Основи на Програмирането

Лекция 10

Модули. Пакети. Обработка на грешките.

Лекция 10

Какво ще научите

- ✓ Програми на Python и модули
- ✓ Зареждане и изпълнение на модули
- ✓ Помощна информация за модули
- ✓ Синтаксис на командите за работа с модули
- ✓ Пакети
- ✓ Инсталиране на пакети
- ✓ Какво представляват грешките и изключенията
- ✓ Йерархия на класовете изключения
- ✓ Обработка на изключения
- ✓ Предизвикване на изключения
- ✓ Как да задавате свои собствени изключения

Как съставяме програми

- Изучихме основните конструкции в езика
- Знаем как да създаваме програма за даден алгоритъм и да я запазваме във файл
- Всеки файл включва множество от команди и изрази
- Всеки файл има собствено пространство от обекти и свойства

Програми на Python

- Програмите и модулите в езика Python се различават само по начина на използване:
  - .py файловете, които се изпълняват от интерпретатора, са програми (често наричани скриптове)
  - .py файлове, към които има обръщение с командата **import,**са модули
- Така един и същи файл може да бъде както програма (скрипт), така и модул

Роля на модулите

- **Многократно използване на команди**
  - Всяка функция може да бъде извиквана многократно по време на изпълнение
  - Функциите могат да бъдат извикани от различни програми
- **Управление областите на имената**
  - Групиране заедно данните с функциите които ги използват
- **Споделени услуги върху набор данни**
  - Глобална структура от данни достъпна от различни функции и програми

Използване на модулите

- **Модулът, който се изпълни пръв (файлът който пръв се зареди в системата) се нарича главен (main)**
- **След това той може да зарежда други модули или от файлове, или от стандартната библиотека**
- **Всеки модул предоставя своите обекти като атрибути:**
- **<име_модул>.<име_атрибут>**
  - Тези атрибути най-често са функции, но могат да са произволни обекти

Зареждане на модул

При първо зареждане на модул, се изпълняват три отделни важни стъпки:

- **Откриване (намиране) на модула (файлът съдържащ неговите команди и дефиниции)**
- **Компилиране на модула (ако е нужно)**
- **Изпълнение на компилирания код за създаване на обектите, които модула предоставя**

Намиране на модул

Python използва вграден път за търсене на модули който зависи от ОС и може да се допълва/променя. Освен пътя, в командата import се изпуска и типа на файла (използват се разширенията приети в ОС)

- **Кой е текущия път се вижда чрез извеждане на sys.path (след import на модула sys)**
- **Пътят може да се променя чрез промяна стойността на този обект (тази променлива)**

Пътища за намиране на модул

Python използва следната йерархия от пътища:

- **Текуща директория (където се намира стартирания файл, или от където се стартира Python)**
- **Пътища указани в променливата: PYTHONPATH**
- **Път към стандартната библиотека**
- **Пътища указани в .pth файлове (в системни папки)**
- **Път към site-packages (други модули в локални пакети)**

Пътища за намиране на модул

Първият, третия и последния елемент от йерархията от пътища е вграден в системата.

Потребителят може да влияе чрез пътищата указани в променливата: PYTHONPATH, както и чрез .pth файлове (последното не се препоръчва, освен за много опитни професионални програмисти)

Компилиране на модул (ако е нужно)

При първо зареждане на модул, той трябва да е наличен в байт код компилиран формат. Освен текстовия файл с текста на програмата (разширение .py) системата съхранява и компилирана версия (с разширение .pyc)

- **Системата проверява времената на последна промяна на двата варианта за модула, и ако .py не е по-нов от .pyc - зарежда .pyc**
- **Иначе, компилира модула и го запазва в .pyc, и после го зарежда. Всички файлове .pyc се намират в папката __pycache__ в основната за .py папка**

Изпълнение на модул

След зареждане на компилирания модул, байт код командите му се изпълняват последователно една след друга за създаване на обектите, които модула предоставя.

Например, дефинициите на функции def се изпълняват и новосъздадените обекти тип функция се записват като атрибути предоставяни от модула.

Всяка следваща команда import за същия модул в същия процес (програма main) не прави нищо, а се използва каквото вече е налично в паметта.

Вградена функция dir

- ✓ Използва се за получаване на списък от всички свойства и атрибути на даден обект.
- ✓ Когато се задава без аргумент връща списък на всички променливи в текущия контекст
- ✓ Типичен начин на използване – когато като аргумент се зададе име на модул, или име на вграден тип от данни
- ✓ За да се покаже тази информация за някакъв модул, той първо трябва да се зареди с import

Вградена функция dir - пример

```
>>> import sys
>>> import math
>>> [a for a in dir(math) if not a.startswith('_')]
['acos', 'acosh', 'asin', 'asinh', 'atan', 'atan2', 'atanh', 'ceil', 'comb',
'copysign', 'cos', 'cosh', 'degrees', 'dist', 'e', 'erf', 'erfc', 'exp', 'expm1',
'fabs', 'factorial', 'floor', 'fmod', 'frexp', 'fsum', 'gamma', 'gcd', 'hypot',
'inf', 'isclose', 'isfinite', 'isinf', 'isnan', 'isqrt', 'lcm', 'ldexp', 'lgamma',
'log', 'log10', 'log1p', 'log2', 'modf', 'nan', 'nextafter', 'perm', 'pi', 'pow',
'prod', 'radians', 'remainder', 'sin', 'sinh', 'sqrt', 'tan', 'tanh', 'tau', 'trunc',
'ulp']
>>>
```

Вградена функция help

```
>>> help(math)
Help on built-in module math:
NAME      math
DESCRIPTION
    This module provides access to the mathematical
functions defined by the C standard.
FUNCTIONS
...
>>> help(math.floor)
Help on built-in function floor in module math:
floor(x, /)
    Return the floor of x as an Integral.
    This is the largest integer <= x.
```

Модули

- **Групират функции и променливи във файлове**
- **Обектите са достъпни с команди from или import**

```
from module import function
function()

import module
module.function()
```

- **Всеки модул е отделна област на имена**
  - Може да се използва за организация на имена:

```
atom.position = atom.position - molecule.position
```

“import” и “from ... import ...”

```
>>> import math
>>> math.cos(0)
1.0
>>> from math import cos, pi
>>> cos(0)
1.0
>>> from math import *
```

import …

```
import somefile
```

- ✓ *Всичко* от файла с име somefile.py се зарежда, но в собствена област на имената
- ✓ За обръщение към обект от модула, се добавя “somefile.” пред името на обекта:

```
somefile.className.method('abc')
somefile.myFunction(34)
```

from … import *

```
from somefile import *
```

- ✓ *Всичко* от somefile.py се зарежда в текущото пространство на имената
- ✓ Обръщение към обект от модула става с името му
- ✓ *Опасност!* Чрез командата import в този вид лесно можем да изтрием функция или променлива от текущото пространство на имената!
- ✓ Този метод за зареждане не се препоръчва

from … import …

```
from somefile import Name
```

- ✓ Само обектът с име *Name* от модула somefile.py се зарежда
- ✓ След зареждане на обекта *Name*, можем да го използваме само по името му – той вече е част от текущото пространство на имената
- ✓ *Внимание*! Така заредения обект може да изтрие обект със същото име от текущото пространство

Изпълнение на модул

Всяка следваща команда import за същия модул в същия процес (програма main) не прави нищо, а се използва каквото вече е налично в паметта.

Това води до проблем, ако искаме да заредим нова версия на програмата, създадена и вече променена във файла с модула

Решение – с функцията reload

reload

Потребител зарежда модул, променя кода му в текстов редактор и го зарежда отново. Това се случва ако се работи интерактивно или с голяма програма, в която модулите се зареждат периодично.

```
import module # първоначален import
... Използване: module.attributes ...
...
from imp import reload #  зареждане на reload
reload(module) # зареждане на нова версия
```

reload

Зарежда новия код и връща неговото пространство. Променя модула на място в паметта:

- Новият код и обекти се зареждат в и изменят текущото пространство на модула
- Имената се заместват с нови обекти. Например, def командата заменя функцията с нов обект
- При пълен import достъпа е до новите обекти
- При клауза from достъпът остава до старите обекти
- reload се използва само за един модул

Многозначно зареждане

```
>>> # Зареждане на няколко модула
>>> import time, sockets, random
>>> # Зареждане на няколко функции
>>> from math import sin, cos, tan
>>> # Зареждане на няколко обекта (атрибути, константи)
>>> from math import pi, e
>>> print(pi)
3.141592653589793
>>> print(cos(45))
0.5253219888177297
>>> print(time.time())
1482807222.7240417
```

Частни обекти в модул

Всички обекти, чиито имена започват с ‘_’, са частни – те не са достъпни след зареждане на модул с клауза from … import *

Въпреки това, те са достъпни ако използваме клаузата за зареждане на модула: import …

Пример:

```
# в модул mod1
_p = 5
# в модул mod2
From mod1 import *
print(_p)
#  Грешка …
```

Специална променлива __all__

С помощта на променливата __all__ се ограничава кои променливи от модул да са достъпни след from … import *

```
# mymodule.py
__all__ = ['imported_by_star']
imported_by_star = 42
not_imported_by_star = 21
>>> from mymodule import *
>>> imported_by_star
42
>>> not_imported_by_star       # Същото и с mymodule.not_imported_by_star
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
NameError: name 'not_imported_by_star' is not defined
```

Специална променлива __all__

Въпреки това, променливите които не са изрично указани в променливата __all__ също могат да бъдат заредени, но само ако бъдат явно указани:

```
>>> from mymodule import not_imported_by_star
>>> not_imported_by_star
21
```

Препоръки за използване на import (PEP8)

1. Всяка клауза import на отделен ред :

```
from math import sqrt, ceil  # Не се препоръчва
from math import sqrt        # Препоръчва се
from math import ceil
```

2. Всички клаузи import да бъдат в началото подредени в следния ред:

- ✓ От стандартната библиотека
- ✓ От други общоприети библиотеки
- ✓ Локални / специфични за дадена библиотека

3. Да се избягва клауза from module import *

4. Да се избягват относителни имена; да се дават явни.

Пакети

Това са специални обединения от модули, групирани в структури от папки, за съвместно използване.

Използването на папки помага за структуриране на модулите по групи функции и улеснява използването на пакета от даден потребител в зависимост от това какви функции и модули от пакета са му нужни.

При тях се използва зареждане от вида:

```
import Sound.Effects.Echo          #зарежда модул Echo
Sound.Effects.echo.echofunc()      #обръщение към обект
```

Зареждане на пакети

```
import Sound.Effects.Echo          #зарежда модул Echo
Sound.Effects.echo.echofunc()      #обръщение към обект

from Sound.Effects import echo     #зарежда модул Echo
echo.echofunc()                    #обръщение към обект

from Sound.Effects.echo import echofunc   #зар. echofunc
# Улеснява обръщението към обектите от модула Echo
echofunc(input, output, delay=0.7, atten=4)
```

Зареждане на пакети

from <пакет> import <елемент>

<елемент> може да бъде под-пакет, модул, обект

Първо се търси като обект, и само ако не се намери, се търси после като модул или под-пакет

import <елем1.елем2.елем3>

Тук <елем1> и <елем2> са задължително под-пакети

<елем3> може да бъде модул или под-пакет, но не може да бъде обект

Разпознаване на пакети

Една папка се възприема като (под)пакет, ако в нея има специален файл с името: __init__.py

Когато се използва зареждане от вида:

>>> import <елем1.елем2.елем3>

- ✓ <елем1> трябва да бъде под-папка на папка от текущия път (sys.path)
- ✓ <елем1> и <елем2> да съдържат в себе си задължително файл с име __init__.py
- ✓ Ако <елем3> е папка (под-пакет), също трябва да съдържа файл с такова име
- ✓ Ако <елем3> е име на файл с разширение .py се възприема като име на модул

Зареждане на пакети

Когато се използва зареждане от вида:

>>> import <елем1.елем2.елем3>

се зареждат последователно (под)пакетите <елем1> (т.е. се изпълнява нейния файл __init__.py), <елем2> (т.е. неговия файл __init__.py) и накрая или (под)пакета <елем3> (т.е. неговия файл __init__.py), или модула <елем3> (т.е. файла с име <елем3>.py)

Друга разлика между пакет и модул е наличието на променливата __path__ само в дефиницията на пакет, която съдържа списък с един елемент името на папката в която е дефиниран пакета.

Зареждане на пакети

from <пакет> import *

В този случай в папката на последния под-пакет от името зададено с <пакет> се търси файла __init__.py и в него променливата от тип списък __all__ , в която се съдържат имената на модулите (файловете) които да се заредят при * от този под-пакет (папка)

Ако имаме командата:

>>> from Sound.Effects import *

във файла Sounds/Effects/__init__.py можем да имаме:

__all__ = ["echo", "surround", "reverse"]

Зареждане на пакети

from <пакет> import *

Ако __all__ не е дефинирана в __init__.py, ще се заредят само обекти създавани в или зареждани от инициализиращия файл __init__.py (евентуално и заредени преди това с предишни команди import)

Като правило пакетите добавят допълнителни папки за търсене на модули, които не са част от стандартния път за търсене съхраняван в sys.path

Инсталиране на пакети

За инсталиране на пакет, наличен в дистрибуцията, изпълнете в команден режим:

pip install package

За проверка дали pip е наличен:

pip –version

За да се инсталира pip, първо се изтегля от:

https://pypi.org/project/pip/

За разглеждане на налични пакети:

https://pypi.python.org/pypi

Инсталиране на пакети 2

За получаване на списък с наличните пакети:

pip list

Result:

```
Package         Version
-----------------------
mysqlclient 1.3.12
pip             18.1
pymongo     3.6.1
setuptools    39.0.1
```

Инсталиране на pip

Ако pip не е наличен:

python -m ensurepip --default-pip

За инсталиране на последна налична версия:

python -m pip install --upgrade pip

Инсталиране на пакети 3

```
pip install "SomeProject"

pip install "SomeProject==1.4"

pip install "SomeProject>=1,<2"

pip install "SomeProject~=1.4.2"
```

Обработка на изключения

Когато по време на изпълнение на програма възникне грешка (опит за достъп до несъществуващ обект или за извършване на несъществуваща операция към дадени обекти), програмата се прекъсва ненормално, всичко създадено до момента в ОП се губи, и се дава съобщение за мястото и причината на грешката.

В тази лекция се разглежда механизъм и средства за нормална обработка на грешките, така че програмата да не губи всичко създадено до момента, и да завърши по нормален начин.

Грешки и изключения при изпълнение на програми

- Изключението (**exception**) е събитие, което прекратява нормалното изпълнение на програмата.
- Най-често това е грешка, но не винаги.
- Всяка грешка предизвиква изключение.
- При настъпване на изключение, изпълнението на програмата се прекратява, и управлението се предава на програма за обработка на изключения.

Грешки и изключения при изпълнение на програми

- Стандартната програма за обработка на изключения действа на ниво главен модул и нейните съобщения се виждат когато изпълним програма с грешки.
- Всеки програмист също може да предизвиква изключения, както и да създава собствена програма за обработка на изключенията.
- Има огромен списък от грешки предизвикващи изключения. Програмистът може да добавя още.

Примери

```
>>> print(7/0)
Traceback (most recent call last):
  File "<pyshell#122>", line 1, in <module>
    print(7/0)
ZeroDivisionError: division by zero
>>> print([1,2] - [3,4])
Traceback (most recent call last):
  File "<pyshell#123>", line 1, in <module>
    print([1,2] - [3,4])
TypeError: unsupported operand type(s) for -: 'list' and 'list'
```

Примери за изключения

IndexError	индекс на редица (контейнер) който не е валиден
KeyError	липсващ ключ за речник
ZeroDivisionError	деление на 0
ValueError	неподходяща стойност за вградена функция
TypeError	използване на функция/операция върху грешен обект
IOError	входно/изходна грешка

Представяне на изключения

Всички изключения в Python са представени като йерархия от класове и обекти

Някои изключения са под-случай (под-клас) на други.

Програмистът може да си дефинира свои.

Пълни подробности относно текущият списък на класове и обекти изключения може да намерите на адрес: https://docs.python.org/3/library/exceptions.html

Обработка на изключения (Exceptions)

- ✓ **Изключенията** са събития, които могат да променят нормалната последователност на изпълнение на програмата.
- ✓ Те се включват автоматично от грешки или други причини.
- ✓ **try/except** : вградени команди за обработка на изключения в Python
- ✓ **try/finally**: специална клауза за последна обработка след реагирането на всички потенциални грешки
- ✓ **raise**: команда за ръчно включване на изключение
- ✓ **assert**: условно включване на изключение

Използване на изключенията

- **Обработка на грешки**
  Когато Python открие грешка, включва изключение
  Стандартна реакция: спиране на програмата
  Цел: програмата сама да хваща грешките, да ги обработва и да не спира ненормално
- **Сигнализиране настъпване на събитие**
- **Реакция на неочаквани ситуации**
- **Действия по нормално завършване на програма**
- **Реализация на нестандартни методи за управление на програми**

Пример

```
>>> 3 / 0
Traceback (most recent call last):
  File "<pyshell#0>", line 1, in <module>
    3 / 0
ZeroDivisionError: division by zero
>>> def div(x,y):
            try:
                    return x/y
            except ZeroDivisionError:
                    return "Деление на 0"

>>> div(7,3)
2.3333333333333335
>>> div(4,0)
'Деление на 0'
```

try/except/else

```
try:
    <блок команди>              #където очакваме грешки
except <име1>:                  #обработка на <име1>
    < блок команди >
except <име2> as <пром>:    #обработка с променлива
    < блок команди >
except (<name3>,<name4>): #обработка на няколко
    < блок команди >
except:                         #обработка на всички други
     < блок команди >
else:                  # изпълнява се ако няма грешки
      < блок команди >
```

try/except/else

- ✓ В един блок try: трябва да има поне една клауза except:
- ✓ Клауза except без израз (име) се допуска, и тя задължително е последна
- ✓ Клаузата else: не е задължителна
- ✓ Ако в някоя клауза се получи друга грешка, тя не се обработва с тази команда try, а се търси следваща нагоре в йерархията на имената команда try, или ако няма – системната програма за обработка на грешки

try/finally

- ✓ Блокът команди в клаузата **finally** се изпълнява винаги, независимо дали в блока команди на **try** е настъпила грешка (изключение) или не
- ✓ Тази клауза не е задължителна, но ако я има, трябва да е последна за **try**
- ✓ Може да се изпълнява и без клауза except:

```
try:
    <block of statements>
finally:
    <block of statements>
```

- ✓ Смисъл: да се изпълнят команди завършващи по смисъл действията в блока команди на **try**, независимо имало ли е изключение

Пример

```
>>> try:
        print(3/0)
except ZeroDivisionError:
        print('Деление на 0')
else: print("няма грешки")
finally:
        print("и накрая почистваме!")

Деление на 0
и накрая почистваме!
```

raise

- **raise** предизвиква явно някакво изключение

raise <име> #предизвиква изключение с <име>

raise <име> from <име1> #указва име1 като причина за име

raise # предизвиква отново последното

- Като име на изключение се задава или вграден, или дефиниран от потребител клас (в този случай трябва да бъде дефиниран като подклас на вградения клас Exception)

```
>>> class Zero(Exception):
        #  тяло на класа
```

Пример

```
>>> def div(x,y):
        try:
            if y == 0:
                raise Zero()
            else:
                return x/y
        except Zero:
            print('Деление на 0!')

>>> div(6,2)
3.0
>>> div(5,0)
Деление на 0!
>>>
```

assert

✓ **assert** е специална условна форма на командата *raise*

Синтаксис:

✓ assert <проверка> [, <съобщение>]

Това е еквивалентно на

✓ if not <проверка> raise AssertionError()

или на

✓ if not <проверка> raise AssertionError(<съобщение>)

Смисълът е да се провери някакво условие и само ако то не е изпълнено, да се предизвика системното изключение AssertionError

Пример за assert

```
>>> def fun(x,y):
        assert x>0, 'x трябва да да бъде положително'
        assert y<0, 'y трябва да бъде отрицателно'
        return y ** x
>>> fun(-4, 2)
16
>>> fun(-4, -2)
Traceback (most recent call last):
  File "<pyshell#130>", line 1, in <module>
    fun(4, -2)
  File "<pyshell#128>", line 3, in fun
    assert x>0, ‘x трябва да бъде положително'
AssertionError: x трябва да бъде положително
```

Изключенията са обекти

- ✓ Всички видове изключения са дефинирани като класове
- ✓ Съществува йерархия от всички възможни изключения в езика Python
- ✓ Всяко конкретно изключение (грешка) е обект от съответния клас
- ✓ Достъп до всеки обект може да се получи чрез атрибута **as** в дефиницията на клаузата за обработката на конкретното изключение:

except <име> **as** <пром>: #обработка с променлива
    < блок команди >

Разглеждане на изключения

sys.exc_info() - връща за всеки обект-изключение редица (tuple) с три елемента:

- type: от кой клас е обекта - изключение
- value: стойността на обекта - изключение
- traceback: копие от стека за изпълнение с информация за йерархията на областите довели до изключението

Вградени класове изключения

- ✓ **Exception** – суперклас за всички изключения
- ✓ **StandardError** – суперклас за всички вградени изключения
- ✓ **ArithmeticError** – суперклас за всички изключения свързани с аритметични операции
- ✓ **OverflowError** – под-клас дефиниращ конкретен вид изключение

Как да използваме изключенията

- ✓ Да се обработват колкото се може по-близко до потенциалните причини – трябва съмнителни команди да включваме в блок на оператор *try* (вход-изход с файлове, Интернет програмиране, …)
- ✓ Ако проблемът е сериозен, програмата може да се прекрати – обработката се използва за по-ясно коментиране на причините за грешката
- ✓ Подобни действия се планират в *try/finally* (тази клауза е символ на т.нар. почистващи действия)
- ✓ По-добре за сложни функции да има една команда *try* свързана с нейното извикване, отколкото много команди *try* в нейното тяло

Заключение

- ✓ Програми на Python и модули
- ✓ Зареждане и изпълнение на модули
- ✓ Помощна информация за модули
- ✓ Синтаксис на командите за работа с модули
- ✓ Пакети
- ✓ Инсталиране на пакети
- ✓ Какво представляват грешките и изключенията
- ✓ Йерархия на класовете изключения
- ✓ Обработка на изключения
- ✓ Предизвикване на изключения
- ✓ Как да задавате свои собствени изключения