"])

print_person(first_name = "David", last_name = "Wood")
```

```
{'first_name': 'David', 'last_name': 'Wood'}
First name: David
Last name: Wood
```

Комбиниране на фиксирани параметри, *args и kwargs**

In [ ]:
```
def print_info(id, *args, **kwargs):
  print(id, args, kwargs)

print_info(1)
print_info(2, "a1", "a2", "a3", "a4", param1="p1", param2="p2")
```

```
1 () {}
2 ('a1', 'a2', 'a3', 'a4') {'param1': 'p1', 'param2': 'p2'}
```

Предаване на аргументи през речник / dict

Аргументи могат да бъдат предадени на функция през речник, като всяка двойка ключ / стойност представлява един аргумент на функцията и ключа е стринг с името на аргумента.

За да се предадат аргументите се използва ** преди речника при изпълнението на функцията:

In [ ]:
```
def print_students(student1, student2, student3):
  print("Student1:", student1)
  print("Student2:", student2)
  print("Student3:", student3)
```

```
students={ "student1": "James", "student2": "David", "student3": "John"}
print_students(**students)
```

```
Student1: James
Student2: David
Student3: John
```

Параметри със стойност по подразбиране

Следващият пример показва как да зададете стойност по подразбиране на определени аргументи на функция. Аргументите със стойности по подразбиране трябва да са последни в списъка с аргументи на функцията.

Ако извикаме функцията без тези аргументи, тя използва стойността по подразбиране:

In [ ]:
```
def my_function(name, city = "Plovdiv", country = "Bulgaria"):
  print(name, "is from", city + "," + country)

# wrong. non-default arguments should be defined before default ones.
#def my_function(city = "Plovdiv", country = "Bulgaria", name):

my_function("Francois", "Paris", "France")
my_function("Anna", "Munich", country="Germany")
my_function("George")
my_function("Nikola", city="Sofia")
my_function("Juan", country="Argentina", city="Bueno Aires")
```

```
Francois is from Paris,France
Anna is from Munich,Germany
George is from Plovdiv,Bulgaria
Nikola is from Sofia,Bulgaria
Juan is from Bueno Aires,Argentina
```

Рекурсия

В python можете да ползвате рекурсия - функциите може да се извикват сами себе си.

In [ ]:
```
def fib(n):
  if (n<=1): return n
  return fib(n-1) + fib(n-2)

print("Fibonacci(20) =", fib(20))
```

```
Fibonacci(20) = 6765
```

Ламбда (lambda) функции

Ламбда функция е малка анонимна функция.

Ламбда функция може да приема произволен брой аргументи, но може да има само един израз:

In [ ]:
```
# with one argument
x = lambda a : a * 2
print(x(5))

# with more arguments
x = lambda a, b : a * b
print(x(5, 3))

# with *args and **kwargs
p = lambda *args, **kwargs: print(*args, **kwargs)
p(10,20,30,sep=",")
# same as print(10,20,30,sep=",")
```

```
10
15
10,20,30
```

Силата на ламбда функциите се показва най-добре, когато се използват като анонимна функция в друга функция.

Да предположим, че имате дефиниция на функция, която приема един аргумент и този аргумент ще бъде умножен с неизвестно число:

In [ ]:
```
def myfunc(n):
  return lambda a : a * n
```

Използваме тази дефиниция на функция, за да направим функция, която винаги удвоява или утроява номера, който се изпраща:

In [ ]:
```
def myfunc(n):
  return lambda a : a * n
```

```
mydoubler = myfunc(2) # same as mydouble = lambda a: a * 2
mytripler = myfunc(3) # same as mydouble = lambda a: a * 3

print(mydoubler(11))
print(mytripler(11))
```

```
22
33
```

Класове и обекти в Python

Python е обектно ориентиран език за програмиране.

- Всичко е обект - дори функциите и модулите.
- Обектите и класовете са динамични - може да ги разширявате с полета и методи по време на изпълнение
- Има множествено наследяване.
- Има „нов стил“ класове и „класически“ класове.
- Може да предефинирате оператори.
- Има и метакласове - както "клас" е тип на обекта, така "метаклас" е тип на класа.

Създаване на клас

Клас се дефинира чрез ключовата дума `class`:

In [ ]:
```
class MyClass:
  x = 10
```

Създаване на обект

Сега можем да използваме нашия клас за да създаваме обекти:

In [ ]:
```
m = MyClass()
print(m.x)
```

```
10
```

Функцията __init__()

Примерите по-горе са класове и обекти в най-простата им форма и не са много полезни в реални приложения.

За да разберем значението на класовете, трябва да разберем вградената функция __init__().

Всички класове имат функция, наречена __init__(), която винаги се изпълнява при инициализиране на класа. Също известна като конструктор на класа.

Използвайте функцията __init__(), за да присвоите стойности на свойствата на обекта или други операции, които са необходими, когато обектът се създава:

In [ ]:
```
class Person:
  name = "John"
  def __init__(self, name, age):
    self.name = name
    self.age = age

p1 = Person("James", 20)

print(p1.name)
print(p1.age)
```

```
James
20
```

Обектни методи

Обектите също могат да съдържат методи. Методите в обектите са функции, които принадлежат на обекта.

Нека създадем метод в класа Person:

In [ ]:
```
class Person:
  def __init__(self, name, age):
    self.name = name
    self.age = age

  def print_info(self):
    print("My name is", self.name, "and I'm", self.age, "years old")

p1 = Person("James", 20)
p1.print_info()
```

```
My name is James and I'm 20 years old
```

`self` параметърът

Параметърът `self` е препратка към текущата инстанция на класа и се използва за достъп до променливи, които принадлежат към класа.

Не е задължително да се именува `self`, можете да го наричате каквото искате, но трябва да е първият параметър на която и да е функция в класа:

In [ ]:
```
class Person:
  def __init__(this, name, age):
    this.name = name
    this.age = age

  def print_info(abc):
    print("My name is", abc.name, "and I'm", abc.age, "years old")

p1 = Person("James", 20)
p1.print_info()
```

```
My name is James and I'm 20 years old
```

Изтриване на полета и обекти

За динамично итриване на полета и цели бекти може да ползвате ключовата дума `del`

In [ ]:
```
del p1.age

del p1
```

Метод __call__

Метод __call__ дава възможност да се пишат класове, където обектите се държат като функции и могат да бъдат извикани като функция. Ако __call__ е дефиниран, x(arg1, arg2, ...) е същото като x.__call__(arg1, arg2, ...).

In [ ]:
```
class Multiply:
  def __init__(self, multiplier):
    self.multiplier = multiplier

  def __call__(self, a):
    return a * self.multiplier

doubler = Multiply(2)
tripler = Multiply(3)

print(doubler(111)) # same as print(doubler.__call__(111))
print(tripler(111))
```

```
222
333
```

Наследяване в Python

Наследяването ни позволява да дефинираме клас, който наследява всички методи и свойства от друг клас.

Parent клас е класът, който се наследява, наричан още основен клас.

Child клас е класът, който наследява от друг клас, наричан още производен клас.

Създаваме parent клас:

In [ ]:
```
class Person:
  def __init__(self, first_name, last_name):
    self.firstname = first_name
    self.lastname = last_name

  def print_info(self):
    print(self.firstname, self.lastname)

x = Person("John", "Smith")
x.print_info()
```

```
John Smith
```

Създаваме класове наследници (child classes):

In [ ]:
```
class Student(Person):
  def __init__(self, first_name, last_name, year):
    # call the parent constructor
    #Person.__init__(first_name, last_name)
    super().__init__(first_name, last_name)
```

```
    self.graduationyear = year

  def print_info(self):
    print("Student:", self.firstname, self.lastname, self.graduationyear)

class Employee(Person):
  def __init__(self, first_name, last_name, year):
    super().__init__(first_name, last_name)
    self.startyear = year

  def print_info(self):
    #super().print_info()
    print("Employee:", self.firstname, self.lastname, self.startyear)


def print_people_info(*people):
  for person in people:
    person.print_info()

p1 = Person("Petar", "Petrov")
s1 = Student("George", "Smith", 2023)
e1 = Employee("David", "Johnson", 2020)

print_people_info(p1,s1,e1)
```

```
Petar Petrov
Student: George Smith 2023
Employee: David Johnson 2020
```

Модули в Python

Модул в Python е същото като библиотека с код (code library).

Той е файл, съдържащ набор от функции, които можете да включите и използвате във вашето приложение.

Създаване на модул

За да създадете модул, просто запазете желания код във файл с разширение .py:

Записваме това във файл с име `mymodule.py`

```
def greeting(name):
  print("Hello, " + name)
```

Използване на модул

Сега можем да използваме току-що създадения модул, като използваме оператора `import`:

```
import mymodule

mymodule.greeting("Jonathan")
```

Променливи в модулите

Модулът може да съдържа функции, но също така и променливи от всякакъв тип (масиви, речници, обекти и т.н.): Добавяме следния код към `mymodule.py`:

```
person1 = {
  "name": "John",
  "age": 20,
  "country": "Bulgaria"
}
```

Сега можем да достъпим person1 речника:

```
import mymodule

a = mymodule.person1["age"]
print(a)
```

Наименоване на модул

Можете да ползвате каквото искате име на модула. Само трябва да завършва с `.py`

Преименуване на модул

Можете да дефинирате псевдоним (alias) на модула, когато използвате import:

```
import mymodule as mx

a = mx.person1["age"]
print(a)
```

Импортиране от модул

Можете да изберете да импортирате само части от модул, като използвате ключовата дума `from`: Импортиране само на person1 от `mymodule`:

```
from mymodule import person1

print(person1["age"])
```

Импортиране на всичко от модул:

```
from mymodule import *

print(person1["age"])
greeting("Anonymous")
```

In [ ]:

```
# example how to install libraries in Google Colab
!pip install numpy
```

```
Requirement already satisfied: numpy in /usr/local/lib/python3.7/dist-packages (1.19.5)
```