Именные функции, инструкция def

Функция в python - объект, принимающий аргументы и возвращающий значение. Обычно функция определяется с помощью инструкции **def**.

Определим простейшую функцию:

```
def add(x, y):

    return x + y
```

Инструкция **return** говорит, что нужно вернуть значение. В нашем случае функция возвращает сумму x и y.

Теперь мы ее можем вызвать:

```
>>>
```

```
>>> add(1, 10)

11

>>> add('abc', 'def')

'abcdef'
```

Функция может быть любой сложности и возвращать любые объекты (списки, кортежи, и даже функции!):

```
>>>
```

```
>>> def newfunc(n):

...     def myfunc(x):

...         return x + n

...     return myfunc

...

>>> new = newfunc(100)   # new - это функция
```

```
>>> new(200)

300
```

Функция может и не заканчиваться инструкцией return, при этом функция вернет значение None:

```
>>>
```

```
>>> def func():

...     pass

...

>>> print(func())

None
```

Аргументы функции

Функция может принимать произвольное количество аргументов или не принимать их вовсе. Также распространены функции с произвольным числом аргументов, функции с позиционными и именованными аргументами, обязательными и необязательными.

```
>>>
```

```
>>> def func(a, b, c=2): # c - необязательный аргумент

...     return a + b + c

...

>>> func(1, 2)   # a = 1, b = 2, c = 2 (по умолчанию)

5

>>> func(1, 2, 3)   # a = 1, b = 2, c = 3

6
```

```
>>> func(a=1, b=3)   # a = 1, b = 3, c = 2

6

>>> func(a=3, c=6)   # a = 3, c = 6, b не определен

Traceback (most recent call last):

  File "", line 1, in

    func(a=3, c=6)

TypeError: func() takes at least 2 arguments (2 given)
```

Функция также может принимать переменное количество позиционных аргументов, тогда перед именем ставится *:

```
>>>
```

```
>>> def func(*args):

...     return args

...

>>> func(1, 2, 3, 'abc')

(1, 2, 3, 'abc')

>>> func()

()

>>> func(1)

(1,)
```

Как видно из примера, args - это кортеж из всех переданных аргументов функции, и с переменной можно работать также, как и с кортежем.

Функция может принимать и произвольное число именованных аргументов, тогда перед именем ставится **:

```
>>>
```

```
>>> def func(**kwargs):
...     return kwargs
...
>>> func(a=1, b=2, c=3)
{'a': 1, 'c': 3, 'b': 2}
>>> func()
{}
>>> func(a='python')
{'a': 'python'}
```

В переменной kwargs у нас хранится словарь, с которым мы, опять-таки, можем делать все, что нам заблагорассудится.

Анонимные функции, инструкция lambda

Анонимные функции могут содержать лишь одно выражение, но и выполняются они быстрее. Анонимные функции создаются с помощью инструкции **lambda**. Кроме этого, их не обязательно присваивать переменной, как делали мы инструкцией def func():

```
>>>
```

```
>>> func = lambda x, y: x + y
>>> func(1, 2)
3
>>> func('a', 'b')
'ab'
```

```
>>> (lambda x, y: x + y)(1, 2)

3

>>> (lambda x, y: x + y)('a', 'b')

'ab'
```

lambda функции, в отличие от обычной, не требуется инструкция return, а в остальном, ведет себя точно так же:

>>>

```
>>> func = lambda *args: args

>>> func(1, 2, 3, 4)

(1, 2, 3, 4)
```

сключения (exceptions) - ещё один тип данных в python. Исключения необходимы для того, чтобы сообщать программисту об ошибках.

Самый простейший пример исключения - деление на ноль:

>>>

```
>>> 100 / 0

Traceback (most recent call last):

  File "", line 1, in

    100 / 0

ZeroDivisionError: division by zero
```

Разберём это сообщение подробнее: интерпретатор нам сообщает о том, что он поймал исключение и напечатал информацию (**Traceback (most recent call last)**).

Далее имя файла (**File ""**). Имя пустое, потому что мы находимся в интерактивном режиме, строка в файле (**line 1**);

Выражение, в котором произошла ошибка (**100 / 0**).

Название исключения (**ZeroDivisionError**) и краткое описание исключения (**division by zero**).

Разумеется, возможны и другие исключения:

```
>>>
```

```
>>> 2 + '1'

Traceback (most recent call last):

  File "", line 1, in

    2 + '1'

TypeError: unsupported operand type(s) for +: 'int' and 'str'



>>> int('qwerty')

Traceback (most recent call last):

  File "", line 1, in

    int('qwerty')

ValueError: invalid literal for int() with base 10: 'qwerty'
```

В этих двух примерах генерируются исключения TypeError и ValueError соответственно. Подсказки дают нам полную информацию о том, где порождено исключение, и с чем оно связано.

Рассмотрим иерархию встроенных в python исключений, хотя иногда вам могут встретиться и другие, так как программисты могут создавать собственные исключения. Данный список актуален для python 3.3, в более ранних версиях есть незначительные изменения.

- **BaseException** - базовое исключение, от которого берут начало все остальные.
  - **SystemExit** - исключение, порождаемое функцией sys.exit при выходе из программы.
  - **KeyboardInterrupt** - порождается при прерывании программы пользователем (обычно сочетанием клавиш Ctrl+C).
  - **GeneratorExit** - порождается при вызове метода close объекта generator.

- **Exception** - а вот тут уже заканчиваются полностью системные исключения (которые лучше не трогать) и начинаются обыкновенные, с которыми можно работать.
  - **StopIteration** - порождается встроенной функцией next, если в итераторе больше нет элементов.
  - **ArithmeticError** - арифметическая ошибка.
    - **FloatingPointError** - порождается при неудачном выполнении операции с плавающей запятой. На практике встречается нечасто.
    - **OverflowError** - возникает, когда результат арифметической операции слишком велик для представления. Не появляется при обычной работе с целыми числами (так как python поддерживает длинные числа), но может возникать в некоторых других случаях.
    - **ZeroDivisionError** - деление на ноль.
  - **AssertionError** - выражение в функции assert ложно.
  - **AttributeError** - объект не имеет данного атрибута (значения или метода).
  - **BufferError** - операция, связанная с буфером, не может быть выполнена.
  - **EOFError** - функция наткнулась на конец файла и не смогла прочитать то, что хотела.
  - **ImportError** - не удалось импортирование модуля или его атрибута.
  - **LookupError** - некорректный индекс или ключ.
    - **IndexError** - индекс не входит в диапазон элементов.
    - **KeyError** - несуществующий ключ (в словаре, множестве или другом объекте).
  - **MemoryError** - недостаточно памяти.
  - **NameError** - не найдено переменной с таким именем.
    - **UnboundLocalError** - сделана ссылка на локальную переменную в функции, но переменная не определена ранее.
  - **OSError** - ошибка, связанная с системой.
    - **BlockingIOError**
    - **ChildProcessError** - неудача при операции с дочерним процессом.
    - **ConnectionError** - базовый класс для исключений, связанных с подключениями.
      - **BrokenPipeError**
      - **ConnectionAbortedError**
      - **ConnectionRefusedError**
      - **ConnectionResetError**
    - **FileExistsError** - попытка создания файла или директории, которая уже существует.
    - **FileNotFoundError** - файл или директория не существует.
    - **InterruptedError** - системный вызов прерван входящим сигналом.

        - **IsADirectoryError** - ожидался файл, но это директория.
        - **NotADirectoryError** - ожидалась директория, но это файл.
        - **PermissionError** - не хватает прав доступа.
        - **ProcessLookupError** - указанного процесса не существует.
        - **TimeoutError** - закончилось время ожидания.
    - **ReferenceError** - попытка доступа к атрибуту со слабой ссылкой.
    - **RuntimeError** - возникает, когда исключение не попадает ни под одну из других категорий.
    - **NotImplementedError** - возникает, когда абстрактные методы класса требуют переопределения в дочерних классах.
    - **SyntaxError** - синтаксическая ошибка.
        - **IndentationError** - неправильные отступы.
            - **TabError** - смешивание в отступах табуляции и пробелов.
    - **SystemError** - внутренняя ошибка.
    - **TypeError** - операция применена к объекту несоответствующего типа.
    - **ValueError** - функция получает аргумент правильного типа, но некорректного значения.
    - **UnicodeError** - ошибка, связанная с кодированием / раскодированием unicode в строках.
        - **UnicodeEncodeError** - исключение, связанное с кодированием unicode.
        - **UnicodeDecodeError** - исключение, связанное с декодированием unicode.
        - **UnicodeTranslateError** - исключение, связанное с переводом unicode.
    - **Warning** - предупреждение.

Теперь, зная, когда и при каких обстоятельствах могут возникнуть исключения, мы можем их обрабатывать. Для обработки исключений используется конструкция **try - except**.

Первый пример применения этой конструкции:

```
>>>
```

```
>>> try:
...     k = 1 / 0
... except ZeroDivisionError:
...     k = 0
...
```

```
>>> print(k)

0
```

В блоке try мы выполняем инструкцию, которая может породить исключение, а в блоке except мы перехватываем их. При этом перехватываются как само исключение, так и его потомки. Например, перехватывая ArithmeticError, мы также перехватываем FloatingPointError, OverflowError и ZeroDivisionError.

```
>>>
```

```
>>> try:

...     k = 1 / 0

... except ArithmeticError:

...     k = 0

...

>>> print(k)

0
```

Также возможна инструкция except без аргументов, которая перехватывает вообще всё (и прерывание с клавиатуры, и системный выход и т. д.). Поэтому в такой форме инструкция except практически не используется, а используется except Exception. Однако чаще всего перехватывают исключения по одному, для упрощения отладки (вдруг вы ещё другую ошибку сделаете, а except её перехватит).

Ещё две инструкции, относящиеся к нашей проблеме, это **finally** и else. Finally выполняет блок инструкций в любом случае, было ли исключение, или нет (применима, когда нужно непременно что-то сделать, к примеру, закрыть файл). Инструкция else выполняется в том случае, если исключения не было.

```
>>> f = open('1.txt')

>>> ints = []

>>> try:
```

```
...     for line in f:

...         ints.append(int(line))

... except ValueError:

...     print('Это не число. Выходим.')

... except Exception:

...     print('Это что ещё такое?')

... else:

...     print('Всё хорошо.')

... finally:

...     f.close()

...     print('Я закрыл файл.')

...     # Именно в таком порядке: try, группа except, затем else, и только потом finally.

...

Это не число. Выходим.

Я закрыл файл.
```

Модулем в Python называется любой файл с программой (да-да, все те программы, которые вы писали, можно назвать модулями). В этой статье мы поговорим о том, как создать модуль, и как подключить модуль, из стандартной библиотеки или написанный вами.

Каждая программа может импортировать модуль и получить доступ к его классам, функциям и объектам. Нужно заметить, что модуль может быть написан не только на Python, а например, на C или C++.

Подключение модуля из стандартной библиотеки

Подключить модуль можно с помощью инструкции import. К примеру, подключим модуль os для получения текущей директории:

```
>>>
```

```
>>> import os

>>> os.getcwd()

'C:\\Python33'
```

После ключевого слова **import** указывается название модуля. Одной инструкцией можно подключить несколько модулей, хотя этого не рекомендуется делать, так как это снижает читаемость кода. Импортируем модули time и random.

```
>>>
```

```
>>> import time, random

>>> time.time()

1376047104.056417

>>> random.random()

0.9874550833306869
```

После импортирования модуля его название становится переменной, через которую можно получить доступ к атрибутам модуля. Например, можно обратиться к константе e, расположенной в модуле math:

```
>>>
```

```
>>> import math

>>> math.e

2.718281828459045
```

Стоит отметить, что если указанный атрибут модуля не будет найден, возбудится исключение AttributeError. А если не удастся найти модуль для импортирования, то ImportError.

```
>>>
```

```
>>> import notexist

Traceback (most recent call last):

  File "", line 1, in

    import notexist

ImportError: No module named 'notexist'

>>> import math

>>> math.Ë

Traceback (most recent call last):

  File "", line 1, in

    math.Ë

AttributeError: 'module' object has no attribute 'Ë'
```

Использование псевдонимов

Если название модуля слишком длинное, или оно вам не нравится по каким-то другим причинам, то для него можно создать псевдоним, с помощью ключевого слова as.

```
>>>
```

```
>>> import math as m

>>> m.e

2.718281828459045
```

Теперь доступ ко всем атрибутам модуля math осуществляется только с помощью переменной m, а переменной math в этой программе уже не будет (если, конечно, вы после этого не напишете import math, тогда модуль будет доступен как под именем m, так и под именем math).

Инструкция from

Подключить определенные атрибуты модуля можно с помощью инструкции from. Она имеет несколько форматов:

```
from <Название модуля> import <Атрибут 1> [ as <Псевдоним 1> ],
[<Атрибут 2> [ as <Псевдоним 2> ] ...]

from <Название модуля> import *
```

Первый формат позволяет подключить из модуля только указанные вами атрибуты. Для длинных имен также можно назначить псевдоним, указав его после ключевого слова as.

>>>

```
>>> from math import e, ceil as c

>>> e

2.718281828459045

>>> c(4.6)

5
```

Импортируемые атрибуты можно разместить на нескольких строках, если их много, для лучшей читаемости кода:

>>>

```
>>> from math import (sin, cos,

...                   tan, atan)
```

Второй формат инструкции from позволяет подключить все (точнее, почти все) переменные из модуля. Для примера импортируем все атрибуты из модуля sys:

>>>

```
>>> from sys import *

>>> version
```

```
'3.3.2 (v3.3.2:d047928ae3f6, May 16 2013, 00:03:43) [MSC v.1600 32
bit (Intel)]'


>>> version_info


sys.version_info(major=3, minor=3, micro=2, releaselevel='final',
serial=0)
```

Следует заметить, что не все атрибуты будут импортированы. Если в модуле определена переменная __all__ (список атрибутов, которые могут быть подключены), то будут подключены только атрибуты из этого списка. Если переменная __all__ не определена, то будут подключены все атрибуты, не начинающиеся с нижнего подчёркивания. Кроме того, необходимо учитывать, что импортирование всех атрибутов из модуля может нарушить пространство имен главной программы, так как переменные, имеющие одинаковые имена, будут перезаписаны.

Создание своего модуля на Python

Теперь пришло время создать свой модуль. Создадим файл mymodule.py, в которой определим какие-нибудь функции:

```
def hello():

    print('Hello, world!')



def fib(n):

    a = b = 1

    for i in range(n - 2):

        a, b = b, a + b

    return b
```

Теперь в этой же папке создадим другой файл, например, main.py:

```
import mymodule


```

```
mymodule.hello()


print(mymodule.fib(10))
```

Выведет:

```
Hello, world!


55
```

Поздравляю! Вы **сделали свой модуль**! Напоследок отвечу ещё на пару вопросов, связанных с созданием модулей:

Как назвать модуль?

Помните, что вы (или другие люди) будут его импортировать и использовать в качестве переменной. Модуль нельзя именовать также, как и ключевое слово (их список можно посмотреть тут). Также имена модулей нельзя начинать с цифры. И не стоит называть модуль также, как какую-либо из встроенных функций. То есть, конечно, можно, но это создаст большие неудобства при его последующем использовании.

Куда поместить модуль?

Туда, где его потом можно будет найти. Пути поиска модулей указаны в переменной sys.path. В него включены текущая директория (то есть модуль можно оставить в папке с основной программой), а также директории, в которых установлен python. Кроме того, переменную sys.path можно изменять вручную, что позволяет положить модуль в любое удобное для вас место (главное, не забыть в главной программе модифицировать sys.path).

Можно ли использовать модуль как самостоятельную программу?

Можно. Однако надо помнить, что при импортировании модуля его код выполняется полностью, то есть, если программа что-то печатает, то при её импортировании это будет напечатано. Этого можно избежать, если проверять, запущен ли скрипт как программа, или импортирован. Это можно сделать с помощью переменной __name__, которая определена в любой программе, и равна "__main__", если скрипт запущен в качестве главной программы, и имя, если он импортирован. Например, mymodule.py может выглядеть вот так:

```
def hello():

    print('Hello, world!')
```

```
def fib(n):

    a = b = 1

    for i in range(n - 2):

        a, b = b, a + b

    return b



if __name__ == "__main__":

    hello()

    for i in range(10):

        print(fib(i))
```