Тема 4_2. Обектно-ориентирано програмиране. Класове и обекти. Капсулиране, атрибути и свойства. Наследяване. Предефиниране на функционалността на базовия клас. Клас атрибути и статични методи. Низово представяне на обект

Обектно-ориентирано програмиране

Класове и обекти

Python има много вградени типове като int, str и други, които могат да се използват при разработване на програми. Но Python също позволява дефиниране на собствени типове. Класът представлява обект. Конкретната реализация на класа е обект.

Аналог на клас от реалния свят е наример човека, той има име, възраст и някакви други характеристики. Човекът може да извършва определени действия - да ходи, да бяга, да мисли и т.н. Или изгледа, който включва набор от характеристики и действия, може да се нарече клас. Конкретното изпълнение на този модел може да варира, например, някои хора имат едно име, други имат друго име. И конкретния човек ще представлява обект от този клас. Класът се дефинира с помощта на ключовата дума class:

```
class име_на_класа:

    атрибути_на_класа
    методи_на_класа
```

Вътре в класа се дефинират атрибутите, които съхраняват различни характеристики на класа, а методите са функциите на класа. Нека да се създаде елементарен клас:

```
class Person:
    pass
```

В този случай се дефинира класът Person, който условно представлява лице. В този случай в класа не са дефинирани никакви методи или атрибути. Въпреки това, тъй като нещо трябва да бъде дефинирано в него, операторът **pass** се използва като заместител на функционалността на класа. Този израз се използва, когато синтактично е необходимо да се дефинира определен код, но ако това не бъде направено, вместо конкретен код може да се вмъкне оператора **pass**.

След като класът е създаден, обектите от този клас могат да бъдат дефинирани. **Например**:

```
class Person:
    pass
tom = Person()      # дефиниран обект tom
bob = Person()      # дефиниран обект bob
```

След като класът Person е дефиниран, се създават два обекта от класа Person, tom и bob. За създаване на обект се използва специална функция - конструктор, който се извиква от името на класа и който връща обект от класа. Т.е. в случая повикването Person() представлява извикване към конструктора. Всеки клас има конструктор по подразбиране без параметри:

```
tom = Person()   # Person() - извикване на конструктор, който връща обект от класа Person
```

Класови методи

Методите на класа всъщност представляват функции, които са дефинирани в класа и които определят поведението му. Например, нека дефинираме клас Person с един метод:

```
class Person:
    def say_hello(self):
        print("Hello")
tom = Person()
tom.say_hello()          # Hello
```

```
main.py                                   Run   Shell
1 class Person:              # дефиниране на клас Person      Hello
2     def say_hello(self):                                     >
3         print("Hello")
4 tom = Person()
5 tom.say_hello()            # Hello
```

Тук е дефиниран методът say_hello(), който условно изпълнява Hello и отпечатва низ в конзолата. Когато се дефинират методи на всеки клас, имайте предвид, че всички те трябва да приемат като свой първи параметър препратка към текущия обект, който по конвенция се нарича self. Чрез тази препратка в класа може да се получи достъп до функционалността на текущия обект. Но когато методът се извика, този параметър не се взема предвид. Използвайки името на обект, може да се обърне към неговите методи. За достъп до методите се използва точкова нотация - след името на обекта се поставя точка и след нея идва извикването на метода:

обект.метод([параметри на метода])

Например, извикване на метод say_hello() за показване на поздрав в конзолата:

```
tom.say_hello()    # Hello
```

В резултат на това тази програма ще отпечата низа "Hello" в конзолата. Ако трябва методът да приеме други параметри, те се дефинират след параметъра self и когато трябва да се извиква такъв метод, трябва да се предадат стойности за тях:

```
class Person:
    def say(self, message):
        print(message)
tom = Person()
tom.say("Hello abv.bg")
```

```
main.py                                     Run    Shell
class Person:                    # дефиниран клас Person     Hello abv.bg
    def say(self, message):      # метод                     >
        print(message)
tom = Person()
tom.say("Hello abv.bg")      # Hello abv.bg
```

Методът дефиниран тук е say(). Необходими са два параметъра: self и message. А за втория параметър - message, при извикване на метода трябва да предадете стойност.

self

Чрез ключовата дума self може да получите достъп до функционалността на текущия обект в класа:

```
self.атрибут        # обръщение към атрибут
self.метод          # обръщение към метод
```

Например, нека се дефинират два метода в класа Person:

```
class Person:

    def say(self, message):
        print(message)

    def say_hello(self):
        self.say("Hello work")        # позовавайки се на метода say


tom = Person()
tom.say_hello()                       # Hello work
```

Тук в един метод - say_hello() се извиква друг метод - say():

```
self.say("Hello work")
```

Тъй като методът say() приема, в допълнение към self, параметри (параметъра message), когато методът се извика, се предава стойност за този параметър. Освен това, когато се извиква обектен метод, определено трябва да се използва думата self, ако не се използва се извежда грешка:

```
    def say_hello(self):
say("Hello work")           # Error
```

Конструктори

За създаване на обект от клас се използва конструктор. Когато се създават обекти от класа Person, по-горе, бе използван конструктор по подразбиране, който не приема параметри и на който всички класове имплицитно имат:

```
tom = Person()
```

Въпреки това, може изрично да се дефинира конструктор в класовете, като се използва специален метод, наречен __init__() (две тирета от всяка страна).

Например, нека променим класа Person, като добавим конструктор към него:

```
class Person:
```

```
    # конструктор
    def __init__(self):
        print("Създане на обект Person")

    def say_hello(self):
        print("Hello")


tom = Person()      # Създаване на обект в Person
tom.say_hello()     # Hello
```

И така, тук в кода на класа Person са дефинирани конструктор и метод say_hello(). Като първи параметър конструкторът, подобно на методите, също взема препратка към текущия обект - self. Обикновено конструкторите се използват за дефиниране на действията, които трябва да се предприемат при създаване на обект.

Сега, когато създавате обект:
tom = Person(), ще бъде извикан конструктор от класа Person __init__(), който ще отпечата низа "Създаване на обект Person" в конзолата.

Атрибути на обекта

Атрибутите съхраняват състоянието на обект. Можете да използвате думата self, за да дефинирате и зададете атрибути в рамките на клас. Например, нека дефинираме следния клас Person:

```
class Person:

    def __init__(self, name):
        self.name = name # име
        self.age = 1         # възраст



tom = Person("Tom")

# обръщение към атрибутите и получаване на стойност
print(tom.name)         # Tom
print(tom.age)          # 1
# промяна на стойност
tom.age = 37
print(tom.age)          # 37
```

Сега конструкторът на класа Person приема още един параметър - name. Чрез този параметър името на създаденото лице ще бъде предадено на конструктора. В конструктора се задават два атрибута - name и age (условно името и възрастта на човек):

```
def __init__(self, name):
    self.name = name
    self.age = 1
```

Атрибутът self.name е зададен като стойност на променливата name. Атрибутът age е зададен да е 1.

Ако сме дефинирали конструктор __init__ в клас, вече няма да може да се извика конструктора по подразбиране. Сега трябва да се извика изрично дефинираният конструктор __init__, на който трябва да бъде предадена стойност за параметъра name:

```
tom = Person("Tom")
```

Освен това, чрез името на обекта, може да се получи достъп до атрибутите на обекта - да се получат и променят стойностите им:

```
print(tom.name)          # получаване на стойността на атрибута name
tom.age = 37             # промяна на  стойността на атрибута age
```

По принцип не е необходимо дефиниране на атрибути вътре в класа - Python позволява това да става динамично извън кода:

```
class Person:

    def __init__(self, name):
        self.name = name # име
        self.age = 1          # възраст


tom = Person("Tom")
tom.company = "Microsoft"
print(tom.company)     # Microsoft
```

Тук динамично се задава атрибутът на фирмата, който съхранява работното място на лицето. И след настройка може да се получи и стойността му. В същото време такова определение е изпълнено с грешки. Например, ако се направи опит да се осъществи достъп до атрибут, преди да е дефиниран, програмата ще генерира грешка:

```
tom = Person("Tom")
print(tom.company)  # ! Error - AttributeError: Person object has no attribute company
```

За достъп до атрибути на обект вътре в клас, думата self също се използва в методите му:

```
class Person:

    def __init__(self, name):
        self.name = name    # име
        self.age = 1        # възраст

    def display_info(self):
        print(f"Name: {self.name}  Age: {self.age}")
tom = Person("Tom")
tom.display_info()        # Name: Tom  Age: 1
```

Това дефинира метода display_info(), който отпечатва информация в конзолата. За достъп до атрибутите на обекта в метода се използва думата self: self.name и self.age

Създаване на обекти

```
class Person:

    def __init__(self, name):
        self.name = name       # име
        self.age = 1           # възраст

    def display_info(self):
        print(f"Name: {self.name}  Age: {self.age}")

tom = Person("Tom")
tom.age = 37
tom.display_info()          # Name: Tom  Age: 37

bob = Person("Bob")
bob.age = 41
bob.display_info()          # Name: Bob  Age: 41
```

Тук се създават два обекта от класа Person: tom и bob. Те съвпадат с дефиницията на класа Person, имат същия набор от атрибути и методи, но състоянието им е различно. Когато се изпълнява програма, Python динамично ще определи self - той представлява обекта, върху който е извикан методът.

Например:

tom.display_info()   # Name: Tom Age: 37

Това ще бъде обектът Tom. И когато се извика:

bob.display_info()

Това ще бъде обектът Bob. В резултат на това ще се получи следния конзолен изход:

Name: Tom Age: 37
Name: Bob Age: 41

Капсулиране, атрибути и свойства

По подразбиране атрибутите в класовете са публични, което означава, че от всяко място в програмата може да се получи атрибут на обект, който може да бъде променен.

Например:

```
class Person:
    def __init__(self, name):
        self.name = name            # установяване на име
        self.age = 1                # установяване на възраст

    def display_info(self):
        print(f"Име: {self.name}\tВъзраст: {self.age}")

tom = Person("Tom")
tom.name = “Човекът-паяк"     # изменение на атрибута name
tom.age = -129                # изменение на атрибута age
tom.display_info()            # Име: Човекът-паяк     Възраст: -129
```



```
1  class Person:
2      def __init__(self, name):
3          self.name = name            # установяване на име
4          self.age = 1                # установяване на възраст
5
6      def display_info(self):
7          print(f"Име: {self.name}\tВъзраст: {self.age}")
8
9  tom = Person("Tom")
10 tom.name = “Човекът-паяк"           # изменение на атрибута name
11 tom.age = -129                      # изменение на атрибута age
12 tom.display_info()                  # Име: Човекът-паяк     Възраст: -129
```

В този случай може да се присвои неправилна стойност на възрастта или името на човек, например да посочим отрицателна възраст. Такова поведение е нежелателно, така че възниква въпросът за контролиране на достъпа до атрибутите на обекта. Концепцията за капсулиране е тясно свързана с този проблем.

Капсулирането е и основната концепция в ООП. Той предотвратява директния достъп до атрибутите на обекта от извикващият код. По отношение на капсулирането директно в езика за програмиране Python, може да скриете атрибутите на класа, като ги направите частни и ограничите достъпа до тях чрез специални методи, които също се наричат **свойства**.

Нека се промени дефинирания клас от предходния слайд, като се дефинират свойства в него:

Тема 4_2. Обектно-ориентирано програмиране. Класове и обекти. Капсулиране, атрибути и свойства. Наследяване. Предефиниране на функционалността на базовия клас. Клас атрибути и статични методи. Низово представяне на обект

```
class Person:
    def __init__(self, name):
        self.__name = name          # установяване на име
        self.__age = 1              # установяване на възраст

    def set_age(self, age):
        if 1 < age < 110:
            self.__age = age
        else:
            print("Недопустима възраст")
    def get_age(self):
        return self.__age
    def get_name(self):
        return self.__name
    def display_info(self):
        print(f"Име: {self.__name}\tВъзраст: {self.__age}")
tom = Person("Tom")
tom.display_info()              # Име: Tom  Възраст: 1
tom.set_age(-3486)              # Недопустима възраст
tom.set_age(25)
tom.display_info()              # Име: Tom  Възраст: 25
```


main.py Run Shell

```
class Person:
    def __init__(self, name):
        self.__name = name          # установяване на име
        self.__age = 1              # установяване на възраст

    def set_age(self, age):
        if 1 < age < 110:
            self.__age = age
        else:
            print("Недопустима възраст")
    def get_age(self):
        return self.__age
    def get_name(self):
        return self.__name
    def display_info(self):
        print(f"Име: {self.__name}\tВъзраст: {self.__age}")
tom = Person("Tom")
tom.display_info()              # Име: Tom  Възраст: 1
tom.set_age(-3486)              # Недопустима възраст
tom.set_age(25)
tom.display_info()              # Име: Tom  Възраст: 25
```

```
Име: Tom	Възраст: 1
Недопустима възраст
Име: Tom	Възраст: 25
>
```

За да създадете частен атрибут, в началото на името му се поставя двойно тире: self.__name. Достъп до такъв атрибут има само от същия клас. Извън този клас няма достъп. Например, ако целим да присвоим стойност на този атрибут няма да се получи:

```
tom.__age = 43
```

Защото в този случай нов атрибут __age просто се дефинира динамично, но няма нищо общо със self.__age.

И опитът да се получи стойността му ще доведе до грешка по време на изпълнение (ако променливата __age не е била предварително дефинирана):

```
print(tom.__age)
```

Въпреки това може да се наложи да се зададе възрастта на потребителя отвън. Използвайки self свойство, може да се получи (да се върне) стойността на атрибут:

```
def get_age(self):
    return self.__age
```

Този метод също често се нарича getter или Accessor.

Друго свойство определено за промяна на възрастта е:

```
def set_age(self, age):
    if 1 < age < 110:
        self.__age = age
    else:
        print("Недопустима възраст")
```

Този метод се нарича още setter или мутатор. Тук вече според условията, може да се реши дали да се нулира възрастта. Не е необходимо създаване на такава двойка свойства за всеки частен атрибут. Така че в примера по-горе може да се зададе името на човека само от конструктора. И за получаване е дефиниран методът get_name.

Пояснения към свойствата

По-горе беше разгледано как да се създадат свойства. Но Python има и друг, по-елегантен начин за дефиниране на свойства. Този метод включва използване на пояснения, които се предшестват от символа @. За да се създаде свойство **getter**, анотацията **@property** се поставя над свойството. За да се създаде свойство за настройка, анотацията **getter_property_name.setter** е зададена над свойството.

Нека се пренапише класа Person, като се използват анотации:

```
class Person:
    def __init__(self, name):
        self.__name = name  # установяване на име
        self.__age = 1      # установяване на възраст

   @property
    def age(self):
        return self.__age

    @age.setter
    def age(self, age):
        if 1 < age < 110:
            self.__age = age
        else:
            print("Недопустима възраст")

 @property
    def name(self):
        return self.__name

    def display_info(self):
        print(f"Име: {self.__name}\tВъзраст:
{self.__age}")


tom = Person("Tom")

tom.display_info()   # Име: Tom  Възраст: 1
tom.age = -3486      # Недопустима възраст
print(tom.age)       # 1
tom.age = 36
tom.display_info()   # Име: Tom  Възраст: 36
```

```
tom = Person("Tom")

tom.display_info()   # Име: Tom  Възраст: 1
tom.age = -3486      # Недопустима възраст
print(tom.age)       # 1
tom.age = 36
tom.display_info()   # Име: Tom  Възраст: 36
```

Първо, трябва да се има предвид, че свойството на setter е дефинирано след свойството getter. Второ, и setter, и getter се наричат едно и също - възраст. Тъй като getter се нарича възраст, анотацията се задава над setter @age.setter. След това се отнасяме както към getter, така и към setter чрез израза tom.age.

Тема 4_2. Обектно-ориентирано програмиране. Класове и обекти. Капсулиране, атрибути и свойства. Наследяване. Предефиниране на функционалността на базовия клас. Клас атрибути и статични методи. Низово представяне на обект

main.py

Run

```
class Person:
    def __init__(self, name):
        self.__name = name  # установяване на име
        self.__age = 1      # установяване на възраст
    @property
    def age(self):
        return self.__age
    @age.setter
    def age(self, age):
        if 1 < age < 110:
            self.__age = age
        else:
            print("Недопустима възраст")
    @property
    def name(self):
        return self.__name
     def display_info(self):
        print(f"Име: {self.__name}\tВъзраст: {self.__age}")
tom = Person("Tom")
tom.display_info()   # Име: Tom  Възраст: 1
tom.age = -3486     # Недопустима възраст
print(tom.age)        # 1
tom.age = 36
tom.display_info()   # Име: Tom  Възраст: 36
```

Shell

```
Име: Tom     Възраст: 1
Недопустима възраст
1
Име: Tom     Възраст: 36
>
```

Наследство

Наследяването позволява да се създаде нов клас на базата на съществуващ клас. Заедно с капсулирането, наследяването е един от крайъгълните камъни на ООП. Ключовите понятия за наследяване са подклас и суперклас. Подкласът наследява всички публични атрибути и методи от суперкласа. Суперкласът се нарича още базов (базов клас) или родител (родителски клас), а подкласът - производен (производен клас) или дъщерен (детски клас). Синтаксисът за наследяване на клас е както следва:

```
class подклас (суперклас):
    методи_на_подкласа
```

Например, класът Person:

```
class Person:

    def __init__(self, name):
        self.__name = name          # име
    @property
    def name(self):
        return self.__name

    def display_info(self):
        print(f"Name: {self.__name} ")
```

Да се предположи, че има нужда от създаване на клас Employee, който описва работата на служител в предприятие. Може да се създаде нов клас от нулата, какъвто е клас Employee:

```
class Employee:

    def __init__(self, name):
        self.__name = name          # име на работника

    @property
    def name(self):
        return self.__name

    def display_info(self):
        print(f"Name: {self.__name} ")

    def work(self):
        print(f"{self.name} works")
```

Въпреки това, класът Employee може да има същите атрибути и методи като класа Person, тъй като служителят е човек. Така че в класът Employee е добавен само методът works, останалата част от кода повтаря функционалността на класа Person. Но за да не се дублира функционалността на един клас в друг, в този случай е по-добре да се използва наследяване. Ако се наследи класа Employee от класа Person реализацията ще е:

```
class Person:

    def __init__(self, name):
        self.__name = name        # име

    @property
    def name(self):
        return self.__name

    def display_info(self):
        print(f"Name: {self.__name} ")

class Employee(Person):
    def work(self):
        print(f"{self.name} works")

tom = Employee("Tom")
print(tom.name)             # Tom
tom.display_info()          # Name: Tom
tom.work()                  # Tom works
```

Класът Employee напълно поема функционалността на класа Person, като добавя само work(). Съответно, когато се създава обект Employee, може да се използва конструктора, наследен от Person:

```
tom = Employee("Tom")
```

Освен това може да се получи достъп до наследени атрибути/свойства и методи:

```
print(tom.name)         # Tom
tom.display_info()      # Name: Tom
```

Трябва да се има предвид, че Employee НЯМА частни атрибути от тип __name. Например, НЕ може да получи достъп до частния атрибут в работния метод self.__name:

```
def work(self):
    print(f"{self.__name} works")        # ! Error
```

```
class Employee(Person):

    def work(self):
        print(f"{self.name} works")


tom = Employee("Tom")
print(tom.name)              # Tom
tom.display_info()           # Name: Tom
tom.work()                   # Tom works
```

Тема 4_2. Обектно-ориентирано програмиране. Класове и обекти. Капсулиране, атрибути и свойства. Наследяване. Предефиниране на функционалността на базовия клас. Клас атрибути и статични методи. Низово представяне на обект

main.py

```python
class Person:
    def __init__(self, name):
        self.__name = name          # име
    @property
    def name(self):
        return self.__name
    def display_info(self):
        print(f"Name: {self.__name} ")
class Employee:
    def __init__(self, name):
        self.__name = name          # име на работника
    @property
    def name(self):
        return self.__name
    def display_info(self):
        print(f"Name: {self.__name} ")
    def work(self):
        print(f"{self.name} works")
class Employee(Person):
     def work(self):
        print(f"{self.name} works")
tom = Employee("Tom")
print(tom.name)                 # Tom
tom.display_info()              # Name: Tom
tom.work()                      # Tom works
```

Shell

```
Tom
Name: Tom
Tom works
>
```

Множествено наследяване

Една от отличителните черти на езика Python е поддръжката за множествено наследяване, тоест един клас може да бъде наследен от няколко класа:

```
#  клас  Employee
class Employee:
    def work(self):
        print("Employee works")

#  клас  Student
class Student:
    def study(self):
        print("Student studies")

 #WorkingStudent наследява класовет
class WorkingStudent(Employee, Student):
    pass

# клас  Working Student
tom = WorkingStudent()
tom.work()              # Employee works
tom.study()             # Student studies
```

main.py   Run   Shell

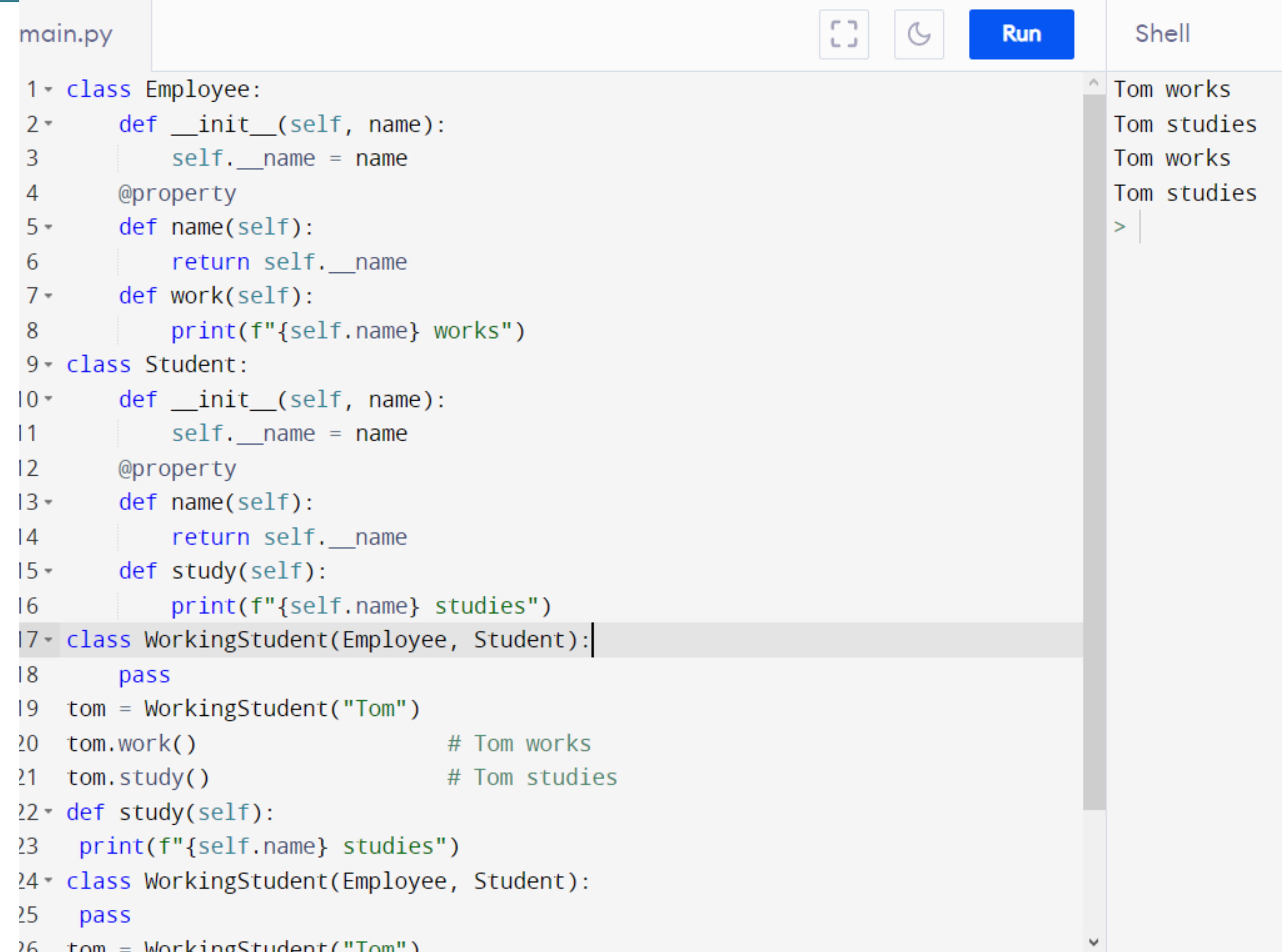
```
1  #клас  Employee
2  class Employee:
3      def work(self):
4          print("Employee works")
5  #клас  Student
6  class Student:
7      def study(self):
8          print("Student studies")
9  #WorkingStudent наследява класовете Employee и Student
10 class WorkingStudent(Employee, Student):
11     pass
12 # клас  Working Student
13 tom = WorkingStudent()
14 tom.work()                  # Employee works
15 tom.study()                 # Student studies
```

```
Employee works
Student studies
>
```

Той дефинира клас Employee, който представлява служител на фирмата, и студентски клас, който представлява Student. Класът WorkingStudent, който представлява работещ студент, не дефинира никаква функционалност, така че дефинира изявление за преминаване. Класът WorkingStudent просто наследява функционалност от двата класа Employee и Student. Съответно могат да бъдат извикани методите на двата класа върху обект от този клас. В същото време наследените класове могат да бъдат по-сложни по функционалност, например:

```
class Employee:
    def __init__(self, name):
        self.__name = name
    @property
    def name(self):
        return self.__name
    def work(self):
        print(f"{self.name} works")
class Student:
    def __init__(self, name):
        self.__name = name
    @property
    def name(self):
        return self.__name
    def study(self):
        print(f"{self.name} studies")
class WorkingStudent(Employee, Student):
    pass
tom = WorkingStudent("Tom")
tom.work()                # Tom works
tom.study()               # Tom studies
def study(self):
 print(f"{self.name} studies")
class WorkingStudent(Employee, Student):
 pass
tom = WorkingStudent("Tom")
tom.work()        # Tom works
tom.study()       # Tom studies
```

Тема 4_2. Обектно-ориентирано програмиране. Класове и обекти. Капсулиране, атрибути и свойства. Наследяване. Предефиниране на функционалността на базовия клас. Клас атрибути и статични методи. Низово представяне на обект




```python
class Employee:
    def __init__(self, name):
        self.__name = name
    @property
    def name(self):
        return self.__name
    def work(self):
        print(f"{self.name} works")
class Student:
    def __init__(self, name):
        self.__name = name
    @property
    def name(self):
        return self.__name
    def study(self):
        print(f"{self.name} studies")
class WorkingStudent(Employee, Student):
    pass
tom = WorkingStudent("Tom")
tom.work()                  # Tom works
tom.study()                 # Tom studies
def study(self):
 print(f"{self.name} studies")
class WorkingStudent(Employee, Student):
 pass
tom = WorkingStudent("Tom")
```

Предефиниране на функционалността на базовия клас

Класът Employee може да възприеме функционалността на класа Person:

```
class Person:

    def __init__(self, name):
        self.__name = name              # име

    @property
    def name(self):
        return self.__name

    def display_info(self):
        print(f"Name: {self.__name} ")


class Employee(Person):

    def work(self):
        print(f"{self.name} works")
```

Но какво ще стане, ако трябва да се промени нещо от тази функционалност? Например да се добави нов атрибут към служител чрез конструктора, който ще съхранява компанията, в която работи, или да се промени реализацията на метода display_info. Python позволява да замени функционалността на базов клас. Например, нека се променят класовете по следния начин:

```
class Person:

    def __init__(self, name):
        self.__name = name      # име

    @property
    def name(self):
        return self.__name

    def display_info(self):
        print(f"Name: {self.__name}")
```

```
class Employee(Person):

    def __init__(self, name, company):
        super().__init__(name)
        self.company = company

    def display_info(self):
        super().display_info()
        print(f"Company: {self.company}")

    def work(self):
        print(f"{self.name} works")


tom = Employee("Tom", "Microsoft")
tom.display_info()              # Name: Tom
                                # Company: Microsoft
```

main.py | Run | Shell

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

```
Name: Tom
Company: Microsoft
>
```

Тук се добавя нов атрибут към класа на служителя - self.company, който се съхранява от компанията на служителя. Съответно методът __init__() приема три параметъра: вторият за задаване на името и третия за настройка на компанията. Но ако конструкторът е дефиниран в базовия клас с помощта на метода __init__ и е необходимо да се промени логиката на конструктора в производния клас, тогава в конструктора на производния клас трябва да се извика конструктора на базовия клас. Т.е. в конструктора Employee трябва да се извика конструктора на клас Person.

Изразът super() се използва за препращане към базовия клас. И така, в конструктора Employee се извършва извикването:
super().__init__(name)

Този израз представлява извикване към конструктора на класа Person, на който се предава името на работника. И това е логично. В крайна сметка името на работника е зададено в конструктора на класа Person. В самия конструктор Employee се задава само свойството на компанията. Освен това в класа Employee, методът display_info() е отменен, като към него се добавя продукцията на компанията на служителя. Методът може да се дефинира по следния начин:

```
def display_info(self):
    print(f"Name: {self.name}")
    print(f"Company: {self.company}")
```

Тогава изходният ред за име ще повтори кода от класа Person. Ако тази част от кода съвпада с метода от класа Person, тогава няма смисъл да се повтаря, така че отново, използвайки израза super(), става обръщане към реализацията на метода display_info в класа Person:

```
def display_info(self):
    super().display_info()        # обръщение към метод display_info в клас Person
    print(f"Company: {self.company}")
```

След това може да се извика конструктора Employee, за да се създаде обект от този клас и да се извика метода display_info:

```
tom = Employee("Tom", "Microsoft")
tom.display_info()
```

Конзолен изход на програмата:

```
Name: Tom
Company: Microsoft
```

Проверка на типа на обекта

При работа с обекти може да се наложи извършване на определени операции в зависимост от вида им. А с вградената функция isinstance() може да провери типа на обекта. Тази функция приема два параметъра:
isinstance(object, type)

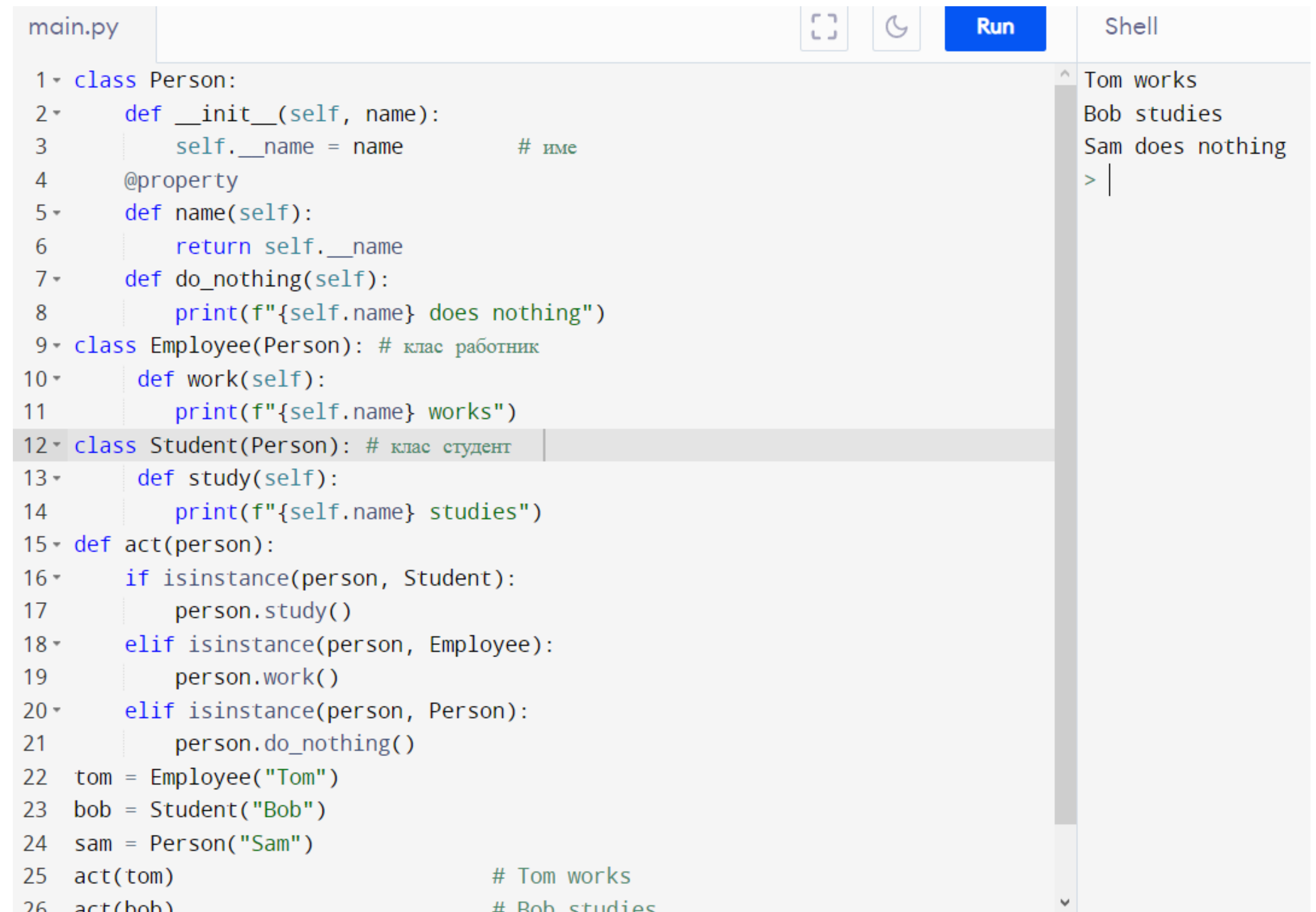
Първият параметър представлява обекта, а вторият параметър представлява типа, за който ще се тества. Ако обектът представлява посочения тип, тогава функцията връща True. Например, ако се разгледа следната йерархия на класове лице-служител/студент:

Тема 4_2. Обектно-ориентирано програмиране. Класове и обекти. Капсулиране, атрибути и свойства. Наследяване. Предефиниране на функционалността на базовия клас. Клас атрибути и статични методи. Низово представяне на обект

```
class Person:
    def __init__(self, name):
        self.__name = name         # име

    @property
    def name(self):
        return self.__name

    def do_nothing(self):
        print(f"{self.name} does nothing")

#  клас работник
class Employee(Person):
    def work(self):
        print(f"{self.name} works")

#  клас студент
class Student(Person):
    def study(self):
        print(f"{self.name} studies")
```

```
def act(person):
    if isinstance(person, Student):
        person.study()
    elif isinstance(person, Employee):
        person.work()
    elif isinstance(person, Person):
        person.do_nothing()


tom = Employee("Tom")
bob = Student("Bob")
sam = Person("Sam")

act(tom)                    # Tom works
act(bob)                    # Bob studies
act(sam)                    # Sam does nothing
```

Тема 4_2. Обектно-ориентирано програмиране. Класове и обекти. Капсулиране, атрибути и свойства. Наследяване. Предефиниране на функционалността на базовия клас. Клас атрибути и статични методи. Низово представяне на обект



Тук класът Employee дефинира метода work(), а класът Student дефинира метода на обучение. Той също така дефинира функция act, която проверява с функцията isinstance дали параметърът person представлява определен тип и в зависимост от резултатите от проверката извиква конкретен метод на обекта.

Атрибути на класа и статични методи

атрибути на класа

В допълнение към атрибутите на обекти в клас, могат да се дефинират атрибути на клас. Такива атрибути се дефинират като променливи на ниво клас. Например:

```
class Person:
    type = "Person"
    description = "Describes a person"

print(Person.type)              # Person
print(Person.description)       # Describes a person

Person.type = "Class Person"
print(Person.type)              # Class Person
```

Тук в класа Person са дефинирани два атрибута: type, който съхранява името на класа, и description, който съхранява описанието на класа. За да се позовем на атрибутите на клас, може да се използва името на класа, например: Person.type, и, подобно на атрибутите на обект, могат да се получат и променят техните стойности. **Подобни атрибути са общи за всички обекти от класа:**

```
class Person:
    type = "Person"
    def __init__(self, name):
        self.name = name

tom = Person("Tom")
bob = Person("Bob")
print(tom.type)          # Person
print(bob.type)          # Person

# изменяме атрибута на класа
Person.type = "Class Person"
print(tom.type)          # Class Person
print(bob.type)          # Class Person
```

main.py | Run | Shell

```
1 class Person:
2       type = "Person"
3       description = "Describes a person"
4 print(Person.type)                  # Person
5 print(Person.description)           # Describes a person
6 Person.type = "Class Person"
7 print(Person.type)                  # Class Person
```

```
Person
Describes a person
Class Person
>
```

Атрибутите на класа могат да се използват за ситуации, в които трябва да се дефинират някои общи данни за всички обекти. Например:

```
class Person:
    default_name = "Undefined"

    def __init__(self, name):
        if name:
            self.name = name
        else:
            self.name = Person.default_name
tom = Person("Tom")
bob = Person("")
print(tom.name)             # Tom
print(bob.name)             # Undefined
```

```
main.py                                          Run   Shell
1  class Person:                                       Tom
2      default_name = "Undefined"                      Undefined
3      def __init__(self, name):                       >
4          if name:
5              self.name = name
6          else:
7              self.name = Person.default_name
8  tom = Person("Tom")
9  bob = Person("")
10 print(tom.name)             # Tom
11 print(bob.name)             # Undefined
```

В този случай атрибутът default_name съхранява името по подразбиране. И ако празен низ за името се предаде на конструктора, тогава стойността на атрибута на класа default_name се предава на атрибута name. За обръщение към атрибут на клас в метод, може да се използва името на класа:

```
self.name = Person.default_name
```

атрибут на класа

Възможно е атрибут на клас и атрибут на обект да имат едно и също име. Ако стойността не е зададена за атрибут на обект в кода, тогава стойността на атрибута на класа може да се използва за него:

```
class Person:
    name = "Undefined"

    def print_name(self):
        print(self.name)

tom = Person()
bob = Person()
tom.print_name()                # Undefined
bob.print_name()                # Undefined

bob.name = "Bob"
bob.print_name()                # Bob
tom.print_name()                # Undefined
```

Тук методът print_name използва атрибута name на обекта, но никъде в кода не е зададен този атрибут. Но на ниво клас атрибутът name е зададен. Следователно, първия път, когато методът print_name бъде извикан, той ще използва стойността на атрибута на класа:

tom = Person()
bob = Person()
tom.print_name()   # Undefined
bob.print_name()   # Undefined

main.py

```
class Person:
    name = "Undefined"
    def print_name(self):
        print(self.name)
tom = Person()
bob = Person()
tom.print_name()                # Undefined
bob.print_name()                # Undefined
bob.name = "Bob"
bob.print_name()                # Bob
tom.print_name()                # Undefined
Person.name = "Some Person"     # промяна на стойността на атрибута на класа
bob.name = "Bob"                          # установяване на атрибута на обекта
bob.print_name()    # Bob
tom.print_name()    # Some Person
```

Shell

```
Undefined
Undefined
Bob
Undefined
Bob
Some Person
>
```

След това обаче може да се промени атрибута set на обекта:

bob.name = "Bob"
bob.print_name()   # Bob
tom.print_name()   # Undefined

Освен това вторият обект - tom ще продължи да използва атрибута клас. И ако се промени атрибута на класа, стойността tom.name също ще се промени:

tom = Person()
bob = Person()
tom.print_name()   # Undefined
bob.print_name()   # Undefined

```
Person.name = "Some Person"     # промяна на стойността на атрибута на класа
bob.name = "Bob"                # установяване на атрибута на обекта
bob.print_name()                # Bob
tom.print_name()                # Some Person
```

Статични методи

В допълнение към обикновените методи, един клас може да дефинира статични методи. Такива методи се предшестват от анотацията @staticmethod и се отнасят до класа като цяло. Статичните методи обикновено дефинират поведение, което е независимо от конкретен обект:

```
class Person:
    __type = "Person"

    @staticmethod
    def print_type():
        print(Person.__type)
Person.print_type()     # Person – достъп до статичен метод чрез името на класа

tom = Person()
tom.print_type()            # Person – обръщение към статичен метод чрез името на обекта
```

Тема 4_2. Обектно-ориентирано програмиране. Класове и обекти. Капсулиране, атрибути и свойства. Наследяване. Предефиниране на функционалността на базовия клас. Клас атрибути и статични методи. Низово представяне на обект

```
class Person:
    def __init__(self, name, age):
        self.name = name            # установяване на име
        self.age = age              # установяване на възраст

    def display_info(self):
        print(f"Name: {self.name}  Age: {self.age}")

tom = Person("Tom", 23)
print(tom)
```

Когато се стартира, програмата се показва следното:

```
<__main__.Person обект на 0x10a63dc00>
```

```
main.py                                                        Run
1  class Person:
2      __type = "Person"
3      @staticmethod
4      def print_type():
5          print(Person.__type)
6  Person.print_type()     #Person - достъп до статичен метод чрез името на класа
7  tom = Person()
8  tom.print_type()        #Person - обръщение към статичен метод чрез името на
       обекта
9  class Person:
10     def __init__(self, name, age):
11         self.name = name #установяване на име
12         self.age = age   #установяване на възраст
13     def display_info(self):
14         print(f"Name: {self.name}  Age: {self.age}")
15 tom = Person("Tom", 23)
16 print(tom)              #Error
```

```
Shell
Person
Person
<__main__.Person object at 0x7f8ba281b400>
>
```

Това не е много информативно съобщение за обекта. Разбира се, може да се заобиколи това, като се дефинира допълнителен метод в класа Person, който показва данните на обекта - в примера по-горе това е методът display_info. Но има и друг изход – дефиниране на метода __str__() в класа Person (две долни черти от всяка страна):

```
class Person:
    def __init__(self, name, age):
        self.name = name      # установяване на име
        self.age = age        # установяване на възраст

    def display_info(self):
        print(self)
        # print(self.__str__())     # или по този начин

    def __str__(self):
        return f"Name: {self.name}  Age: {self.age}"

tom = Person("Tom", 23)
print(tom)                   # Name: Tom  Age: 23
tom.display_info()           # Name: Tom  Age: 23
```

Методът __str__трябва да върне низ. И в този случай се връща основната информация за лицето. Ако трябва да се използва тази информация в други методи на класа, тогава може да се използва израза self.__str__()

Изходът на конзолата ще бъде различен:

Name: Tom Age: 23

Name: Tom Age: 23

main.py

```
class Person:
    def __init__(self, name, age):
        self.name = name          # установяване на име
        self.age = age            # установяване на възраст

    def display_info(self):
        print(self)
        # print(self.__str__())   # или по този начин

    def __str__(self):
        return f"Name: {self.name}  Age: {self.age}"

tom = Person("Tom", 23)
print(tom)                        # Name: Tom  Age: 23
tom.display_info()                # Name: Tom  Age: 23
```

Shell

```
Name: Tom  Age:
Name: Tom  Age:
>
```

Използвана литература:

[1]. Joakim Sundnes, Introduction to Scientific Programming with Python, Simula Springer Briefs on Computing, 2020
[2]. Brian Heinold, A Practical Introduction to Python Programming, Department of Mathematics and Computer Science Mount St. Mary’s University, 2012
[3]. David Mertz, Functional Programming in Python, O’Reilly Media, 2015
[4]. Steven Lott, Functional Python Programming, Published by Packt Publishing Ltd., 2015
[5]. Mark Lutz, Learning Python, Fourth Edition, O’Reilly Media, 2009

Списъци, кортежи и речници

Списък

За да работи с набори от данни, Python предоставя вградени типове като списъци, кортежи и речници. Списък е тип данни, който съхранява набор или последователност от елементи. Много езици за програмиране имат подобна структура от данни, наречена масив.

Създаване на списък

За създаване на списък се използват квадратни скоби [], вътре в които се изброяват елементите на списъка, разделени със запетаи. Например, нека да се дефинира списък с числа:

```
numbers = [1, 2, 3, 4, 5]
```

По същия начин може да се дефинира списък с други типове данни, като списък от низове:

```
people = ["Tom", "Sam", "Bob"]
```

Може също да се използва функцията за конструктор list(), за да се създаде списък:

```
numbers1 = []
numbers2 = list()
```

И двете дефиниции на списъци са сходни - те създават празен списък. Списъкът не трябва да съдържа само обекти от същия тип. Може да се поставят низове, числа, обекти и от други типове данни в един и същ списък по едно и също време:

```
objects = [1, 2.6, "Hello", True]
```

За да се проверят елементите на списък, може да се използва стандартна функция за печат, която отпечатва съдържанието на списъка в четима форма:

```
numbers = [1, 2, 3, 4, 5]
people = ["Tom", "Sam", "Bob"]
print(numbers)          # [1, 2, 3, 4, 5]
print(people)           # ["Tom", "Sam", "Bob"]
```

Конструктор на списък

Може да приеме набор от стойности, възоснова на които се създава самият списък:

```
numbers1 = [1, 2, 3, 4, 5]
numbers2 = list(numbers1)
print(numbers2)          # [1, 2, 3, 4, 5]

letters = list("Hello")
print(letters)           # ['H', 'e', 'l', 'l', 'o']
```

Ако трябва да се създаде списък, в който една и съща стойност се повтаря няколко пъти, се използва символа звездичка *, като да се приложи операцията за умножение към вече съществуващ списък:

```
numbers = [5] * 6                # 6 пъти се повтаря числото 5
print(numbers)                   # [5, 5, 5, 5, 5, 5]

people = ["Tom"] * 3             # 3 пъти се повтаря "Tom"
print(people)                    # ["Tom", "Tom", "Tom"]

students = ["Bob", "Sam"] * 2    # 2 пъти се повтаря "Bob", "Sam"
print(students)                  # ["Bob", "Sam", "Bob", "Sam"]
```

Достъп до елементи от списъка

За достъп до елементите на списък трябва да се използват индекси, които представляват номера на елемента в списъка. Индексите започват от нула. Т.е. първият елемент ще има индекс 0, вторият елемент ще има индекс 1 и т.н. За достъп до елементи от края може да се използват отрицателни индекси, започващи от -1. Т.е. последният елемент ще има индекс -1, предпоследният ще има индекс -2 и т.н.

```
people = ["Tom", "Sam", "Bob"]
# получаване на елементи от началото на списъка
print(people[0])   # Tom
print(people[1])   # Sam
print(people[2])   # Bob

# получаване на елементи от края на списъка
print(people[-2])   # Sam
print(people[-1])   # Bob
print(people[-3])   # Tom
```

За да се промени елемент от списък, достатъчно е да му се зададе нова стойност:

```
people = ["Tom", "Sam", "Bob"]
people[1] = "Mike"  # промяна на втория елемент
print(people[1])       # Mike
print(people)           # ["Tom", "Mike", "Bob"]
```

Разлагане на списък

Python позволява да се декомпозира списък на отделни елементи:

```
people = ["Tom", "Bob", "Sam"]
tom, bob, sam = people
print(tom)      # Tom
print(bob)      # Bob
print(sam)      # Sam
```

В този случай променливите tom, bob и sam са последователно присвоени елементи от списъка с хора. Обаче, трябва да се има предвид, че броят на променливите трябва да е равен на броя на елементите от присвоения списък.

Итерация върху елементи

Може да се използва, както цикъл for, така и цикъл while, за да се преглеждат елементите.

Итерация с for цикъл:

```
people = ["Tom", "Sam", "Bob"]
for person in people:
    print(person)
```

Тук списъкът с хора ще бъде повторен и всеки елемент ще бъде поставен в променливата person.

Итерацията може да се направи и с цикъл while:

```
people = ["Tom", "Sam", "Bob"]
i = 0
while i < len(people):
    print(people[i])    # използване на индекс, за получаване на елемент
    i += 1
```

В цикъла се използва функцията len(), чрез, която се получава дължината на списъка. С помощта на брояча i се отпечатва елемент по елемент, докато стойността на брояча е равна на дължината на списъка.

Сравнение на списъци

Два списъка се считат за равни, ако съдържат един и същ набор от елементи:

```
numbers1 = [1, 2, 3, 4, 5]
numbers2 = list([1, 2, 3, 4, 5])
if numbers1 == numbers2:
    print("numbers1 equal to numbers2")
else:
    print("numbers1 is not equal to numbers2")
```

В този случай и двата списъка ще бъдат равни.

Изброяване на методи и функции

Списъците имат редица методи за манипулиране на елементи. Някои от тях:

- **append(item):** добавя елемент в края на списъка;
- **insert(index, item):** добавя елемент към списъка с индекс;
- **extend(items):** добавя набор от елементи в края на списъка;
- **remove(item):** премахва елемента item. Премахва се само първата поява на елемента. Ако елементът не бъде намерен, се хвърля изключение ValueError;
- **clear():** премахва всички елементи от списъка;
- **index(item):** Връща индекса на елемента. Ако елементът не бъде намерен, хвърля изключение ValueError;
- **pop([index]):** премахва и връща елемента в index. Ако не се предава индекс, тогава просто премахва последния елемент;
- **count(item):** връща броя на появата на елемент в списъка;
- **sort([key]):** сортира елементите. Сортира във възходящ ред по подразбиране. Но с помощта на ключовия параметър може да се предаде функция за сортиране;
- **reverse():** обръща всички елементи в списък;
- **copy():** копира списъка.

В допълнение, Python предоставя редица вградени функции за работа със списъци:

- len(list): връща дължината на списъка;
- sorted(list, [key]): връща сортиран списък;
- min(list): връща най-малкия елемент от списъка;
- max(list): връща най-големия елемент от списъка.

Добавяне и премахване на елементи

Методите се използват за добавяне на елемент: append(), extend и insert, а за премахване на елемент: remove(), pop(), clear().

Използване на методи:

people = ["Tom", "Bob"]
 # добавяне в края на списъка
people.append("Alice") # ["Tom", "Bob", "Alice"]
 # добавяне на втора позиция
people.insert(1, "Bill") # ["Tom", "Bill", "Bob", "Alice"]
 # добавяне на набор от елементи ["Mike", "Sam"]
people.extend(["Mike", "Sam"])   # ["Tom", "Bill", "Bob", "Alice", "Mike", "Sam"]
 # вземане на индекса на елемента
index_of_tom = people.index("Tom")

 # премахване на елемент по индекс
removed_item = people.pop(index_of_tom) # ["Bill", "Bob", "Alice", "Mike", "Sam"]
 # премахване на последния елемент
last_item = people.pop() # ["Bill", "Bob", "Alice", "Mike"]
 # премахване на елемент "Alice"
people.remove("Alice") # ["Bill", "Bob", "Mike"]
print(people) # ["Bill", "Bob", "Mike"]
 # премахване на всички елементи
people.clear()
print(people) # []

main.py

```
1  people = ["Tom", "Bob"]
2   # добавяне в края на списъка
3  people.append("Alice")  # ["Tom", "Bob", "Alice"]
4  # добавяне на втора позиция
5  people.insert(1, "Bill")  # ["Tom", "Bill", "Bob", "Alice"]
6  # добавяне на набор от елементи ["Mike", "Sam"]
7  people.extend(["Mike", "Sam"])      # ["Tom", "Bill", "Bob", "Alice", "Mike",
       "Sam"]
8  # вземане на индекса на елемента
9  index_of_tom = people.index("Tom")
10 # премахване на елемент по индекс
11 removed_item = people.pop(index_of_tom)     # ["Bill", "Bob", "Alice", "Mike",
       "Sam"]
12 # премахване на последния елемент
13 last_item = people.pop()     # ["Bill", "Bob", "Alice", "Mike"]
14 # премахване на елемент "Alice"
15 people.remove("Alice")      # ["Bill", "Bob", "Mike"]
16 print(people)                      # ["Bill", "Bob", "Mike"]
17 # премахване на всички елементи
18 people.clear()
19 print(people)       # []
```

Shell

```
['Bill', 'Bob', 'Mike']
[]
>
```

Проверка дали съществува елемент

Ако конкретен елемент не бъде намерен, тогава методите remove и index предизвикват изключение. За да се избегне тази ситуация, преди да работите с елемент, може да проверите за присъствието му, като използвате ключовата дума in:

people = ["Tom", "Bob", "Alice", "Sam"]
if "Alice" in people:
people.remove("Alice")
print(people)

main.py

```
1 people = ["Tom", "Bob", "Alice", "Sam"]
2 if "Alice" in people:
3     people.remove("Alice")
4 print(people)
```

Shell

```
['Tom', 'Bob', 'Sam']
>
```

Изразът if "Alice" in people връща True, ако елементът "Alice" присъства в списъка с people. Следователно, конструкцията if "Alice" in people може да изпълни следващия блок от инструкции, в зависимост от присъствието на елемента в списъка.

Изтриване с del

Python също така поддържа друг начин за премахване на елементи от списък, като използва изразът del. Елементът или наборът от елементи, които трябва да бъдат премахнати, се предава като параметър на този оператор:

people = ["Tom", "Bob", "Alice", "Sam", "Bill", "Kate", "Mike"]

main.py

```
1 people = ["Tom", "Bob", "Alice", "Sam", "Bill", "Kate", "Mike"]
2 del people[1]        # премахване на втория елемент
3 print(people)        # ["Tom", "Alice", "Sam", "Bill", "Kate", "Mike"]
4 del people[:3]    # премахване на 4-ят елемент, без да се включва (премахване на
      всички 4-ри елемента)
5 print(people)        # ["Bill", "Kate", "Mike"]
6 del people[1:]       # премахване от втория елемент
7 print(people)        # ["Bill"]
```

Shell

```
['Tom', 'Alice', 'Sam', 'Bill', 'Kate', 'Mike']
['Bill', 'Kate', 'Mike']
['Bill']
>
```

Брой на срещанията в списък

Ако трябва да разберете колко пъти даден елемент присъства в списъка, може да използвате метода count():

```
people = ["Tom", "Bob", "Alice", "Tom", "Bill", "Tom"]
people_count = people.count("Tom")
print(people_count)                    # 3
```

Сортиране

Методът sort() се използва за сортиране във възходящ ред:

```
people = ["Tom", "Bob", "Alice", "Sam", "Bill"]
people.sort()
print(people)          # ["Alice", "Bill", "Bob", "Sam", "Tom“]
```

Ако е необходимо да сортирате данните в обратен ред, тогава може да приложите метода след сортиране reverse():

```
people = ["Tom", "Bob", "Alice", "Sam", "Bill"]
people.sort()
people.reverse()
print(people)          # ["Tom", "Sam", "Bob", "Bill", "Alice"]
```

Сортирането всъщност сравнява два обекта и кой от тях е "по-малко" се поставя пред този, който е "по-голям". Понятията "повече" и "по-малко" са доста произволни. И ако за числата всичко е просто - числата са подредени във възходящ ред, то за низове и други обекти ситуацията е по-сложна. По-специално, **низовете се оценяват по първите символи.** Ако първите знаци са равни, се оценяват вторите и т.н. **При което цифров знак се счита за "по-малко" от буквен главен знак, а главен знак се счита за по-малък от символ с малки букви.** По този начин, ако списъкът комбинира низове с главни и малки букви, тогава могат да получат не съвсем правилни резултати, тъй като за нас низът "bob" трябва да идва преди низа "Tom". И за да се промени стандартното поведение на сортиране, може да се предаде sort() функцията, като параметър на метода:

```
people = ["Tom", "bob", "alice", "Sam", "Bill"]

people.sort()                   # стандартна сортировка
print(people)                   # ["Bill", "Sam", "Tom", "alice", "bob"]

people.sort(key=str.lower)      # сортировка без счетоводен регистър
print(people)                   # ["alice", "Bill", "bob", "Sam", "Tom"]
```

В допълнение към метода за сортиране, може да се използва сортираната вградена функция , която има две форми:

- **sorted(list):** сортира списък
- **sorted(list, key):** сортира списъка със списък, като прилага ключовата функция към елементите

```
people = ["Tom", "bob", "alice", "Sam", "Bill"]
sorted_people = sorted(people, key=str.lower)
print(sorted_people)            # ["alice", "Bill", "bob", "Sam", "Tom"]
```

Когато се използва тази функция, трябва да се има предвид, че тази функция не променя сортирания списък, но поставя всички сортирани елементи в нов списък, който се връща като резултат.

Минимални и максимални стойности

Вградените функции на Python min() и max() позволява търсене на минималните и максималните стойности:

```
numbers = [9, 21, 12, 1, 3, 15, 18]
print(min(numbers))            # 1
print(max(numbers))            # 21
```

Копиране на списъци

Когато копирате списъци, трябва да се има предвид, че списъците представляват променлив тип, така че ако и двете променливи сочат към един и същ списък, тогава промяната на една променлива ще повлияе на друга променлива:

```
people1 = ["Tom", "Bob", "Alice"]
people2 = people1
people2.append("Sam")       # добавете елемент към втория списък
                            # people1 и people2 сочат към един и същ списък
print(people1)              # ["Tom", "Bob", "Alice", "Sam"]
print(people2)              # ["Tom", "Bob", "Alice", "Sam"]
```

Това е така нареченото **"плитко копие".** И като правило такова поведение е нежелателно. За да копирате елементите, но в същото време променливите да сочат към различни списъци, трябва да извършите дълбоко копиране. Можете да използвате метода copy() за това:

```
people1 = ["Tom", "Bob", "Alice"]
people2 = people1.copy()    # копирайте елементи от people1 в people2
people2.append("Sam")       # добавете елемент САМО към втория списък
                            # people1 и people2 вече са различни списъци
print(people1)              # ["Tom", "Bob", "Alice"]
print(people2)              # ["Tom", "Bob", "Alice", "Sam"]
```

Копиране на част от списък

Ако е необходимо да се копира не целия списък, а само конкретна част от него, тогава може да се използва специален синтаксис, който може да приеме следните форми:

- **list[:end]:** индексът на елемента, към който трябва да се копира списъкът, се предава през крайния параметър
- **list[start:end]:** началният параметър сочи към индекса на елемента, от който да се копират елементите
- **list[start:end:step]:** параметърът step показва стъпката, през която ще бъдат копирани елементите от списъка. По подразбиране този параметър е 1.

```
people = ["Tom", "Bob", "Alice", "Sam", "Tim", "Bill"]

slice_people1 = people[:3]          # от 0 до 3
print(slice_people1)                # ["Tom", "Bob", "Alice"]

slice_people2 = people[1:3]         # от 1 до 3
print(slice_people2)                # ["Bob", "Alice"]

slice_people3 = people[1:6:2]       # от 1 до 6 през 2 стъпки
print(slice_people3)                # ["Bob", "Sam", "Bill"]
```

Списъци за свързване

Операторът за добавяне (+) се използва за комбиниране на списъци:

```
people1 = ["Tom", "Bob", "Alice"]
people2 = ["Tom", "Sam", "Tim", "Bill"]
people3 = people1 + people2
print(people3)                    # ["Tom", "Bob", "Alice", "Tom", "Sam", "Tim", "Bill"]
```

Списъци със списъци

Списъците, освен стандартни данни като низове, числа, могат да съдържат и други списъци. Такива списъци могат да бъдат свързани с таблици, където вложените списъци действат като редове. Например:

```
people = [
    ["Tom", 29],
    ["Alice", 33],
    ["Bob", 27]
]

print(people[0])                  # ["Tom", 29]
print(people[0][0])               # Tom
print(people[0][1])               # 29
```

За обръщение към елемент от вложен списък, трябва да се използват двойка индекси: people[0][1] - достъп до втория елемент от първия вложен списък. Добавянето, изтриването и промяната на общия списък, както и на вложените списъци, е подобно на начина, по който се прави с обикновени (едномерни) списъци:

```
people = [
    ["Tom", 29],
    ["Alice", 33],
    ["Bob", 27]
]

# създаване на вложен списък
person = list()
person.append("Bill")
person.append(41)

# добавяне на вложен списък
people.append(person)
print(people[-1])         # ["Bill", 41]

# добавяне към вложен списък
people[-1].append("+79876543210")
print(people[-1])         # ["Bill", 41, "+79876543210"]
```

main.py

```
people = [
    ["Tom", 29],
    ["Alice", 33],
    ["Bob", 27]
]
# създаване на вложен списък
person = list()
person.append("Bill")
person.append(41)
# добавяне на вложен списък
people.append(person)
print(people[-1])          # ["Bill", 41]
# добавяне към вложен списък
people[-1].append("+79876543210")
print(people[-1])          # ["Bill", 41, "+79876543210"]
# премахване на последния елемент от вложен списък
people[-1].pop()
print(people[-1])            # ["Bill", 41]
# изтриване на целия последен вложен списък
people.pop(-1)
# промяна на първия елемент
people[0] = ["Sam", 18]
print(people)                # [ ["Sam", 18], ["Alice", 33], ["Bob", 27]]
```

Shell

```
['Bill', 41]
['Bill', 41, '+79876543210']
['Bill', 41]
[['Sam', 18], ['Alice', 33], ['Bob', 27]]
>
```

```
# премахване на последния елемент от вложен списък
people[-1].pop()
print(people[-1])          # ["Bill", 41]

# изтриване на целия последен вложен списък
people.pop(-1)

# промяна на първия елемент
people[0] = ["Sam", 18]
print(people)              # [ ["Sam", 18], ["Alice", 33], ["Bob", 27]]
```

Итерация върху вложени списъци:

```
people = [
    ["Tom", 29],
    ["Alice", 33],
    ["Bob", 27]
]

for person in people:
    for item in person:
        print(item, end=" | ")
print("")
```

main.py

```
1 people = [
2     ["Tom", 29],
3     ["Alice", 33],
4     ["Bob", 27]
5 ]
6 for person in people:
7     for item in person:
8         print(item, end=" | ")
9 print("")
```

Shell

```
Tom | 29 | Alice | 33 | Bob | 27
>
```

Изход в конзолата:

Tom | 29 | Alice | 33 | Bob | 27 |

Кортежи

Кортежът представлява **последователност от елементи, подобно на списък, с изключение на това, че кортежът е неизменяем тип.** **Следователно не може да се добавят или премахват елементи в кортеж, така, че да се променя.** За създаване на кортеж се използват скоби, в които се поставят стойностите му, разделени със запетаи:

```
tom = ("Tom", 23)
print(tom)                    # ("Tom", 23)
```

Също така, за да се дефинира кортеж, може просто да се изброят стойностите, разделени със запетаи, без да се използват скоби:

```
tom = "Tom", 23
print(tom)              # ("Tom", 23)
```

Ако кортежът се състои от един елемент, тогава след единствения елемент от кортежа трябва да се постави запетая:

```
tom = ("Tom",)
```

За да се създаде кортеж от друг набор от елементи, като списък, може да се предаде списъка на функцията tuple(), която ще върне кортеж:

```
data = ["Tom", 37, "Google"]
tom = tuple(data)
print(tom)              # ("Tom", 37, "Google")
```

Може да се получи дължината на кортежа с помощта на вградената функция len():

```
tom = ("Tom", 37, "Google")
print(len(tom))     # 3
```

Достъп до елементи от кортеж

Достъпът до елементи в кортеж се осъществява по същия начин, както в списък, чрез индекс. Индексирането също започва от нула при получаване на елементи от началото на списъка и от -1 при получаване на елементи от края на списъка:

```
tom = ("Tom", 37, "Google", "software developer")
print(tom[0])      # Tom
print(tom[1])      # 37
print(tom[-1])     # software developer
```

Но тъй като кортежът е неизменяем тип, не може да се променят елементите му, т.е. записът:

tom[1] = "Tim„   (**няма да работи.)**

Ако е необходимо, може да се декомпозира кортежа на отделни променливи:

```
name, age, company, position = ("Tom", 37, "Google", "software developer")
print(name)        # Tom
print(age)         # 37
print(position)    # software developer
print(company)     # Google
```

Получаване на подкортеж

Както при списъците, може да се получи част от кортеж като друг кортеж:

```
tom = ("Tom", 37, "Google", "software developer")

# подкортеж от 1-ви до 3-ти елемент (без да се включва 1-ви и 3-ти елемент)
print(tom[1:3])     # (37, "Google")

# подкортеж от 0 до 3-ти елемент (без да се включва 3-ти елемент)
print(tom[:3])     # ("Tom", 37, "Google")

# подкортеж от 1 до последния елемент (не се включва 0-ят елемент)
print(tom[1:])     # (37, "Google", "software developer")
```

Кортеж като параметър и резултат от функции

Особено удобно е да се използват кортежи, когато трябва да се върнат няколко стойности от функция наведнъж. Когато функция връща множество стойности, тя всъщност връща кортеж:

```
def get_user():
    name = "Tom"
    age = 22
    company = "Google"
    return name, age, company

user = get_user()
print(user)     # ("Tom", 22, "Google")
```

Когато предавате кортеж към функция с помощта на оператора *, той може да бъде разложен на отделни стойности, които се предават на параметрите на функцията:

```
def print_person(name, age, company):
    print(f"Name: {name}  Age: {age}  Company: {company}")

tom = ("Tom", 22)
print_person(*tom, "Microsoft")     # Name: Tom  Age: 22  Company: Microsoft

bob = ("Bob", 41, "Apple")
print_person(*bob)                  # Name: Bob  Age: 41  Company: Apple
```

Итерация върху кортежи

Може да използвате стандартните цикли for и while, за да итерирате над кортеж. **С цикъл for:**

```
tom = ("Tom", 22, "Google")
for item in tom:
    print(item)
```

С цикъл while:

```
tom = ("Tom", 22, "Google")
i = 0
while i < len(tom):
    print(tom[i])
    i += 1
```

Проверка дали съществува стойност

Що се отнася до списък, използващ израз елемент in кортеж, може да се провери наличието на елемент в кортеж:

```
user = ("Tom", 22, "Google")
name = "Tom"
if name in user:
    print("Името на потребителя е Tom")
else:
    print("Потребителят има различно име")
```

Обхвати

Диапазоните или диапазонът представляват неизменяем последователен набор от числа. За да се създадат диапазони, използвайте range, който има следните форми:

- диапазон(стоп): връща всички цели числа от 0 до стоп
- диапазон(старт, стоп): връща всички цели числа между начало (включително) и стоп (с изключение).
- диапазон(начало, спиране, стъпка): връща цели числа между начало (включително) и стоп (без да се включва), увеличено със стойността на стъпката

Примери за извикване на функция за диапазон:

```
range(5)            # 0, 1, 2, 3, 4
range(1, 5)         # 1, 2, 3, 4
range(2, 10, 2)     # 2, 4, 6, 8
range(10, 2, -2)    # 10 8 6 4
```

Диапазоните се използват най-често в циклите for. Например, нека се отпечатат всички числа от 0 до 4 последователно:

```
for i in range(5):
    print(i, end=" ")
# Изход в конзолата
# 0, 1, 2, 3, 4
```

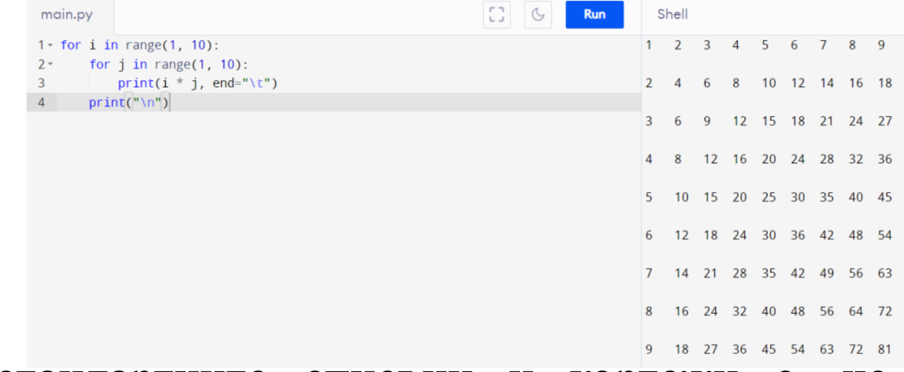
Друг пример е да се покаже таблицата за умножение:

```
for i in range(1, 10):
    for j in range(1, 10):
        print(i * j, end="\t")
    print("\n")
```

1	2	3	4	5	6	7	8	9
2	4	6	8	10	12	14	16	18
3	6	9	12	15	18	21	24	27
4	8	12	16	20	24	28	32	36
5	10	15	20	25	30	35	40	45
6	12	18	24	30	36	42	48	54
7	14	21	28	35	42	49	56	63
8	16	24	32	40	48	56	64	72
9	18	27	36	45	54	63	72	81

Ако е необходим последователен списък с числа, тогава е удобно да се използва функцията за диапазон, за да бъде създаден:

```
numbers = list(range(10))
print(numbers)      # [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
numbers = list(range(2, 10))
print(numbers)      # [2, 3, 4, 5, 6, 7, 8, 9]
numbers = list(range(10, 2, -2))
print(numbers)      # [10, 8, 6, 4]
```



Предимството на диапазоните пред стандартните списъци и кортежи е, че диапазонът винаги ще заема едно и също малко количество памет, независимо какъв набор от числа представлява диапазонът. В действителност диапазонът съхранява само началните, крайните и нарастващите стойности.

Речници

Речникът в Python съхранява колекция от елементи, където всеки елемент има уникален ключ и някаква стойност, свързана с него.

Дефиницията на речника, синтаксис:

dictionary = { ключ1:значение1, ключ2:значение2, ...}

Къдравите скоби дефинират последователност от елементи, разделена със запетая, където за всеки елемент първо се посочва ключът и стойността му се разделя с двоеточие.

Нека се дефинира речник:

users = {1: "Tom", 2: "Bob", 3: "Bill"}

Потребителският речник използва числа като ключове и низове като стойности. Т.е. елементът с ключ 1 има стойността "Tom", елементът с ключ 2 има стойността "Bob" и т.н.

Пример:

emails = {"tom@gmail.com": "Tom", "bob@gmail.com": "Bob", "sam@gmail.com": "Sam"}

Речникът за имейли използва низове като ключове - потребителски имейл адреси и низове - потребителски имена като стойности.

Но не е задължително ключовете и низовете да са от един и същи тип. Те могат да представляват различни видове:

objects = {1: "Tom", "2": True, 3: 100.6}

Може също да се дефинира празен речник без елементи:

objects = {}

или така:

objects = dict()

Преобразуване на списъци и кортежи в речник

Въпреки че речникът и списъкът са структурно различни типове, все още е възможно някои видове списъци да бъдат преобразувани в речник с помощта на вградената функция dict(). За да направите това, списъкът трябва да съхранява набор от вложени списъци. Всеки вложен списък трябва да се състои от два елемента - когато се преобразува в речник, първият елемент става ключ, а вторият става стойността:

```
users_list = [
    ["+111123455", "Tom"],
    ["+384767557", "Bob"],
    ["+958758767", "Alice"]
]
users_dict = dict(users_list)
print(users_dict)        # {"+111123455": "Tom", "+384767557": "Bob", "+958758767": "Alice"}
```

main.py | Run | Shell

```
1 users_list = [
2     ["+111123455", "Tom"],
3     ["+384767557", "Bob"],
4     ["+958758767", "Alice"]
5 ]
6 users_dict = dict(users_list)
7 print(users_dict)
```

```
{'+111123455': 'Tom', '+384767557': 'Bob', '+958758767': 'Alice'}
>
```

По същия начин, два кортежа могат да бъдат преобразувани в речник, който от своя страна съдържа два кортежа:

```
users_tuple = (
    ("+111123455", "Tom"),
    ("+384767557", "Bob"),
    ("+958758767", "Alice")
)
users_dict = dict(users_tuple)
print(users_dict)
```

Получаване и промяна на елементи

За препратка към елементите на речника, след неговото име, ключът на елемента е посочен в квадратни скоби:

```
dictionary[ключ]
```

Например, за да се получат и промени в елементите в речник:

```
users = {
    "+11111111": "Tom",
    "+33333333": "Bob",
    "+55555555": "Alice"
}

# получаване на елемент с ключ "+11111111"
print(users["+11111111"])      # Tom

# задаване на стойността на елемент с ключ "+33333333"
users["+33333333"] = "Bob Smith"
print(users["+33333333"])     # Bob Smith
```

Ако при задаване на стойността на елемент с такъв ключ няма елемент в речника, тогава той ще бъде добавен:

```
users["+4444444"] = "Sam"
```

Но ако се опита да се получи стойност с ключ, който не е в речника, тогава Python ще изведе KeyError:

```
user = users["+4444444"]       # KeyError
```

За да се предотврати тази ситуация, преди да се осъществи достъп до елемента, може да се провери за наличието на ключ в речника, като се използва ключа на израза в dictionary. Ако ключът е в речника, тогава този израз връща True:

```
key = "+4444444"
if key in users:
    user = users[key]
    print(user)
else:
    print("Елементът не е открит")
```

Може също да се използва метода get, за да се получат елементи, има две форми:

- **get(key):** връща елемента с ключа от речника. Ако няма елемент с този ключ, той връща None.
- **get(key, default):** връща елемента с ключа от речника. Ако няма елемент с такъв ключ, той връща стойността по подразбиране

```
users = {
    "+11111111": "Tom",
    "+33333333": "Bob",
    "+55555555": "Alice"
}
```

```
user1 = users.get("+55555555")
print(user1)    # Alice
user2 = users.get("+33333333", "Unknown user")
print(user2)    # Bob
user3 = users.get("+44444444", "Unknown user")
print(user3)    # Unknown user
```

Премахване

Инструкцията del се използва за премахване на елемент по ключ:

```
users = {
    "+11111111": "Tom",
    "+33333333": "Bob",
    "+55555555": "Alice"
}
del users["+55555555"]
print(users)    # { "+11111111": "Tom", "+33333333": "Bob"}
```

Но трябва да се има предвид, че ако такъв ключ не е в речника, тогава ще бъде изведено изключение KeyError. Ето защо, отново, преди изтриването е желателно да се провери наличието на елемент с даден ключ.

```
users = {
    "+11111111": "Tom",
    "+33333333": "Bob",
    "+55555555": "Alice"
}
key = "+55555555"
if key in users:
    del users[key]
    print(f"Елемент с ключ {key} е премахнат")
else:
    print("Елемента не е открит")
```

Друг метод за премахване е методът pop(). Има също две форми:

- pop(key): премахва елемента с ключ и връща премахнатия елемент. Ако няма елемент с дадения ключ, тогава се хвърля изключение KeyError.
- pop(key, default): премахва елемента с ключ и връща премахнатия елемент. Ако няма елемент с дадения ключ, тогава се връща по подразбиране.

```
users = {
    "+11111111": "Tom",
    "+33333333": "Bob",
    "+55555555": "Alice"
}
key = "+55555555"
user = users.pop(key)
print(user)     # Alice
user = users.pop("+4444444", "Unknown user")
print(user)     # Unknown user
```

Ако трябва да се премахнат всички елементи, тогава в този случай може да се използва метода clear():

```
users.clear()
```

Копиране и сливане на речници

Методът copy() копира съдържанието на речник, връщайки нов речник:

```
users = {"+1111111": "Tom", "+3333333": "Bob", "+5555555": "Alice"}
students = users.copy()
print(students)     # {"+1111111": "Tom", "+3333333": "Bob", "+5555555": "Alice"}
```

Методът update() обединява два речника:

```
users = {"+1111111": "Tom", "+3333333": "Bob"}

users2 = {"+2222222": "Sam", "+6666666": "Kate"}
users.update(users2)

print(users)    # {"+1111111": "Tom", "+3333333": "Bob", "+2222222": "Sam", "+6666666": "Kate"}
print(users2)   # {"+2222222": "Sam", "+6666666": "Kate"}
```

В този случай речникът на users2 остава непроменен. Променя се само потребителският речник, към който се добавят елементи от друг речник. Но ако е необходимо и двата изходни речника да останат непроменени и резултатът от обединението е някакъв трети речник, тогава може първо да се копира един речник в друг:

```
users3 = users.copy()
users3.update(users2)
```

Итерация на речник

Може да използвате цикъл for, за да преминете през речник:

```
users = {
   "+11111111": "Tom",
   "+33333333": "Bob",
   "+55555555": "Alice"
}
for key in users:
   print(f"Phone: {key}  User: {users[key]} ")
```

Когато се итерира през елементите, се получава ключа на текущия елемент и може да се използва, за да се получи самия елемент.

Друг начин за повторение на елементи е да се използва метода items():

```
users = {
   "+11111111": "Tom",
   "+33333333": "Bob",
   "+55555555": "Alice"
}
for key, value in users.items():
   print(f"Phone: {key}  User: {value} ")
```

Методът items() връща набор от кортежи. Всеки кортеж съдържа ключа и стойността на елемента, които при повторение можем веднага да влезем в променливите с ключ и стойност.

Има и отделни опции за итерация по ключове и итерация върху стойности. За да се премине през ключовете, може да се извика метода keys() на речника:

```
for key in users.keys():
    print(key)
```

Вярно е, че този метод на изброяване няма смисъл, тъй като без да се извика метода keys (), може да се изброят ключовете, както е показано по-горе.
За да се повторят само стойностите, може да се извика метода values() на речника :

```
            for value in users.values():
    print(value)
```

Сложни речници

В допълнение към най-простите обекти като числа и низове, в речниците могат да съхраняват и по-сложни обекти - същите списъци, кортежи или други речници:

```
users = {
    "Tom": {
        "phone": "+971478745",
        "email": "tom12@gmail.com"
    },
    "Bob": {
        "phone": "+876390444",
        "email": "bob@gmail.com",
        "skype": "bob123"
    }
}
```

В този случай стойността на всеки елемент от речника от своя страна представлява отделен речник.

За достъп до елементите на вложен речник, съответно, трябва да се използват два ключа:

```
old_email = users["Tom"]["email"]
users["Tom"]["email"] = "supertom@gmail.com"
print(users["Tom"])     # { phone": "+971478745", "email": "supertom@gmail.com }
```

Но ако се прави опит да се получи стойност чрез ключ, който не е в речника, Python ще изведе изключение KeyError:

```
tom_skype = users["Tom"]["skype"]   # KeyError
```

За да се избегне грешка, може да се провери за наличието на ключ в речника:

```
key = "skype"
if key in users["Tom"]:
    print(users["Tom"]["skype"])
else:
    print("skype is not found")
```

Във всички останали отношения работата със сложни и вложени речници е подобна на работата с обикновени речници.

Комплекти

Наборът съдържа само уникални елементи. За дефиниране на набор се използват къдрави скоби, в които са изброени елементите:

```
users = {"Tom", "Bob", "Alice", "Tom"}
print(users)    # {"Alice", "Bob", "Tom"}
```

Забележете, че въпреки че функцията за печат извежда елемента "Tom" веднъж, дефиницията на набора съдържа този елемент два пъти. Това е така, защото наборът съдържа само уникални стойности.
Също така, за да се дефинира набор, може да се използва функцията set() , на която се предава списък или набор от елементи:

```
people = ["Mike", "Bill", "Ted"]
users = set(people)
print(users)    # {"Mike", "Bill", "Ted"}
```

Функцията set е полезна за създаване на празен набор:

```
users = set()
```

За да получите дължината на набор, използвайте вградената функция len():

```
users = {"Tom", "Bob", "Alice"}
print(len(users))       # 3
```

Добавяне на елементи

За да добавите един елемент, се извиква методът add():

```
users = set()
users.add("Sam")
print(users)
```

Премахване на елементи

За да се премахне един елемент, се извиква методът remove(), предавайки елемента, който трябва да бъде премахнат. Но ако такъв елемент не е в набора, тогава ще се генерира грешка. Следователно, преди да изтриете, трябва да проверите за наличието на елемент, като използвате оператора in:

```
users = {"Tom", "Bob", "Alice"}

user = "Tom"
if user in users:
    users.remove(user)
print(users)    # {"Bob", "Alice"}
```

Може също да се използва и метода discard() за премахване, който няма да хвърли изключения, ако елементът липсва:

```
users = {"Tom", "Bob", "Alice"}

users.discard("Tim")    # елемент "Tim" не съществува, и метода не прави нищо
print(users)    #  {"Tom", "Bob", "Alice"}

users.discard("Tom")    # елемент "Tom" съществува, и метода премахва елемента
print(users)    #  {"Bob", "Alice"}
```

За да премахнете всички елементи, се извиква методът clear():

```
users.clear()
```

Итерация върху набор

Може да използвате цикъл for, за да преглеждате елементите:

```
users = {"Tom", "Bob", "Alice"}

for user in users:
    print(user)
```

При повторение всеки елемент се поставя в потребителската променлива.

Задайте операции

Използвайки метода copy(), може да копирате съдържанието на един набор в друга променлива:

```
users = {"Tom", "Bob", "Alice"}
students = users.copy()
print(students)     # {"Tom", "Bob", "Alice"}
```

Методът union() комбинира два набора и връща нов набор:

```
users = {"Tom", "Bob", "Alice"}
users2 = {"Sam", "Kate", "Bob"}

users3 = users.union(users2)
print(users3)   # {"Bob", "Alice", "Sam", "Kate", "Tom"}
```

Пресечната точка на множествата позволява да се получат само онези елементи, които са едновременно и в двата набора. Методът intersection() изпълнява операция за пресичане на набор и връща нов набор:

```
users = {"Tom", "Bob", "Alice"}
users2 = {"Sam", "Kate", "Bob"}
users3 = users.intersection(users2)
print(users3)   # {"Bob“}
```

Вместо метода на пресичане може да се използва операцията за логическо умножение:

```
users = {"Tom", "Bob", "Alice"}
users2 = {"Sam", "Kate", "Bob"}
print(users & users2)   # {"Bob"}
```

В този случай ще получим същия резултат. Друга операция - set different връща онези елементи, които са в първия набор, но липсват във втория. За да се получи разликата от набори, може да се използва метода difference или операцията на изваждане:

```
users = {"Tom", "Bob", "Alice"}
users2 = {"Sam", "Kate", "Bob"}
users3 = users.difference(users2)
print(users3)          # {"Tom", "Alice"}
print(users - users2)  # {"Tom", "Alice"}
```

Отделен вид разлика в набора, симетричната разлика, се произвежда с помощта на метода symmetric_difference() . Връща всички елементи от двата набора, с изключение на общите:

```
users = {"Tom", "Bob", "Alice"}
users2 = {"Sam", "Kate", "Bob"}
users3 = users.symmetric_difference(users2)
print(users3)  # {"Tom", "Alice", "Sam", "Kate"}
```

Връзки между множества

Методът issubset позволява да се разбере дали текущият набор е подмножество (тоест част) от друг набор:

```
users = {"Tom", "Bob", "Alice"}
superusers = {"Sam", "Tom", "Bob", "Alice", "Greg"}
print(users.issubset(superusers))   # True
print(superusers.issubset(users))   # False
```

Методът isuperset, от друга страна, връща True, ако текущият набор е супернабор (т.е. съдържа първият) от друг набор:

```
users = {"Tom", "Bob", "Alice"}
superusers = {"Sam", "Tom", "Bob", "Alice", "Greg"}
print(users.issuperset(superusers))   # False
print(superusers.issuperset(users))   # True
```

Замразен комплект

Типът замразен набор е вид набор, който не може да бъде променен. За да се създаде, се използва функцията frozenset:

users = frozenset({"Tom", "Bob", "Alice"})

На функцията frozenset се предава набор от елементи - списък, кортеж, друг набор. Не може да се добавят нови елементи към такъв набор, както и да се премахват съществуващи от него. Ето защо замразеният набор поддържа ограничен набор от операции:

- len(s): връща дължината на набора
- x in s: връща True, ако елемент x присъства в набор s
- x not in s: връща True, ако елемент x не е в набор s
- s.issubset(t): връща True, ако t съдържа набор от s
- s.issuperset(t): връща True, ако t се съдържа в s
- s.union(t): връща обединението на множествата s и t
- s.intersection(t): връща пресечната точка на s и t
- s.difference(t): връща разликата на множествата s и t
- s.copy(): връща копие на набора s

Използвана литература:

[1]. Joakim Sundnes, Introduction to Scientific Programming with Python, Simula Springer Briefs on Computing, 2020
[2]. Brian Heinold, A Practical Introduction to Python Programming, Department of Mathematics and Computer Science Mount St. Mary's University, 2012
[3]. David Mertz, Functional Programming in Python, O'Reilly Media, 2015
[4]. Steven Lott, Functional Python Programming, Published by Packt Publishing Ltd., 2015
[5]. Mark Lutz, Learning Python, Fourth Edition, O'Reilly Media, 2009

Струни. Работа със струни

Низът е поредица от символи на Unicode, затворени в кавички. Освен това Python позволява използването, както на единични, така и на двойни кавички за дефиниране на низове:

```
message = "Hello World!"
print(message)  # Hello World!
name = 'Tom'
print(name)  # Tom
```

Ако редът е дълъг, може да го разделите на части и да ги поставите на различни редове код. В този случай целият низ е затворен в скоби, а отделните му части са в кавички:

```
text = ("Laudate omnes gentes laudate "
      "Magnificat in secula ")
print(text)
```

Ако трябва да се дефинира многоредов текст, тогава такъв текст е затворен в тройни двойни или единични кавички:

```
'''
Това е коментар
'''
text = '''Laudate omnes gentes laudate
Magnificat in secula
Et anima mea laudate
Magnificat in secula
'''
print(text)
```

Когато се използват тройни единични кавички, не трябва да се бъркат с коментари: ако текстът в тройни единични кавички е присвоен на променлива, тогава това е низ, а не коментар.

Escape последователности в низ

Низът може да съдържа редица специални символи - escape-последователности. Някои от тях:

- **\:** позволява да се добави наклонена черта вътре в низа
- **\':** позволява да се добави единична кавичка вътре в низа
- **\":** позволява да се добавят двойни кавички вътре в низа
- **\n:** Преминава към нов ред
- **\t:** добавя раздел (4 отстъпа)

Нека се разгледат някои последователности:

```
text = "Message:\n\"Hello World\""
print(text)
```

Конзолен изход на програмата:

```
Message:
"Hello World"
```

Подобни последователности могат да помогнат при поставяне на цитат в низ, при направа на таблица, за преход на нов ред, но могат и да пречат. **Пример:**

```
path = "C:\python\name.txt"
print(path)
```

Променливата **path** съдържа път до файла, въпреки че, в низа има символи "\n", които ще се интерпретират, като escape последователност, което ще доведе до следния конзолен изход:

```
c:\python
ame.txt
```

За да се избегне подобна ситуация, символът **r** се поставя преди низа.

```
path = r"C:\python\name.txt"
print(path)
```

Вмъкване на стойности в низ

Python позволява да се вграждат стойностите на други променливи в низ. За да се направи това, вътре в низа, променливите се поставят в къдрави скоби {}, а символът f се поставя преди целия низ :

main.py

```
1 userName = "Tom"
2 userAge = 37
3 user = f"name: {userName}  age: {userAge}"
4 print(user)   # name: Tom  age: 37
```

Shell

```
name: Tom  age: 37
>
```

В този случай, стойността на променливата userName ще бъде вмъкната на място {userName}, а стойността на променливата userAge ще бъде вмъкната на място {userAge}.

Тема 8. Струни. Работа със струни. Основни низови методи. Форматиране

Референтни знаци в низ

Може да се получи достъп до отделните знаци на низ чрез индекс в квадратни скоби. Индексирането започва от нула, така че първият символ на низа ще има индекс 0. И ако се направи опит да се осъществи достъп до индекс, който не е в низа, ще се получи изключение **IndexError**. Например, в горния случай низът е дълъг 11 знака, така че неговите знаци ще имат индекси от 0 до 10. За достъп до знаци, започващи от края на низ, могат да се използват отрицателни индекси. И така, индекс -1 ще представлява последния знак, а -2 ще представлява предпоследния знак и т.н.:

```
string = "hello world"
c0 = string[0]  # h
print(c0)
c6 = string[6]  # w
print(c6)
c11 = string[11]  # грешка IndexError: string index out of range
print(c11)
```

main.py

```
1 string = "hello world"
2 c0 = string[0]  # h
3 print(c0)
4 c6 = string[6]  # w
5 print(c6)
6 c11 = string[11]  # грешка IndexError: string index out of range
7 print(c11)
```

Run

Shell

```
h
w
Traceback (most recent call last):
File "<string>", line 6, in <module>
IndexError: string index out of range
>
```

Тема 8. Струни. Работа със струни. Основни низови методи. Форматиране

Когато се работи със знаци, трябва да се има предвид, че **низът е неизменяем тип**, така че ако се направи опит да се промени всеки отделен знак от низ, ще се получи грешка, както в следния случай:

string = "hello world"
string[1] = "R"

```
main.py
1 string = "hello world"
2 string[1] = "R"
3
```

```
Shell
Traceback (most recent call last):
  File "<string>", line 2, in <module>
TypeError: 'str' object does not support item assignment
```

Може да се нулира напълно стойността на низ само като му се присвои различна стойност.

Итерация на линия

Може да се премине през всички знаци в низ с цикъл for :

string = "hello world"
for char in string:
    print(char)

```
main.py
1 string = "hello world"
2 for char in string:
3     print(char)
4
```

```
Shell
h
e
l
l
o

w
o
r
l
d
```

Получаване на подниз

Ако е необходимо, може да се получат не само отделните символи от низ, но и подниз. Затова се използва следният синтаксис:

- **string[:end]:** извлича поредица от знаци от индекс 0 до края на индекса (без да се включва)
- **string[start:end]:** извлича поредица от знаци от началото на индекса до края на индекса (без да се включва)
- **string[start:end:step]:** извличане на последователността от знаци от началото на индекса до края на индекса (без да се включва) чрез стъпка

Използват се всички опции за получаване на подниз:
string = "hello world"

```
# от 0 до 5 индекс
sub_string1 = string[:5]
print(sub_string1)      # hello

# от 2 до 5 индекс
sub_string2 = string[2:5]
print(sub_string2)      # llo

# от 2 до 9 индекс през един знак
sub_string3 = string[2:9:2]
print(sub_string3)      # lowr
```

main.py   Run   Shell

```
1  string = "hello world"
2  # от 0 до 5 индекс
3  sub_string1 = string[:5]
4  print(sub_string1)      # hello
5
6  # от 2 до 5 индекс
7  sub_string2 = string[2:5]
8  print(sub_string2)      # llo
9
10 # от 2 до 9 индекс през един знак
11 sub_string3 = string[2:9:2]
12 print(sub_string3)      # lowr
```

```
hello
llo
lowr
>
```

Конкатенация на низове

Една от най-често срещаните операции върху низове е тяхното обединяване или конкатенация. Операцията по добавяне се използва за конкатениране на низове:

```
name = "Tom"
surname = "Smith"
fullname = name + " " + surname
print(fullname)  # Tom Smith
```

main.py   Run   Shell

```
1  name = "Tom"
2  surname = "Smith"
3  fullname = name + " " + surname
4  print(fullname)  # Tom Smith
```

```
Tom Smith
>
```

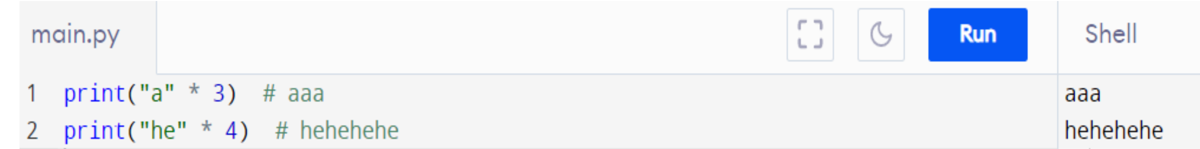
Свързването на два низа е лесно, но какво ще стане, ако трябва да се добави низ и число? В този случай трябва да се **прехвърли числото към низ с помощта на функцията str():**

```
name = "Tom"
age = 33
info = "Name: " + name + " Age: " + str(age)
print(info)  # Name: Tom Age: 33
```

Повторение на линия

За да се повтори низ определен брой пъти, се използва операцията умножение:

```
print("a" * 3)  # aaa
print("he" * 4)  # hehehehe
```



Сравнение на низове

Специално трябва да се спомене сравненията на низове. При сравняване на низове се вземат предвид знаците и техният регистър. И така, цифровият знак е условно по-малък от всеки азбучен знак. Главните букви на азбуката обикновено са по-малки от малките букви. Например:

Тема 8. Струни. Работа със струни. Основни низови методи. Форматиране

```
str1 = "1a"
str2 = "aa"
str3 = "Aa"
print(str1 > str2)  #
print(str2 > str3)  #
```

main.py Run Shell

```
1  str1 = "1a"
2  str2 = "aa"
3  str3 = "Aa"
4  print(str1 > str2)  # False, тъй като първият знак в str1 е цифра
5  print(str2 > str3)  # True, тъй като първият знак в str2 е с малки букви
```

```
False
True
>
```

Следователно низът "1a" е условно по-малък от низа "aa". Първият знак се сравнява първи. Ако водещите символи на двата низа представляват цифри, тогава по-малката цифра се счита за по-малка, например "1a" е по-малко от "2a". Ако първоначалните знаци представляват азбучни знаци в същия случай, тогава потърсете по азбучен ред. Така че "aa" е по-малко от "ba", а "ba" е по-малко от "ca". Ако първите знаци са еднакви, вторите знаци, ако има такива, се вземат предвид. Чувствителността на малки и големи букви не винаги е желателна, тъй като всъщност работата е с едни и същи низове. В този случай, преди да се направи сравнението, може да се прехвърлят двата низа към един от регистрите.

Функцията lower() преобразува низ в малки букви, а функцията upper() го преобразува в главни букви.

main.py Run Shell

```
1  str1 = "Tom"
2  str2 = "tom"
3  print(str1 == str2)  # False - струните не са равни
```

```
False
>
```

```
str1 = "Tom"
str2 = "tom"
print(str1 == str2)  # False - струните не са равни
```

print(str1.lower() == str2.lower()) # True

функции ord и len

Тъй като низът съдържа Unicode символи, можем да се използва функцията ord(), за да се получи числовата стойност за Unicode символ:

print(ord("A")) # 65

Може да се използва функцията len(), за да се получи дължината на низа:

string = "hello world"

length = len(string)

print(length) # 11

```
main.py                                   Run    Shell
1  str1 = "Tom"                                  True
2  str2 = "tom"                                  65
3  print(str1.lower() == str2.lower())           11
4                                                >
5  print(ord("A"))     # 65
6
7  string = "hello world"
8  length = len(string)
9  print(length)   # 11
```

Търсене в низ

С помощта на израз term in string може да се намери term-а на подниз в друг низ. Ако поднизът бъде намерен, тогава изразът ще върне стойността True, в противен случай стойността ще бъде върната False:

string = "hello world"

exist = "hello" in string

print(exist) # True

exist = "sword" in string

print(exist) # False

```
main.py                                   Run    Shell
1  string = "hello world"                        True
2  exist = "hello" in string                     False
3  print(exist)    # True                        >
4
5  exist = "sword" in string
6  print(exist)    # False
```

Основни низови методи

Основните низови методи, които могат да се приложат са:

- **isalpha():** връща True, ако низът се състои само от азбучни знаци
- **islower():** връща True, ако низът се състои само от малки букви
- **isupper():** връща True, ако всички знаци в низа са главни
- **isdigit():** връща True, ако всички знаци в низ са цифри
- **isnumeric():** връща True, ако низът е число
- **startswith(str):** връща True, ако низът започва с подниз str
- **endswith(str):** връща True, ако низът завършва с подниз str
- **low():** преобразува низ в малки букви
- **upper():** преобразува низ в главни букви
- **title():** началните знаци на всички думи в низа се преобразуват в главни букви

- **title():** началните знаци на всички думи в низа се преобразуват в главни букви
- **capitalize():** главна буква само на първата дума от низа
- **lstrip():** премахва водещите интервали от низ
- rstrip(): премахва крайните интервали от низ
- **strip():** премахва началните и крайните интервали от низ
- **ljust(width):** ако дължината на низа е по-малка от параметъра ширина, отдясно на низа се добавят интервали, за да допълнят стойността на ширината, а самият низ се подравнява вляво
- **rjust(width):** ако дължината на низа е по-малка от параметъра ширина, отляво на низа се добавят интервали, за да допълнят стойността на ширината, а самият низ е подравнен вдясно
- **center(width):** ако дължината на низа е по-малка от параметъра ширина, интервали се добавят равномерно отляво и отдясно на низа, за да допълнят стойността на ширината, а самият низ е центриран
- **find(str[, start [, end]]):** връща индекса на подниз в низ. Ако поднизът не бъде намерен, се връща числото -1
- **replace(old, new[, num]):** замества един подниз в низ с друг
- **split([delimeter[, num]]):** разделя низ на поднизове в зависимост от разделителя
- **join(strs):** конкатенира низове в един низ, чрез вмъкване на специфичен разделител между тях.

Например, ако числото вече е въведено от клавиатурата, тогава преди да се преобразува въведеното в число, може да се провери дали числото е валидно с помощта на метода isnumeric() и така ще се изпълни операцията за преобразуване:

```
string = input("Въведете число: ")
if string.isnumeric():
    number = int(string)
    print(number)
```

```
main.py                                   Run    Shell
1  string = input("Въведете число: ")            Въведете число: 53
2  if string.isnumeric():                         53
3      number = int(string)                       >
4      print(number)
```

Проверка дали низ започва или завършва с конкретен подниз:

```
file_name = "hello.py"
starts_with_hello = file_name.startswith("hello")
ends_with_exe = file_name.endswith("exe")
print(starts_with_hello)
print(ends_with_exe)
```

```
main.py                                                          Run    Shell
1  file_name = "hello.py"                                               True
2  starts_with_hello = file_name.startswith("hello")   # True           False
3  ends_with_exe = file_name.endswith("exe")           # False          >
4  print(starts_with_hello)
5  print(ends_with_exe)
```

Премахване на интервали в началото и кра

```
string = "   hello  world!  "
string = string.strip()
print(string)           # hello  world!
```

```
main.py                                        Run    Shell
1  string = "   hello  world!  "                      hello  world!
2  string = string.strip()                            >
3  print(string)          # hello  world!
```

Допълване на низ с интервали и подравняв

```
print("iPhone 7:", "52000".rjust(10))
print("Huawei P10:", "36000".rjust(10))
```

Изход на конзолата:

```
iPhone 7: 52000
Huawei P10: 36000
```

```
main.py                                        Run    Shell
1  print("iPhone 7:", "52000".rjust(10))             iPhone 7:       52000
2  print("Huawei P10:", "36000".rjust(10))           Huawei P10:      36000
```

Тема 8. Струни. Работа със струни. Основни низови методи. Форматиране

Търсене в низ

За да се намери подниз в низ, Python използва метода find(), който връща индекса на първото появяване на подниз в низ и има три форми:

- **find(str):** подниз str се търси от началото на низа до края му
- **find(str, start):** началният параметър определя началния индекс, от който да се търси
- **find(str, start, end):** параметърът end определя крайния индекс, към който ще отиде търсенето

Ако поднизът не бъде намерен, методът връща -1:

```
welcome = "Hello world! Goodbye world!"
index = welcome.find("wor")
print(index)       # 6
 # търсене от индекс 10
index = welcome.find("wor",10)
print(index)       # 21
 # търсене от 10-ти до 15-ти индекс
index = welcome.find("wor",10,15)
print(index)       # -1
```

Замяна в низ

За да замените един подниз в низ с друг, използвайте метода replace() :

- **replace(old, new):** замества подниза old с нов
- **replace(old, new, num):** параметърът num указва колко пъти поява на подниза old да се замени с new

```
phone = "+1-234-567-89-10"
 # замяна на тирета с интервали
edited_phone = phone.replace("-", " ")
print(edited_phone)     # +1 234 567 89 10
 # премахване на тире
edited_phone = phone.replace("-", "")
print(edited_phone)     # +12345678910
 # заместване само на първото тире
edited_phone = phone.replace("-", "", 1)
print(edited_phone)     # +1234-567-89-10
```


```
main.py
1  phone = "+1-234-567-89-10"
2   # замяна на тирета с интервали
3  edited_phone = phone.replace("-", " ")
4  print(edited_phone)     # +1 234 567 89 10
5   # премахване на тире
6  edited_phone = phone.replace("-", "")
7  print(edited_phone)     # +12345678910
8   # заместване само на първото тире
9  edited_phone = phone.replace("-", "", 1)
10 print(edited_phone)     # +1234-567-89-10
```


```
Shell
+1 234 567 89 10
+12345678910
+1234-567-89-10
>
```

Разделяне на поднизове

Методът split() разделя низ на списък с поднизове, в зависимост от разделителя. Разделителят може да бъде произволен знак или последователност от знаци. Този метод има следните форми:

- **split():** пространството се използва като разделител
- **split(delimeter):** разделител се използва като разделител
- **split(delimeter, num):** Параметърът num указва колко пъти появата на разделител се използват за разделяне. Останалата част от низа се добавя към списъка, без да се разделя на поднизове.

```
text = "Честита Коледа,  и Весела Нова 2023 година!"
splitted_text = text.split()
print(splitted_text)       # разделяне на думи
print(splitted_text[5])     # извеждане на поредна дума в израза
splitted_text = text.split(",")
print(splitted_text)        # извеждане на поредна дума в израза
print(splitted_text[0])     #  извеждане на израза до запетаята
splitted_text = text.split(" ", 3) # извеждане на думите с разделител празен символ
print(splitted_text)
print(splitted_text[0]) # извеждае на първата дума в низа
```

main.py

```
1  text = "Честита Коледа,  и Весела Нова 2023 година!"
2  splitted_text = text.split()
3  print(splitted_text)       # разделяне на думи
4
5  print(splitted_text[5])     # извеждане на поредна дума в израза
6
7  splitted_text = text.split(",")
8  print(splitted_text)        # извеждане на поредна дума в израза
9
10 print(splitted_text[0])     #  извеждане на израза до запетаята
11
12
13 splitted_text = text.split(" ", 3)
14 # извеждане на думите с разделител празен символ
15
16 print(splitted_text)
17 print(splitted_text[0])
18 # извеждае на първата дума в низа
```

Shell

```
['Честита', 'Коледа,', 'и', 'Весела', 'Нова', '2023', 'година!']
2023
['Честита Коледа', '  и Весела Нова 2023 година!']
Честита Коледа
['Честита', 'Коледа,', '', 'и Весела Нова 2023 година!']
Честита
>
```

Съединяване на низове

До тук бяха разгледани най-простите операции върху низове, като бе посочено как да свържем низове с помощта на операцията **text.split().** Друга възможност за свързване на низове е **методът join():** той обединява списък от низове. Освен това текущият ред, на който се извиква този метод, се използва като разделител:

words = ["Let", "me", "speak", "from", "my", "heart", "in", "English"]
 # разделител - интервал
sentence = " ".join(words)
print(sentence) # Let me speak from my heart in English
# разделител – вертикална черта
sentence = " | ".join(words)
print(sentence) # Let | me | speak | from | my | heart | in | English

```
main.py
1 words = ["Let", "me", "speak", "from", "my", "heart", "in", "English"]
2  # разделител - интервал
3 sentence = " ".join(words)
4 print(sentence)  # Let me speak from my heart in English
5
6 # разделител – вертикална черта
7 sentence = " | ".join(words)
8 print(sentence)  # Let | me | speak | from | my | heart | in | English
```

```
Shell
Let me speak from my heart in English
Let | me | speak | from | my | heart | in | English
>
```

Вместо списък, може да се подаде обикновен низ към метода на присъединяване, след което разделителят ще бъде вмъкнат между знаците на този низ:

word = "hello"
joined_word = "|".join(word)
print(joined_word)   # h|e|l|l|o

```
main.py
1 word = "hello"
2 joined_word = "|".join(word)
3 print(joined_word)      # h|e|l|l|o
```

```
Shell
h|e|l|l|o
>
```

Форматиране

Вмъкване на някои стойности в низ, може да стане, като се постави префикс на низа с f :

```
first_name="Tom"
text = f"Hello, {first_name}."
print(text)     # Hello, Tom.
name="Bob"
age=23
info = f"Name: {name}\t Age: {age}"
print(info)     # Name: Bob  Age: 23
```


main.py Run Shell

```
1 first_name="Tom"
2 text = f"Hello, {first_name}."
3 print(text)     # Hello, Tom.
4 name="Bob"
5 age=23
6 info = f"Name: {name}\t Age: {age}"
7 print(info)     # Name: Bob  Age: 23
```

```
Hello, Tom.
Name: Bob    Age: 23
>
```


В Python има и друг алтернативен начин, който предоставя метода format(). Този метод позволява да се форматира низ, като се вмъкват определени стойности в него на мястото на заместители. За вмъкване в низ се използват специални параметри, които са рамкирани с къдрави скоби ({}).

Именувани параметри

Във форматирания низ може да се дефинират параметри, в метода може да се използва format() за да се предадат стойности за тези параметри:

```
text = "Hello, {first_name}.".format(first_name="Tom")
print(text)     # Hello, Tom.
info = "Name: {name}\t Age: {age}".format(name="Bob", age=23)
print(info)     # Name: Bob  Age: 23
```


main.py Run Shell

```
1 text = "Hello, {first_name}.".format(first_name="Tom")
2 print(text)     # Hello, Tom.
3 info = "Name: {name}\t Age: {age}".format(name="Bob", age=23)
4 print(info)     # Name: Bob  Age: 23
```

```
Hello, Tom.
Name: Bob    Age: 23
>
```


Освен това в метода format аргументите се дефинират с … низа. Така че, ако параметърът се извика first_name, както в първия случай, тогава аргументът, на който е присвоена стойността, също се извиква first_name.

Параметри по позиция

Можем също така **последователно** да се предаде набор от аргументи към метода format и да се вмъкнат тези аргументи в самия низ за форматиране, като се посочи техният брой в къдрави скоби (номерирането започва от нула):

info = "Name: {0}\t Age: {1}".format("Bob", 23)

print(info) # Name: Bob Age: 23

В този случай аргументите могат да бъдат вмъкнати в низа няколко пъти:

text = "Hello, {0} {0} {0}.".format("Tom")

```
main.py
1 info = "Name: {0}\t Age: {1}".format("Bob", 23)
2 print(info)      # Name: Bob   Age: 23
3
4 text = "Hello, {0} {0} {0}.".format("Tom")
5 print(text)
```

```
Shell
Name: Bob     Age: 23
Hello, Tom Tom Tom.
>
```

Замени

Друг начин за предаване на форматирани стойности към низ е използването на заместванията или специални заместители, на мястото на които се вмъкват конкретни стойности. За форматиране може да се използват следните заместители:

- **s:** За вмъкване на редове
- **d:** За вмъкване на цели числа
- **f:** За вмъкване на дробни числа. За този тип също така е възможно да се определи броят на десетичните знаци чрез точка
- **%:** Умножава стойността по 100 и добавя знак за процент
- **e:** Отпечатва число в научна нотация

Общият синтаксис за заместител е както следва:

{:плейсхолдер}

Могат да се добавят допълнителни параметри в зависимост от заместителя. Например, за да се форматират числа с плаваща стойност, може да се използват следните опции

{:[количество_символи][запетая][.брой_цифри_в_дробната_част] **плейсхолдер}**

Тема 8. Струни. Работа със струни. Основни низови методи. Форматиране

При извикване на метода format стойностите се предават към него като аргументи, които се вмъкват на мястото на заместителите:

welcome = "Hello {:s}"
name = "Tom"
formatted_welcome = welcome.format(name)
print(formatted_welcome)   # Hello Tom

```
main.py                                   Run      Shell
1 welcome = "Hello {:s}"                           Hello Tom
2 name = "Tom"                                     >
3 formatted_welcome = welcome.format(name)
4 print(formatted_welcome)       # Hello Tom
```

В резултат на това методът format() връща нов форматиран низ. Форматиране на цели числа:

source = "{:d} символи"
number = 5
target = source.format(number)
print(target)   # 5 символи

```
main.py                                   Run      Shell
1 source = "{:d} символи"                          5 символи
2 number = 5                                       >
3 target = source.format(number)
4 print(target)   # 5 символи
```

Ако числото, което трябва да се форматира, е по-голямо от 999, тогава може да се посочи в дефиницията на заместителя, че искаме да използваме запетая като разделител на хилядите:

source = "{:,d} символи"
print(source.format(5000))   # 5,000 символи

Освен това, заместителите могат да се използват и във f-низове:

n = 5000
source = f"{n:,d} символи"
print(source)   # 5,000 символа

```
main.py                                   Run      Shell
1 source = "{:,d} символи"                         5,000 символи
2 print(source.format(5000))    # 5,000 символи    5,000 символи
3 n = 5000                                         >
4
5 source = f"{n:,d} символи"
6 print(source)   # 5,000 символа
```

За дробни числа, т.е. тези от тип float, преди кода за място след точката, може да се посочи колко знака в дробната част искаме да покажем:

```
number = 23.8589578
print("{:.2f}".format(number))   # 23.86
print("{:.3f}".format(number))   # 23.859
print("{:.4f}".format(number))   # 23.8590
print("{:,.2f}".format(10001.23554))    # 10,001.24
```

main.py | Run | Shell

```
1 number = 23.8589578
2 print("{:.2f}".format(number))   # 23.86
3 print("{:.3f}".format(number))   # 23.859
4 print("{:.4f}".format(number))   # 23.8590
5 print("{:,.2f}".format(10001.23554))    # 10,001.24
```

```
23.86
23.859
23.8590
10,001.24
>
```

Друг параметър позволява да се зададе минималната ширина на форматираната стойност в знаци:

```
print("{:10.2f}".format(23.8589578))    #     23.86
print("{:8d}".format(25))               #     25
```

main.py | Run | Shell

```
1 print("{:10.2f}".format(23.8589578))    #     23.86
2 print("{:8d}".format(25))               #     25
```

```
23.86
25
```

Подобен пример с f-низове:

```
n1 = 23.8589578
print(f"{n1:10.2f}")   #     23.86
n2 = 25
print(f"{n2:8d}")      #     25
```

main.py | Run | Shell

```
1 n1 = 23.8589578
2 print(f"{n1:10.2f}")    #     23.86
3 n2 = 25
4 print(f"{n2:8d}")       #     25
```

```
23.86
      25
>
```

За теглене на лихва е по-добре да се използва кода "%":

```
number = .12345
print("{:%}".format(number))       # 12.345000%
print("{:.0%}".format(number))     # 12%
print("{:.1%}".format(number))     # 12.3%

print(f"{number:%}")       # 12.345000%
print(f"{number:.0%}")     # 12%
print(f"{number:.1%}")     # 12.3%
```

main.py   Run   Shell

```
1 number = .12345
2 print("{:%}".format(number))       # 12.345000%
3 print("{:.0%}".format(number))     # 12%
4 print("{:.1%}".format(number))     # 12.3%
5
6 print(f"{number:%}")       # 12.345000%
7 print(f"{number:.0%}")     # 12%
8 print(f"{number:.1%}")     # 12.3%
```

```
12.345000%
12%
12.3%
12.345000%
12%
12.3%
>
```


За показване на число в експоненциална нотация се използва заместителят "е":

```
number = 12345.6789
print("{:e}".format(number))       # 1.234568e+04
print("{:.0e}".format(number))     # 1e+04
print("{:.1e}".format(number))     # 1.2e+04

print(f"{number:e}")       # 1.234568e+04
print(f"{number:.0e}")     # 1e+04
print(f"{number:.1e}")     # 1.2e+04
```

main.py   Run   Shell

```
1 number = 12345.6789
2 print("{:e}".format(number))       # 1.234568e+04
3 print("{:.0e}".format(number))     # 1e+04
4 print("{:.1e}".format(number))     # 1.2e+04
5
6 print(f"{number:e}")       # 1.234568e+04
7 print(f"{number:.0e}")     # 1e+04
8 print(f"{number:.1e}")     # 1.2e+04
```

```
1.234568e+04
1e+04
1.2e+04
1.234568e+04
1e+04
1.2e+04
>
```

Тема 8. Струни. Работа със струни. Основни низови методи. Форматиране

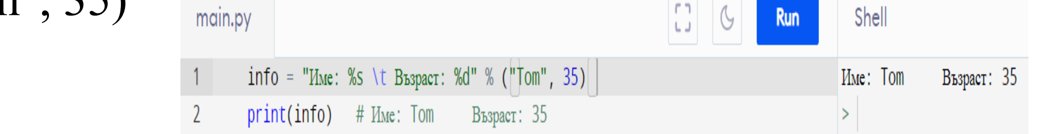
Форматиране без метода за форматиране

Има и друг начин за форматиране със следния синтаксис:

линия%(параметър1, параметър2,..параметърN)

Т.е., в началото има ред, който съдържа същите заместители, които бяха обсъдени по-горе (с изключение на заместващия %), след реда се поставя знак за процент % и след това списък със стойности, които се вмъкват в линията. Всъщност знакът за процента представлява операция, която води до нов ред:

```
info = "Име: %s \t Възраст: %d" % ("Tom", 35)
print(info)   # Име: Tom     Възраст: 35
```



Заместителят е последван от знак за процент и за разлика от функцията за форматиране тук не се изискват фигурни скоби. Освен това тук се използват и методи за форматиране на числа:

```
number = 23.8589578
print("%0.2f  - %e" % (number, number))   # 23.86  - 2.385896e+01
```

```
main.py
1 number = 23.8589578
2 print("%0.2f  - %e" % (number, number))   # 23.86  - 2.385896e+01

Shell
23.86  - 2.385896e+01
```

Програма за броене на думи

Нека да разгледаме работата с низове с помощта на малък пример, който представлява програма за броене на думи. Нека целият програмен код изглежда така:

```
# Програма за броене на думи във файл
import os


def get_words(filename):
    with open(filename, encoding="utf8") as file:
        text = file.read()
    text = text.replace("\n", " ")
    text = text.replace(",", "").replace(".", "").replace("?", "").replace("!", "")
    text = text.lower()
    words = text.split()
    words.sort()
    return words
```

```
def get_words_dict(words):
    words_dict = dict()

    for word in words:
        if word in words_dict:
            words_dict[word] = words_dict[word] + 1
        else:
            words_dict[word] = 1
    return words_dict

def main():
    filename = input("Въведете пътя до файла: ")
    if not os.path.exists(filename):
        print("Посоченият файл не съществува")
    else:
        words = get_words(filename)
        words_dict = get_words_dict(words)
        print(f"Брой думи: {len(words)}")
        print(f"Брой уникални думи: {len(words_dict)}")
        print("Всички използвани думи:")
        for word in words_dict:
            print(word.ljust(20), words_dict[word])
if __name__ == "__main__":
    main()
```

Тук във функцията get_words() се извършва първоначалното сегментиране на текста в думи. В този случай всички препинателни знаци се премахват, а преводите на дренажа се заменят с интервали. След това текстът се разделя на думи. Разделителят по подразбиране е интервал. По-нататък във функцията get_words_dict() се получава речник на думите, където ключът е уникална дума, а стойността е броят на срещанията на тази дума в текста. В основната функция се въвежда пътя до файла и се извикват по-горе дефинираните функции, както и изходът на цялата статистика. Конзолен изход на програмата:

Въведете пътя до файла: C:\SomeDir\hello.txt – Това е примерна програма с примерни данни!!!

Брой думи: 66

Брой уникални думи: 54

Всички използвани думи:

благодетел 2

в 1

общо 1

ти 1

горчица 1

нейната 1

ако 3

още 1

Използвана литература:

[1]. Joakim Sundnes, Introduction to Scientific Programming with Python, Simula Springer Briefs on Computing, 2020
[2]. Brian Heinold, A Practical Introduction to Python Programming, Department of Mathematics and Computer Science Mount St. Mary's University, 2012
[3]. David Mertz, Functional Programming in Python, O'Reilly Media, 2015
[4]. Steven Lott, Functional Python Programming, Published by Packt Publishing Ltd., 2015
[5]. Mark Lutz, Learning Python, Fourth Edition, O'Reilly Media, 2009