# Изключения и тестване

д-р Филип Андонов

21 юни 2022 г.

# Съдържание

# Изключения

В хода на изпълнение на програмата понякога възникват събития, които не са предвидени. Някои от тях са грешки, други просто неочаквани събития. Изключителните събития могат да се обработват с if клаузи, но това е трудоемко и прави кода тромав. Python предлага като решение на този проблем изключителни обекти. Ако такъв обект не бъде обработен по никакъв начин, програмата прекратява действието си със съобщение за грешка.

# Изключения

Listing 1: Програмен код

```
>>> 1/0
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ZeroDivisionError: integer division or modulo by
    zero
```

# Изключения

Ако това беше всичко, което правят изключенията, то те щяха да бъдат изключително безинтересни. Изключенията са инстанции на някакъв клас и тези инстанции се прихващат по различни начини, даващи възможности за обработка на грешката вместо цялата програма да прекрати действието си.

# Пораждане на изключения

За да се породи изключение, се използва израза raise с аргумент или клас или екземпляр. Ако аргументът е клас, екземпляр се създава автоматично. Опционно може да се предаде и аргумент символен низ след класа, поставен в скоби.

# Пораждане на изключения

Listing 2: Програмен код

```
>>> raise Exception
Traceback (most recent call last):
File "<stdin>", line 1, in ?
Exception
>>> raise Exception('user IQ too low')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
Exception: user IQ too low
```

# Вградени изключения

Python съдържа множество от вградени изключения, които се пораждат при възникването на изключителни обстоятелства при употреба на вградени функции. Можем да видим всички вградени изключения с тяхното описание със следния код.

Listing 3: Програмен код

```
exceptions = []
temp = list([Exception])
while temp:
    exception = temp.pop()
    exceptions.append(exception)
    temp.extend(exception.__subclasses__())

for e in exceptions:
    print(e.__name__)
    print(e.__doc__)
    print("="*20)
```

# Вградени изключения

Ето и някои по-важни от тях:

'AttributeError' – когато референция към атрибут или присвояване се провалят

'Exception' – Базов клас за всички изключения

'IOError' входно-изходна грешка, например опит за отваряне на несъществуващ файл

'KeyError' – използване на несъществуващ ключ при съответствие

'IndexError' – обръщение към несъществуващ елемент на редица

'SystemError' – лошо формиран код

'TypeError' – вградена операция или функция е приложена върху обект от грешен тип

'ValueError' – вградена операция е приложена върху обект от правилен тип, но с грешна стойност

'ZeroDivisionError' – деление на нула

# Собствени изключения

Когато искаме да обработваме определени типове изключения на база техния клас с код, който е специфичен за това изключение, е необходимо да създадем своя клас изключение. Това се прави лесно – просто създаваме клас, който обаче директно или индиректно е наследник на Exception.

Listing 4: Собствено изключение

```
1 class SomeCustomException(Exception): pass
```

# Прихващане на изключения

Смисълът на изключенията е, че за разлика от грешките те могат да се обработват. Това означава, че при пораждането на изключение то няма да прекъсне изпълнението на програмата, ако има код, който да го прихваща и обработва. Нека разгледаме следния невинен на пръв поглед код.

Listing 5: Изключение без прихващане

```
1 x = input('Enter the first number: ')
2 y = input('Enter the second number: ')
3 print(int(x)/int(y))
```

Когато потребителят въведе 0 за делител се поражда изключение за деленние на нула.

# Прихващане на изключения

Ще прихванем и обработим това изключение така:

Listing 6: Прихващане на изключение

```
1 x = input('Enter the first number: ')
2 y = input('Enter the second number: ')
3 try:
4       print(int(x)/int(y))
5 except ZeroDivisionError:
6       print("you can not divide by zero!")
```

# Прихващане на изключения

Ако искаме да прихванем изключение, но след това да го породим отново, можем да извикаме raise без аргументи.

Listing 7: Прихващане и пораждане отново

```
1 try:
2         holdConversation(user)
3 except UserIQTooLow:
4     if user != boss:
5         print('User IQ too low')
6     else:
7         raise
```

# Обработка на повече от едно изключение

В рамките на един блок могат да се породят различен вид изключения. Допустимо е с един try – except блок да прихванем повече от едно изключения. В програма 6 кодът се изпълнява в защитен блок, но единственото изключение, което се прихваща и обработва е деление на нула. Ако потребителят въведе стойност, която не е число, се поражда ValueError на ред 4. Можем да решим този проблем с дадения по-долу код.

# Обработка на повече от едно изключение

Listing 8: Обработка на повече изключения

```
x = input('Enter the first number: ')
y = input('Enter the second number: ')
try:
    print(int(x) / int(y))
except ZeroDivisionError:
    print("you can not divide by zero!")
except ValueError:
    print("Numbers can be divided only by
        numbers")
```

# Обработка на две изключения с един блок

Понякога има смисъл да имаме само един блок за обработка на различни видове изключения.

Listing 9: Две изключения с един блок

```
x = input('Enter the first number: ')
y = input('Enter the second number: ')
try:
    print(int(x) / int(y))
except (ZeroDivisionError, ValueError) as e:
    print("Error:", e)
```

# Обработка на две изключения с един блок

Ако искаме да прихващаме всички възникнали изключения, можем да извикаме except без да описваме изключения:

Listing 10: Всички изключения

```
1 x = input('Enter the first number: ')
2 y = input('Enter the second number: ')
3 try:
4         print(x/y)
5 except:
6         print("Something went wrong...")
```

# Обработка на две изключения с един блок

Горното не е толкова добра идея, защото не можем да разберем естеството на грешката. По-добрият вариант е:

Listing 11: По-добър вариант на всички изключения

```
1 x = input('Enter the first number: ')
2 y = input('Enter the second number: ')
3 try:
4         print(repr(x)/repr(y))
5 except Exception as e:
6         print(e)
```

# Обработка на две изключения с един блок

Можем да комбинираме прихващане на определени изключения с прихващане на неизвестни:

Listing 12: Прихващане на определени и неизвестни изключения

```
try:
        x = input('Enter the first number: ')
        y = input('Enter the second number: ')
        print(int(x) / int(y))
except ZeroDivisionError as e:
    print("Stop dividing by zero!!!")
except:
    print("something went wrong")
```

# Ако всичко е наре

С клауза else можем да създадем блок, който да се изпълнява, ако не е възникнало изключение. В следния пример ще изпълняваме цикъла докато потребителят въвежда стойности пораждащи изключение и ще извикваме break когато не се породи изключение.

Listing 13: Ако не е възникнало изключение

```
1  while 1:
2      try:
3          x = input('Enter the first number: ')
4          y = input('Enter the second number: ')
5          value = (int(x) / int(y))
6          print('x/y is', value)
7      except:
8          print('Invalid input. Please try again.')
9      else:
10         break
```

# try...finally

За да направим приложенията си по-устойчиви е добре да имаме механизъм, който гарантира, че даден код ще се изпълни независимо дали възникне изключителна ситуация. Типичен пример за това е заемането на даден ресурс. Искаме той да бъде освободен независимо дали използването му е било успешно или се е случило нещо неочаквано. Ако например четем файл и искаме да го затворим независимо дали се е породило изключения, можем да използваме блок try...finally. Finally частта ще се изпълни независимо дали се е породило изключение или не.

# try...except...finally

В ранните версии на Python try се комбинира или с finally или с except, но двете не могат да бъдат част от един try блок. В съвременните версии това е допустимо, поради което можем да напишем код от типа:

```
try:
    блок 1
except Exception1:
    блок 2
except Exception2:
    блок 3
else:
    блок 4
finally:
    блок 5
```

# Разпространение на изключения

Ако изключението се породи във функция, и то не се обработи в нея, изключението изплува нагоре към мястото, на което функцията е извикана. Ако и там не се обработи, продължава нагоре докато стигне до главната програма и ако там няма обработка на изключенията, програмата прекратява действието си с информация за това какво се е объркало (stack trace).

Listing 14: Разпространение на изключения

```
1 def makeError():
2     raise Exception("BAM!")
3 def callFaulty():
4     makeError()
5 callFaulty()
```

# Разпространение на изключения

Stack trace е стекът, проследяващ породилите изключението извиквания. Това е удобен инструмент при отстраняване на грешки в кода.

```
Traceback (most recent call last):
  File "exceptions_propagation.py", line 5, in <module>
    callFaulty()
  File "exceptions_propagation.py", line 4, in callFaulty
    makeError()
  File "exceptions_propagation.py", line 2, in makeError
    raise Exception("BAM!")
Exception: BAM!
```

# Тестване

Класическият подход при програмиране се нарича „пиши малко, тествай малко“. При т. нар. test-driven подход, първо се прави тестът и след това се пише кодът. Използват се т. нар. unit тестове. Те се основават на тестови случаи, които са най-малкия градивен блок от тестваем код. В идеалния случай ще тестваме целия си код с всички възможни входове и изходи, но това на практика е невъзможно.

Когато използваме PyUnit, тестов случай е просто обект с поне един тестови метод, който изпълнява код и след това проверява резултата с множество стойности, които знаем, че трябва да се получат. Името на пакета се нарича PyUnit, но модулът, който трябва да включим в кода си се нарича unittest. Всеки тестов вариант (test case) е наследник на класа TestCase. Най-простите тестови варианти пренаписват метода runTest. В него пишем кода, който искаме да тестваме.

# Методи

```
1 import unittest
2 class BasicMathTest (unittest.TestCase):
3     def runTest (self):
4         self.failUnless (2 + 2 == 2 * 2, 'two plus
              two and two by two not the same!')
5         self.failIf (2 + 2 != 2 * 2, 'two plus two
              and two by two are different')
6         self.failUnlessEqual (2 + 2, 2 * 2, 'two
              plus two and two by two should be equal'
              )
```

# Методи

```
7  def suite():
8      suite = unittest.TestSuite()
9      suite.addTest(BasicMathTest())
10     return suite
11 if __name__ == '__main__':
12     runner = unittest.TextTestRunner()
13     test_suite = suite()
14     runner.run(test_suite)
```

На ред 1 внасяме модула unittest. На следващия ред създаваме клас BasicMathTest, който наследява класа Testcase от модула unittest. Нашият клас има само един метод – runTest, който изпълнява същинското тестване. Добра идея е този метод да има собствен docstring. На ред 8 създаваме тестов пакет, който е екземпляр на класа TestSuite. Към него се добавят тестове с метода addTest (ред 9). За да се изпълнят всичките тестове от пакета, е необходимо да се създаде runner клас (ред 12) и пакетът да се подаде като аргумент на неговия метод run (ред 14). Разбира се всички показани в примера тестове трябва да минат успешно, но в реална среда ще тестваме не дали хардуерът изчислява правилно, а наш код, за който не сме толкова сигурни в правилността му.

Методите на TestCase започват с fail или assert. Следва списък с наличните методи. Трябва да имаме предвид, че те в известна степен са взаимозаменяеми и е въпрос на предпочитания с кой метод ще тестваме някакво условие.

assertTrue(expr[, msg]))
Тестът се проваля, ако изразът е False, като извежда и незадължително съобщение.
failUnless(expr[, msg])
Същото като assertTrue.
assertEqual(x, y[, msg])
Тестът се проваля, ако двете стойности са различни, печатайки двете стойности в дневника за трасиране.
failUnlessEqual(x, y[, msg])
Същото като assertEqual.
assertNotEqual(x, y[, msg])
Обратното на assertEqual.

assertAlmostEqual(x, y[, places[, msg]])
Подобно на assertEqual, но с толеранс за числа с плаваща запетая.
failUnlessAlmostEqual(x, y[, places[, msg]])
Същото като assertAlmostEqual.
assertNotAlmostEqual(x, y[, places[, msg]])
Обратното на assertAlmostEqual.
failIfAlmostEqual(x, y[, msg])
Същото като assertNotAlmostEqual.

assertRaises(exc, callable, ...)
Тестът се проваля, освен ако callable не предизвика exc, когато бъде извикан (с незадължителни аргументи).
failUnlessRaises(exc, callable, ...)
Същото като assertRaises.
failIf(expr[, msg])
Обратното на assertTrue.
fail([msg])
Безусловно проваляне с незадължително съобщение

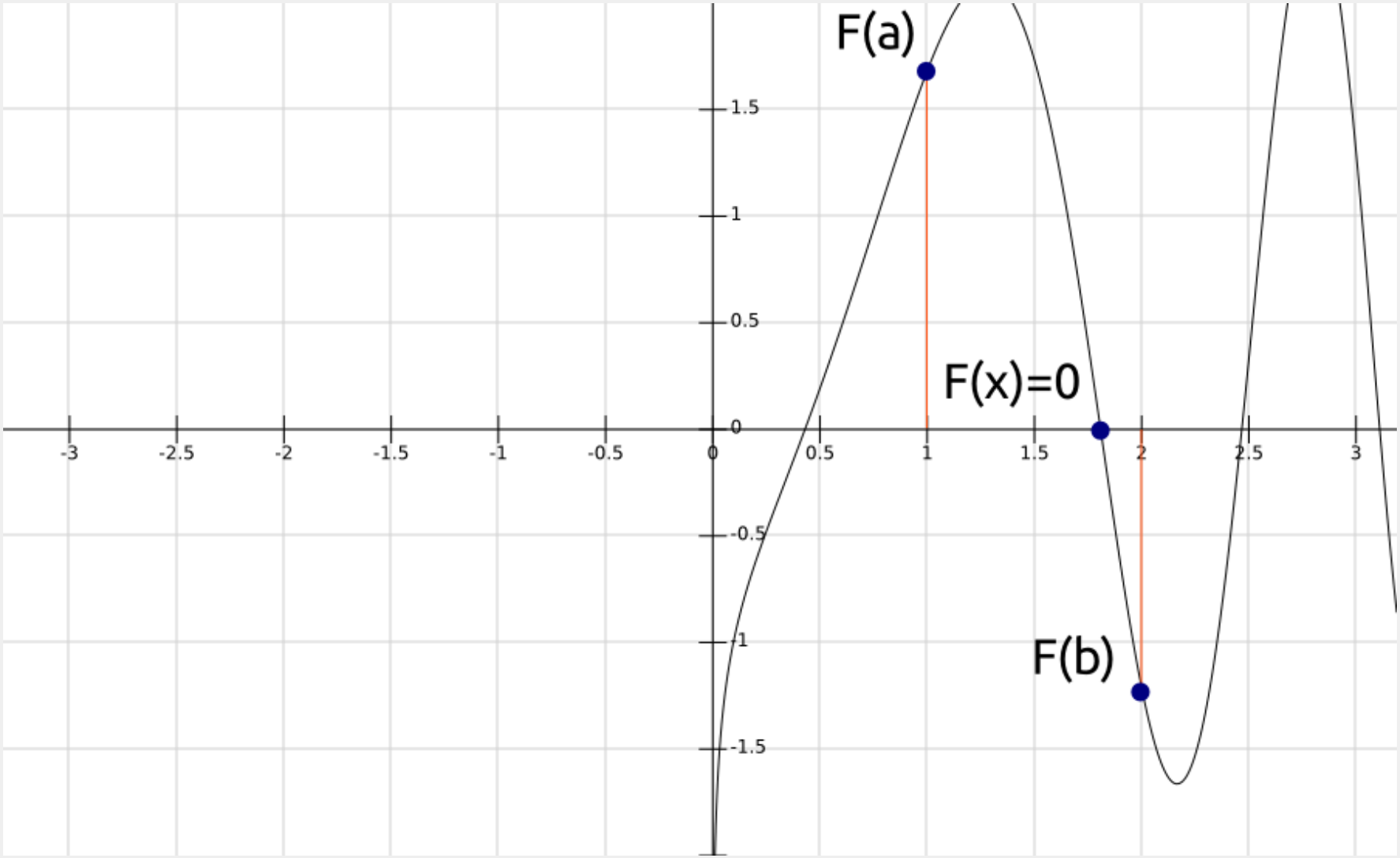
# Задача

Да се реализира т. нар. метод на дихотомичното търсене на корен на уравнение. На английски се нарича bisection method. Методът многократно разделя на две интервала и избира за следваща обработка този под-интервал, в който трябва да се намира корена. Методът е много елементарен и надежден, но е сравнително бавен. Поради това, той често се използва за намиране на грубо приближение до решението, което после се използва от някой по-бързо сходим метод.

# Задача

Методът още се нарича „метод на двоично търсене“ (binary search), тъй като е подобен на двоичното търсене в нареден масив, например. Приложим е когато искаме да решим уравнението $(f(x) = 0)$, където $f$ е непрекъсната функция в интервала $[a, b]$ и $f(a)$ и $f(b)$ имат противоположни знаци. В такъв случай в интервала $[a, b]$ със сигурност има корен, тъй като $f$ трябва поне на едно място да пресича оста $x$.

# Изображение



# Задача

На всяка стъпка на метода интервалът се разделя на две като се изчислява точката в средата на интервала ($c = (a + b)/2$) и стойноста на функцията $f(c)$ в тази точка. Ако самото с не е корен (много малко вероятно, но не е невъзможно), то имаме две възможности: или $f(a)$ и $f(c)$ имат противоположни знаци и обграждат корен, или $f(c)$ и $f(b)$ имат противоположни знаци и обграждат корен. Методът избира подинтервал, който обгражда/съдържа корен за нов интервал, който да използва на следващата итерация. По този начин интервалът, който съдържа корен, се намалява наполовина на всяка стъпка. Процесът продължава, докато интервалът не стане достатъчно малък.

# Задача

И в двата случая новото $f(a)$ и $f(b)$ имат противоположни знаци, така че методът е приложим за новия по-малък интервал. Накратко, ако $f(a)$ и $f(c)$ са с противоположни знаци, избираме $c$ да бъде новото $b$, ако $f(b)$ и $f(c)$ имат противоположни знаци, то избираме с да бъде новото a. Ако $f(c) = 0$, то $c$ е решение и процесът спира. Методът е гарантирано сходим към корен на $f$ ако $f$ е непрекъсната в интервала $[a, b]$ и $f(a)$ и $f(b)$ имат противоположни знаци. $(f(x) = x^3 + 3x - 5)$, където $[a = 1, b = 2]$ и точността = 0.001.

Таблица: Табулиране на процеса

| i | a | x | b | f(a) | f(x) | f(b) |
|---|---|---|---|---|---|---|
| 1 | 1 | 1.5 | 2 | -1 | 2.875 | |
| 2 | 1 | 1.25 | 1.5 | -1 | 0.703125 | 9 |
| 3 | 1 | 1.125 | 1.25 | -1 | –0.201171 | 2.875 |
| 4 | 1.125 | 1.875 | 1.25 | 0.201171 | 0.237060 | 0.703125 |
| 5 | 1.125 | 1.5625 | 1.1875 | 0.201171 | 0.014556 | 0.703125 |
| 6 | 1.125 | 1.140625 | 1.15625 | 0.201171 | -0.094142 | 0.237060 |
| 7 | 1.140625 | 1.148437 | 1.15625 | 0.094142 | -0.040003 | 0.014556 |
| 8 | 1.148437 | 1.152343 | 1.15625 | 0.04003 | -0.012775 | 0.014556 |
| 9 | 1.152343 | 1.154296 | 1.15625 | 0.012775 | 0.000877 | 0.014556 |

# Задача

Да се напише програма, която намира корен на уравнения по метода на дихотомията. Функцията $f(x)$ да е изнесена в програмна конструкция функция за лесно промяна. Числата $a$ и $b$, които задават интервала на търсене, да се въвеждат от клавиатурата. Числата да се четат като символен низ и след това да се конвертират до число. Ако въведените от потребителя стойности не са числа, както и ако $f(a)$ и $f(b)$ имат един и същи знак, да се пораждат съответни изключения.
Да се намери корен на уравнението $exp(x) - 2x - 2 = 0$
Да се намери корен на уравнението $x3 + 3x - 5 = 0$

Благодаря за вниманието!

Въпроси?

# References