Абстракция. Функции. Рекурсия

Филип Андонов

14 юни 2022 г.

Съдържание

Функции

Програмирането повишава ефективността като премахва излишното повтаряне на едни и същи действия. Когато в една програма има сходен код, който се изпълнява на различни места, практиката е той да се отдели в подпрограма (процедура или функция). Освен това, когато се използват подпрограми със смислени имена, кодът става по-четим, защото без тях детайлите пречат да се разбере логиката.

В Python дали идентификатор е име на променлива или функция е трудно за определяне. За да проверим дали нещо може да бъде извикано като функция, използваме вградената такава, наречена callable.

Listing 1: Функция callable

```
>>> import math
>>> x = 5
>>> callable(x)
False
>>> callable(math.sqrt)
True
```

Дефиниране

Дефинирането на функции става по следния лесен начин:

Listing 2: Дефиниране на функция

```
1 def hello(name):
2     return 'Hello, ' + name + '!'
3     print(hello 'world')
```

Дефиниране

В Python няма разлика между процедура и функция като начин на деклариране и ключови думи, просто процедурите не съдържат оператор return. Във функциите, както и в други езици, операторите след return не се изпълняват.

Listing 3: Процедури в Python

```
1 def bmi(weight, height):
2     return weight / (height * height)
3     print("After the return statement")
```

Дефиниране

Всъщност, когато няма return клауза, функцията връща стойност, просто тя е None. В примерът имаме функция с поведение на процедура. Тя печата изчислената стойност. Но когато разпечатим резултата, върнат от нея, виждаме че той е None.

Listing 4: Процедурите връщат стойност

```
1 def bmi(weight, height):
2     print(weight / (height * height))
3 bmi(80,1.75)
4 print(bmi(80,1.75))
5 26.1224489796
6 None
```

Коментари могат да се пишат стандартно с #коментар, но ако се напише символен низ в началото на функцията, то той става част от нея и се нарича docstring и може да се достъпи както е показано на ред 5.

Listing 5: Docstring

```
1 def bmi(weight, height):
2     'Calculates Body mass index, weight in kg,
        height in meters'
3     return weight/(height*height)
4 print(bmi(80,1.75))
5 print(bmi.__doc__)
```

Параметри

Имената, които пишем в скобите след def, се наричат формални параметри, а стойностите, които подаваме при извикване на функцията – фактически параметри или аргументи. При извикване на функция с дадени аргументи, те се присвояват на съответстващите им параметри. Както и в другите езици, функциите имат своя област на видимост на променливите, така че всяко присвояване на стойност на параметър не променя нищо извън функцията.

Listing 6: Неизменими типове като аргументи

```
1 def test(param):
2     param = "very_important"
3 p = "alabala"
4 test(p)
5 print(p)
```

Символните низове, числата и кортежите са неизменими, те не могат да се модифицират. Но интересното е какво става, когато подадем като аргумент изменим тип данни (например списък):

Listing 7: Измениими типове като аргументи

```
1 def change_param(param):
2     param[0] = "very important"
3 p = ["alabala","portakala"]
4 change_param(p)
5 print(p[0])
```

Ред 5 ще изведе very important, т.е. промяната, извършена вътре във функцията остава и след нейното приключване. Това, което се случва е, че param е нова референция към същия списък. Така имаме две референции към един и същ списък. Същият механизъм, но без функция:

Listing 8: Обяснение на измеими аргументи

```
1 p = ["alabala","portakala"]
2 p2 = p
3 p2[0] = "very important"
4 print(p[0])
```

Параметри

Ако наистина трябва да подадем аргумент от изменим тип, който да е не може да се променя във функцията, то трябва да направим копие. Това се постига с използването на срезове, тъй като те винаги връщат копие. Програмата ще изведе alabala, тъй като функцията работи с копието на p, а не го променя.

Listing 9: Защита на агументи от изменим тип

```
1 def change_param(param):
2     param[0] = "very important"
3 p = ["alabala","portakala"]
4 change_param(p[:])
5 print(p[0])
```

Ако аргументът е неизменим, не можем да направим промяна, която да остане и извън функцията. Например в Pascal има функция за инкрементиране на целочислена променлива Inc и можем да напишем нещо от типа:

Listing 10: Действие на функция Inc в Pascal

```
1 a := 5;
2 Inc(a);
3 writeln(a);
4 6
```

Параметри

В Python за съжаление това не е възможно. Най-близкото като ефект е:

Listing 11: Аналог на Inc в Python

```
1 def inc(x): return x + 1
2 >>> foo = 10
3 >>> foo = inc(foo)
4 >>> foo
5 11
```

Функции като аргумент

В програмните езици е полезно да може да се предават функции (по-точно референции към функции) като аргумент на други функции. Често това са т. нар. callback функции, които се извикват от функцията-получател, когато тя е готова за тях.

Listing 12: Функция като аргумент

```
1 def callback():
2     print("callback")

4 def accept(function):
5     print("running function")
6     function()
7     print("end function")

9 accept(callback)
```

Именувани параметри

Дотук разгледаните примери са за позиционни параметри, които са аналогични на тези в C, например. Python ни улеснява и тук, като дава възможност при извикване на функция да се указва името на параметъра, чиято стойност се подава. Така подредбата на подадените параметри няма значение.

Listing 13: Пример с позиционни параметри

```
1 def bmi(weight, height):
2     print(weight / (height * height))
3 bmi(height = 1.75,weight = 80)
```

Именувани параметри

Именуваните параметри изискват повече писане, но предимството им е, че внасят яснота. Нека сравним следните две обръщения към една и съща функция:

Listing 14: Именувани параметри

```
store('Mr. Brainsample', 10, 20, 13, 5)
store(patient = 'Mr. Brainsample', hour = 10,
    minute = 20, day = 13, month = 5)
```

Именувани параметри

Listing 15: Употреба на именувани параметри

```
1 def bmi(weight = 80, height = 1.75):
2 print(weight/(height*height))
3 bmi()
4 bmi(weight = 90)
5 bmi(height = 1.87)
6 bmi (height = 1.90, weight = 88)
7 bmi (weight = 88, height = 1.90)
```

Неопределен брой параметри

В много случаи е удобно да може да се дефинира функция, която да приема неопределен брой параметри. Например може да искаме функция sum, която изчислява сбора на всички параметри, които е получила, или avg, която намира средното им аритметично, или concat, която конкатенира всички символни низове, които ѝ се подадат.

За да се дефинира функция с променлив брой параметри, пред параметъра се поставя звезда и всички стойности се получават в кортеж.

Неопределен брой параметри

Listing 16: Неопределен брой параметри на функция

```
1 def address(name, *titles):
2     print(titles, name)
3 address('Petrov','prof.','d-r','hadji')
```

Неопределен брой параметри

Ако вместо една звезда бъдат сложени две при дефиниране на функцията, то параметрите ще бъдат получени в речник, а не в кортеж. Тъй като речникът изисква двойки ключ-стойност, при извикването на функцията подаването на параметрите трябва да бъде именувано.

Listing 17: Неопределен брой параметри в речник

```
1 def print_params_3(**params):
2     print(params)
3 print_params_3(x=1, y=2, z=3)
```

Неопределен брой параметри

Именуваните параметри, неопределеният брой параметри като кортеж и като речник могат да се комбинират. Въпреки че е допустимо, това може да доведе до проблеми с четимостта на кода и не е препоръчително да се използва, ако нямаме основателна причина за това.

Listing 18: Различни видове неопределен брой параметри

```
1 def print_params_4(x, y, z=3, *pospar, **keypar):
2     print(x, y, z)
3     print(pospar)
4     print(keypar)
5 print_params_4(1,2, 3, 5, 6, 7, foo=1, bar=2)
```

Подредба на аргументите

Въпреки че различните видове параметри могат да се смесват, това не може да става в произволен ред. Правилото за подредбата е следното:

1. позиционни параметри
2. неопределен брой параметри в списък *args
3. именувани параметри
4. неопределен брой параметри в речник **kwargs

Разпакетиране на списъка с аргументи

Има случаи, в които трябва да подадем на функция параметри, чийто брой не знаем или функцията очаква отделни позиционни параметри. Например имаме универсална функция за тестване, която получава списък с аргументи и референция към функцията, която трябва да тества. Тя трябва да подаде всеки един от елементите на списъка на отделна позиция. Можем да направим това, като разпакетираме списъка с аргументи при подаването му, поставяйки отпред звезда. Разликата от променливия брой параметри е, че тук пишем звезда при извикването, а не при дефинирането.

Разпакетиране на списъка с аргументи

Listing 19: Разпакетиране на аргументи

```
1  def checkImportedClass(filename, init_args,
       method_args, method_name, class_name):
2      ImportModule = importlib.import_module(
           filename, package=None)
3      TestClass = getattr(ImportModule, class_name)
4      instance = TestClass(*init_args)
5      method = getattr(TestClass, method_name)
6      if method_args:
7          result = method(instance,*method_args)
8      else:
9          result = method(instance)
10     print(result)
```

Разпакетиране на списъка с аргументи

В този пример имаме функция, която получава име на модул и го внася в програмата по име (благодарение на importlib модула). На ред 3 получаваме референция към клас, чието име също е получено като аргумент. Аргументите, които се подават на конструктора, нашата функция получава като списък. Именно него трябва да разпакетираме, което се случва на ред 4. Същото става и на ред 7, за да се изпълни метода на класа, който искаме да тестваме.

Декларирането на променливи прилича на задаване на стойност на ключ в речник, само че речникът е невидим. Това всъщност не е далеч от истината, името на скрития речник е vars.

Listing 20: Речник vars

```
1 x = 1
2 initials="I.F."
3 print(vars())
4 {'__builtins__': <module '__builtin__' (built-in)
      >, '__file__':
5 'example39.py', '__package__': None, 'x': 1, '
      __name__':
6 '__main__', '__doc__': None, 'initials': 'I.F.'}
```

Аналогично на него има и речник globals, който съдържа двойки име-стойност на всички глобални променливи. Това е полезно, когато се сблъскаме с проблема за засенчената глобална променлива. Когато имаме глобална и локална променлива с едно и също име, глобалната не може да бъде достъпена директно.

Listing 21: Засенчване на глобални променливи

```
1 def shadow_var():
2     name = "Petrov"
3     #Here we have a problem if we want to access
          the global "name"
4 name = "Ivanov"
5 shadow_var()
```

Може обаче да се извърши обръщение към globals, за да се достъпи стойността на глобалната променлива.

Listing 22: Използване на globals за достъп до засенчена глобална променлива

```
1 def shadow_var():
2     name = "Petrov"
3     print(name)
4     print(globals()['name'])

6 name = "Ivanov"
7 shadow_var()
```

Видимост и обхват

Всяка променлива има обхват, т.е. участък от кода, в който се вижда и може да бъде променяна. Глобалните променливи са такива, които са достъпни от всички части на програмата. Те се различават от локалните променливи, които са достъпни само за функцията, в която са дефинирани. Обикновено глобалните се дефинират в „основната програма“, а локалните – в съответната функция. В Python присвояванията винаги засягат най-вътрешния обхват, освен ако не е явно указано друго с изразите global и nonlocal.

Видимост и обхват

С ключовата дума global указваме, че дадена променлива съществува в глобалния обхват на видимост и трябва да бъде обвързана повторно там. Аналогично nonlocal указва, че дадена променлива има за обхват на видимост обхващащата функция. С помощта на global можем да правим две неща - да създаваме глобална променлива в тялото на функция, или ако глобална променлива с такова име съществува - да можем да ѝ присвоим нова стойност.

Listing 23: Дефиниране на глобална променлива във функция

```
1  x = 1
2  y = 100
3
4  def change_global():
5      global x
6      x = x + 1
7      y = 100
8
9  change_global()
10 print(x)
11 print(y)
```

Вложени функции

Python позволява влагането на функции, т.е. декларирането на една функция в друга. Смисълът от това е, че понякога имаме нужда от група действия само в рамките на една функция, но те се повтарят много пъти в нея. Тогава няма нужда да извеждаме тази група действия във функция, която е с глобална видимост. Вместо това, можем да я вградим в тази, която ще ги използва. Например при сортирането на Хоор, процедурата по създаване на дялове има смисъл само в контекста на алгоритъма и като такава може да бъде декларирана като вградена във функцията за сортиране.
Вложената функция има достъп до променливите, декларирани в обхвата на външната.

Вложени функции

Listing 24: Вложена функция

```
1 def outer():
2     def inner():
3         print("This is the inner function")
4
5     print("This is the outer function")
6     inner()
7
8 outer()
```

Listing 25: Видимост във вложена функция

```
1  import math

3  def quadratic():
4      a, b, c = 2, 5, 3

6      def D():
7          return b*b-4*a*c

9      return ((-b+math.sqrt(D()))/(2*a),(-b-math.
           sqrt(D()))/(2*a))

11 print(quadratic())
```

Трябва да имаме предвид обаче, че вложената функция може да чете, но не и да изменя стойността на променливите от външната функция.

Listing 26: Още за видимост във вложена функция

```
1 def outer():
2     a = True
3     def inner():
4         a = False
5
6     print(a)
7
8 outer()
```

Рекурсия

Рекурсията е мощен механизъм, почиващ на идеята, че една функция може да извиква сама себе си.
Как обаче работи? Когато се извика една функция, за нея се заделя място, обикновено в стековата памет. Там се пазят параметрите и локалните променливи на всяка „жива“, т.е. започнала и неприключила изпълнението си функция. Когато функция извика сама себе си, това означава, че в паметта се заделя място за ново копие на функцията, тя има два екземпляра със свои собствени стойности на променливите. Разбира се, рекурсията трябва да има дъно, което означава, че извикването трябва да е в условен оператор и това условие да се променя. Така има вариант, в който функцията ще спре да създава свои копия и ще започне процесът на спиране и премахване от паметта на копията на функцията.

Listing 27: Рекурсия

```
1 def factorial(n):
2     if n == 1:
3         return 1
4     else:
5         return n * factorial(n-1)
```

Ламбда изрази

В последите години в програмните езици започнаха да навлизат елементи от функционалното програмиране. Един такъв аспект са анонимните функции, известни като ламбда изрази. В Python те са малки анонимни функции, започващи с ключовата дума lambda. Те са единственият вид функции, вложени в друга функция. Тъй като всеки ламбда израз е стойност, с него може да се прави всичко, което и с всяка друга стойност: да се присвои на променлива, да бъде част от съставна стойност, да бъде аргумент при обръщение към функция и дори да бъде върнат като резултат от изпълнението на функция. Ограничението на ламбда изразите в Python е, че тялото на такава функция е израз, а не команда или поредици от команди.

Функцията map

Функцията map образува редица чрез прилагане на дадена функция върху всеки от членовете на дадена редица. Например, ако имаме списък от числа и искаме да съставим друг списък, в който числата са удвоени:

Listing 28: Функция map

```
1 numbers = [72, 101, 108, 108, 111, 44, 32, 119,
      111, 114,
2 108, 100, 33]
3 n2 = map(lambda n: 2*n, numbers)
4 for element in n2:
5     print(element)
```

Функцията filter

Функцията filter очаква като аргументи булева функция и списък и образува нов списък от онези елементи на получения, за които функцията има стойност True.

Listing 29: Функция filter

```
1 numbers=[72, 101, 108, 108, 111, 44, 32, 119,
      111, 114,
2 108, 100, 33]
3 n2 = filter(lambda n: n % 2 == 0, numbers)
4 for element in n2:
5     print(element)
```

Ламбда изрази

Използването на ламбда изрази е особено полезно, когато искаме да сортираме сложна структура по даден елемент или набор от обекти по даден атрибут. Тогава използваме ламбда израза, за да укажем кое връща стойността, която е ключ на сортирането.

Listing 30: Използване на ламбда израз при сортиране

```
1 student_tuples = [
2 ('John', 21),
3 ('Jane', 22),
4 ('Dave', 25),
5 ]
6 sorted(student_tuples, key=lambda student:
      student[1])
7 # sort by age
8 [('John', 21), ('Jane', 22), ('Dave', 25)]
```

Пример

Нека направим програма, която извежда всички файлове в дадена директория и всички директории в нея и т.н. Това е най-лесно да се реши рекурсивно. Нашата функция ще обхожда всички файлове в текущата директория заедно с пълния им път и когато един от тях е директория, ще се извиква рекурсивно с нейния път.

Listing 31: Рекурсивно обхождане на под-директории

```
1 import os
2 def getFileList(path):
3     for filename in os.listdir(path):
4         print(path + "/" + filename)
5         if os.path.isdir(path + "/" + filename):
6             getFileList(path + "/" + filename)
7 getFileList('/home/user')
```

Пример

Лабиринтите винаги са били интересни. Ще напишем програма, която създава лабиринт с дадени размери, реализирайки алгоритъм за търсене в дълбочина. Нашият лабиринт ще бъде представим в двумерна структура (списък от списъци), в който стените ще означаваме с 1, а свободните позиции с 0. Преди да започне същинското генериране, ще инициализираме полето, като го оградим със стени и вътре в тях имаме редуващи се стени и свободни клетки.

Пример

Алгоритъмът започва с произволно избрана клетка, маркира я като посетена (ред 11) и съставя списък на съседите ѝ (ред 12). За всеки съсед, започвайки с произволно избран, се прави следното: ако той не е вече посетен, премахва стената между текущата клетка и този съсед (ред 17), след което извиква рекурсивно процедурата за този съсед като текуща клетка. Обхождането в дълбочина означава, че започвайки от първата клетка, посещаваме един от нейните съседи, след това един от съседите на нейния съсед и т. н. докато стигне до задънена улица, т.е. клетка, на която всички съседи са посетени. Тогава се връща до последната посетена клетка, която има непосетен съсед и продължава оттам.

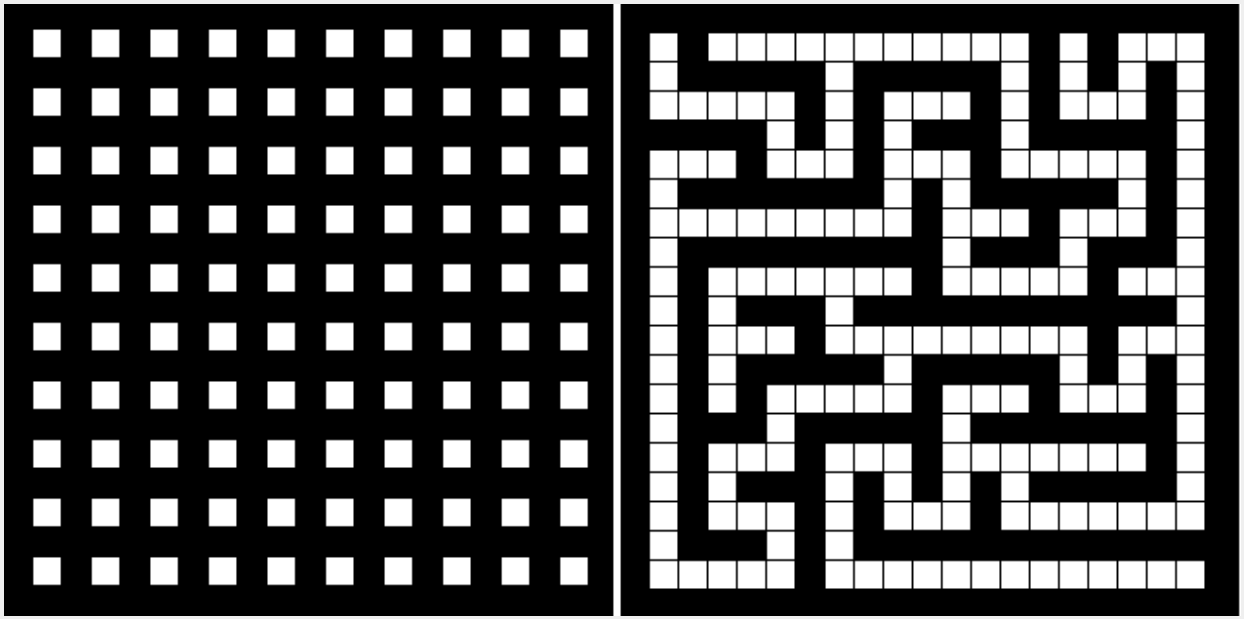
Listing 32: Генериране на лабиринт

```
1  import random
2
3  def init_maze(l,w):
4      maze = [[1] * l for i in range(w)]
5      for i in range(1,w-1,2):
6          for j in range(1,l-1,2):
7              maze[i][j]=0
8      return maze
```

Listing 33: Генериране на лабиринт

```
9  def walk(maze, x, y):
10   maze[x][y] = 3
11   neighbors = [[x - 2, y],[x + 2, y],[x, y - 2],[
         x, y + 2]]
12   random.shuffle(neighbors)
13   for newx,newy in neighbors:
14     if (newx in range(1,len(maze)) and newy in
           range(1, len(maze[0]))):
15       if maze[newx][newy] == 0:
16         maze[(newx + x) / 2][(newy + y) / 2] = 3
17         walk(maze, newx, newy)
```

На фигурата по-долу можем да видим първоначално инициализираното поле и крайния резултат (изчертани с pyGame).



Задачи

Да се създаде функция за изчисляване на първите n числа на Фибоначи, и тяхното съхранение в подходяща структура, така че когато се стигне до дадена позиция от редицата, първо да се проверява дали стойността вече не е изчислявана и ако е да се вземе от структурата, а да не се изчислява повторно.

Задачи

Да се създаде рекурсивна функция, която при зададено число и степен, връща числото, повдигнато на степента.

Задачи

Двоичното търсене в редица е измамно проста задача. Въпреки че алгоритъмът не е сложен, той много лесно може да бъде сбъркан. По своята същност той е рекурсивен и макар че той съвсем лесно може да бъде написан итеративно, рекурсивната му реализация е добро упражнение.

Алгоритъм:

Дадени са сортиран списък от цели числа, търсен елемент, лява и дясна граница на търсенето. Търси се индекс на търсения елемент или индикатор, че той не присъства в списъка.

Задачи

1. Намираме централния елемент
2. ако той е търсеният, връщаме индекса му
3. ако той не е търсеният
4. ако лявата граница е по-малка от дясната има 2 варианта
   - 4.1 централният елемент е по-голям от търсения => преместваме дясната граница на индекс централният елемент -1 и викаме рекурсивно функцията
   - 4.2 централният елемент е по-малък от търсения => преместваме лявата граница на индекс централният елемент +1 и викаме рекурсивно функцията
5. ако лявата граница не е по-малка от дясната търсеният елемент не се намира в масива

Задачи

Да се напише рекурсивна функция, която връща дали даден символен низ е палиндром (дали низът и обърнатият символен низ са същите).

Благодаря за вниманието!

Въпроси?

References