

ОСНОВЫ ПРОГРАММИРОВАНИЯ НА ЯЗЫКЕ Python

Урок 6

Модули, пакеты,
исключения

Contents

Использование модулей	4
Что такое модуль?.....	4
Как использовать модули?.....	6
Импортирование модулей.....	8
Импортирование модулей: ключевое слово as.....	16
Полезные модули.....	18
Работа со стандартными модулями	18
Некоторые функции из модуля math.....	20
Есть ли настоящая случайность в компьютерах?	23
Некоторые функции из модуля random.....	25
Как узнать где вы?.....	29
Некоторые функции из модуля platform	31
Каталог модулей Python.....	36

Модули и пакеты	38
Что такое пакет?	38
Ваш первый модуль	39
Ваш первый модуль: продолжение	41
Ваш первый пакет	51
Ошибки — программист	61
Ошибки, сбои и другие неприятности	61
Исключения	62
Анатомия исключений	73
Исключения	73
Полезные исключения	84
Встроенные исключения	84

Материалы урока прикреплены к данному PDF-файлу. Для доступа к материалам, урок необходимо открыть в программе [Adobe Acrobat Reader](#).

Использование модулей

Что такое модуль?

Компьютерный код имеет тенденцию к росту. Можно сказать, что код, который не растет, вероятно, полностью непригоден или устарел. Реальный, востребованный и широко используемый код постоянно развивается, так как требования пользователей и их пользователей развиваются в своем собственном ритме.

Код, который не способен удовлетворить потребности пользователей, будет быстро забыт и немедленно заменен новым, более качественным и гибким кодом. Будьте готовы к этому, и никогда не рассчитывайте, что какая-то из ваших программ в конечном итоге будет полностью завершена. Завершение является переходным состоянием и обычно быстро проходит после первого сообщения об ошибке. Сам Python является хорошим примером того, как действует это правило.

Растущий код на самом деле является растущей проблемой. Большой код всегда означает более сложное обслуживание. Поиск ошибок всегда проще, когда код небольшой (точно так же проще искать механические поломки в небольшом и простом механизме).

Более того, когда вы ожидаете, что создаваемый код будет очень большим (вы можете использовать общее количество строк исходного текста в качестве полезной, но не очень точной меры измерения кода), вы вероятно

захотите (или, скорее, вас заставят) разделить его на множество частей, параллельно реализуемых несколькими, десятками, несколькими десятками или даже несколькими сотнями отдельных разработчиков.

Конечно, этого нельзя сделать, используя один большой исходный файл, который редактируется всеми программистами одновременно. Это, безусловно, было бы эффектным зрелищем.

Если вы хотите, чтобы такой программный проект был успешно завершен, у вас должны быть средства, позволяющие вам:

- разделить все задачи между разработчиками;
- объединить все созданные части в одно рабочее целое.

Например, определенный проект можно разделить на две основные части:

- пользовательский интерфейс (часть, которая взаимодействует с пользователем с помощью виджетов и графического экрана);
- логическая часть (часть обработки данных и получения результатов).

Каждую из этих частей можно (скорее всего) разделить на более мелкие и так далее. Такой процесс часто называют декомпозицией.

Например, если бы вас попросили устроить свадьбу, вы бы не делали все сами — вы бы нашли профессионалов и разделили задачу между ними.

Как разделять элемент программного обеспечения на отдельные, но взаимодействующие части? В этом и вопрос. Ответом являются модули

Как использовать модули?

Работа с модулями состоит из двух разных проблем:

- *первая* (вероятно, самая распространенная) возникает, когда вы хотите использовать уже существующий модуль, написанный кем-то другим или созданный вами самостоятельно во время работы над каким-либо сложным проектом — в этом случае вы являетесь пользователем модуля;
- *вторая* же появляется, когда вы хотите создать новый модуль либо для собственного использования, либо для облегчения жизни других программистов — вы являетесь поставщиком модуля.

Давайте обсудим их отдельно.

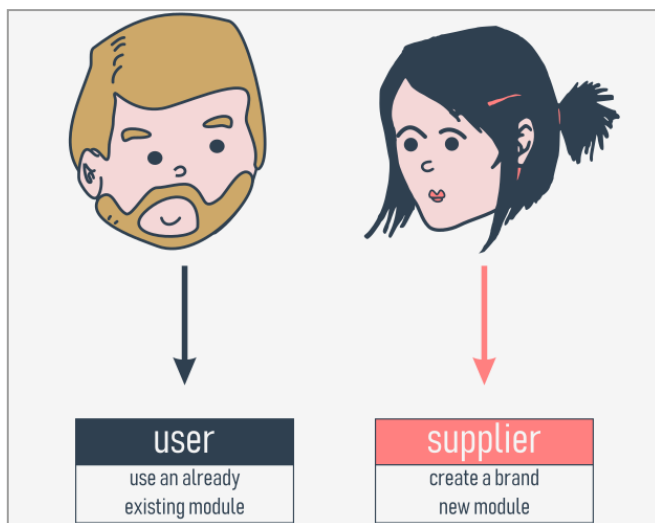


Рисунок 1

Прежде всего, модуль идентифицируется по имени. Если вы хотите использовать какой-либо модуль, вам

нужно знать имя. Ряд модулей (довольно большой) поставляется вместе с самим Python. Вы можете считать их «дополнительным оборудованием Python».

Все эти модули вместе со встроенными функциями образуют стандартную библиотеку Python — особую библиотеку, в которой модули играют роль книг (можно даже сказать, что папки играют роль полок). Если вы хотите взглянуть на полный список всех «томов», собранных в этой библиотеке, вы можете найти его здесь: <https://docs.python.org/3/library/index.html>.

Каждый модуль состоит из сущностей (как книга состоит из глав). Этими сущностями могут быть функции, переменные, константы, классы и объекты. Если вы знаете, как получить доступ к определенному модулю, вы можете использовать любые объекты, которые он хранит.

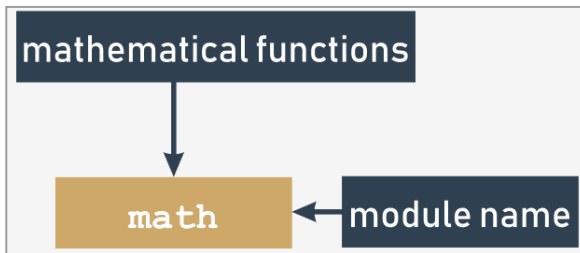


Рисунок 2

Давайте начнем обсуждение с одного из наиболее часто используемых модулей с именем `math`. Его имя говорит само за себя — модуль содержит богатый набор сущностей (не только функций), которые позволяют программисту эффективно реализовывать вычисления, требующие использования математических функций, таких как `sin()` или `log()`.

Импортирование модулей

Чтобы сделать модуль пригодным для использования, вы должны импортировать его (представьте, что вы берете книгу с полки). Импортирование модуля выполняется инструкцией с именем `import`. Примечание: `import` также является ключевым словом (со всеми вытекающими отсюда последствиями).

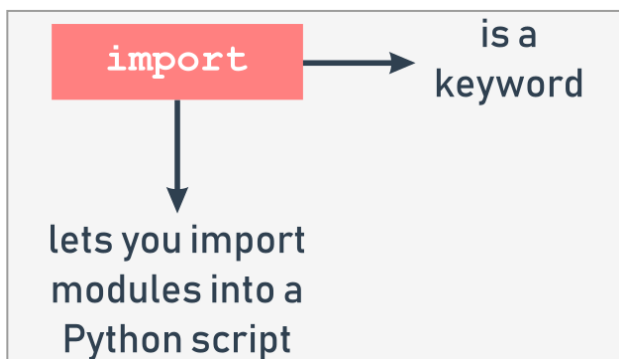


Рисунок 3

Предположим, вы хотите использовать две сущности, предоставляемые модулем `math`:

- *символ (константа)*, представляющий точное (как можно более точное, используя числа двойной точности) значение π (хотя использование греческой буквы для именования переменной полностью возможно в Python, этот символ называется `pi` — это более удобное решение, особенно для той части мира, которая не имеет и не собирается использовать греческую клавиатуру)
- *функция* с именем `sin()` (компьютерный эквивалент математической функции синуса)

Обе эти сущности доступны через модуль `math`, но способ их использования сильно зависит от того, как был выполнен импорт.

Самый простой способ импортировать конкретный модуль — использовать инструкцию импорта следующим образом:

```
import math
```

Конструкция содержит:

- ключевое слово `import`;
- имя модуля, который подлежит импорту.

Инструкция может находиться где угодно в вашем коде, но она должна быть размещена перед первым использованием любой из сущностей модуля.

Если вы хотите (или должны) импортировать более одного модуля, вы можете сделать это, повторив конструкцию `import` или перечислив модули после ключевого слова `import`, как здесь:

```
import math, sys
```

Инструкция импортирует два модуля, первый с именем `math`, а затем второй с именем `sys`.

Список модулей может быть произвольно длинным.

Чтобы продолжить, вам нужно ознакомиться с важным термином: пространство имен.

Не волнуйтесь, мы не будем вдаваться в подробности — объяснение будет максимально кратким.

Пространство имен — это пространство (понимаемое в нефизическом контексте), в котором существуют

некоторые имена, и имена не конфликтуют друг с другом (то есть нет двух разных объектов с одинаковыми именами). Можно сказать, что каждая социальная группа является пространством имен — группа имеет тенденцию называть каждого из своих членов уникальным образом (например, родители не будут давать своим детям одинаковые имена).

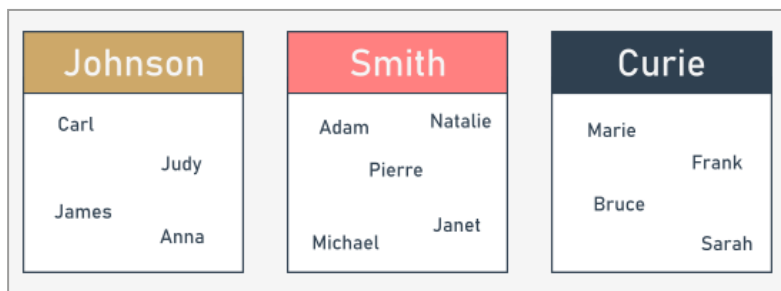


Рисунок 4

Эту уникальность можно достичь многими способами, например, используя псевдонимы вместе с именами (это будет работать в небольшой группе, такой как класс в школе), или назначая специальные идентификаторы всем членам группы (номер социального страхования является хорошим примером такой практики).

Внутри определенного пространства имен каждое имя должно оставаться уникальным. Таким образом, некоторые имена могут исчезнуть, когда в пространство имен входит любая другая сущность уже известного имени. Мы вам покажем, как это работает и как это контролировать, но сначала вернемся к импорту.

Если модуль с указанным именем существует и доступен (модуль фактически является исходным файлом

Python), Python импортирует его содержимое, т.е. все имена, определенные в модуле, становятся известными, но они не входят в пространство имен вашего кода.

Это означает, что вы можете иметь свои собственные сущности с именем `sin` или `pi`, и импорт никак не повлияет на них.



Рисунок 5

В этот момент вам может быть интересно, как получить доступ к `pi` из модуля `math`.

Чтобы сделать это, вы должны присвоить `pi` имя его исходного модуля.

Посмотрите на фрагмент ниже, это способ, которым вы соотносите имена `pi` и `sin` с именем его исходного модуля:

```
math.pi  
math.sin
```

Все просто, вы пишете:

- имя модуля (здесь — `math`);
- точку;
- имя сущности (здесь — `pi`).

Такая форма четко указывает пространство имен, в котором существует имя.

Примечание: использование этой спецификации обязательно, если модуль был импортирован по инструкции модуля `import`. Неважно, конфликтуют ли какие-либо имена из вашего кода и из пространства имен модуля.

Этот первый пример не будет очень сложным — мы просто хотим вывести значение $\sin(1/2\pi)$.

Посмотрите на код ниже. Вот как мы это проверяем.

```
1 import math
2 print (math.sin (math.pi/2))
```

Код выводит ожидаемое значение: 1.0.

Примечание: удаление любой из двух спецификаций сделает код ошибочным. Нет другого способа войти в пространство имен `math`, если вы сделали следующее: `import math`.

Теперь мы покажем вам, как могут сосуществовать два пространства имен (ваше и модуля).

Посмотрите на пример ниже.

```
1 import math
2
3 def sin(x):
4     if 2 * x == pi:
5         return 0.99999999
6     else:
7         return None
8
9 pi = 3.14
10
11 print(sin(pi/2))
12 print(math.sin (math.pi/2))
```

Здесь мы определили наши `pi` и `sin`.

Запустите программу. Код должен выдавать следующий результат:

```
0.999999999
1.0
```

Как видите, сущности не влияют друг на друга.

Во втором методе синтаксис `import` точно указывает, какая сущность (или сущности) модуля являются приемлемыми в коде:

```
from math import pi
```

Инструкция состоит из следующих элементов:

- ключевое слово `from`;
- имя модуля, который надо (выборочно) импортировать;
- ключевое слово `import`;
- имя или список имен сущности/сущностей, которые импортируются в пространство имен.

Инструкция приводит к следующему:

- перечисленные сущности (и только они) импортируются из указанного модуля;
- имена импортированных сущностей доступны без спецификации.

Примечание: другие сущности не импортируются. Более того, вы не можете импортировать дополнительные сущности, используя спецификацию — такая строка:

```
print(math.e)
```

приведет к ошибке («e» это число Эйлера: 2,71828...).

Давайте перепишем предыдущий скрипт, чтобы включить новую технику.

Вот так:

```
from math import sin, pi
print(sin(pi/2))
```

Вывод должен быть таким же, как и раньше, так как на самом деле мы использовали те же сущности, что и раньше: **1.0**. Скопируйте код, вставьте его в ваш редактор и запустите программу.

Код выглядит проще? Возможно, но внешний вид — не единственный плюс такого рода импорта.

Посмотрите на код ниже. Внимательно его проанализируйте:

```
1 from math import sin, pi
2
3 print(sin(pi/2))
4
5 pi = 3.14
6
7 def sin(x):
8     if 2 * x == pi:
9         return 0.99999999
10    else:
11        return None
12
13 print(sin(pi/2))
```

- *строка 01*: выполнить выборочный импорт;
- *строка 03*: использовать импортированные сущности и получить ожидаемый результат (**1.0**)
- *строки с 05 по 11*: переопределить значения **pi** и **sin** - фактически они заменяют первоначальные (импортированные) определения в пространстве имен кода

- *строка 13*: получить **0.99999999**, что подтверждает наши выводы.

Давайте сделаем еще один тест. Посмотрите на следующий код:

```
pi = 3.14                                # строка 01
def sin(x):
    if 2 * x == pi:
        return 0.99999999
    else:
        return None                      # строка 07

print(sin(pi/2))                         # строка 09
from math import sin, pi                 # строка 12
print(sin(pi/2))                         # строка 14
```

Здесь мы изменили последовательность операций кода:

- *строки с 01 по 07*: определить наши собственные **pi** и **sin**;
- *строка 09*: использовать их (на экране появляется **0.99999999**);
- *строка 12*: выполнить импорт — импортированные символы заменяют свои предыдущие определения в пространстве имен;
- *строка 14*: получить в результате **1.0**.

В третьем методе синтаксис **import** является более агрессивной формой ранее представленного:

```
from module import *
```

Как видите, имя сущности (или список имен сущностей) заменяется одной звездочкой (*).

Такая инструкция импортирует все сущности из указанного модуля.

Это удобно? Да, так как это освобождает вас от обязанности перечислять все имена, которые вам нужны.

Это небезопасно? Да, это так — если вы не знаете всех имен, предоставленных модулем, вы вероятно не сможете избежать конфликтов имен. Считайте это временным решением и старайтесь не использовать его в обычном коде.

Импортирование модулей: ключевое слово **as**

Если вы используете вариант модуля импорта и вам не нравится имя конкретного модуля (например, оно совпадает с одним из уже определенных вами сущностей, поэтому спецификация становится проблематичной), вы можете дать ему любое имя, которое вам нравится — это называется псевдонимом (**aliasing**).

При псевдониме модуль будет идентифицирован под другим именем, а не под исходным. Это также может сократить специфицированные имена.

Создание псевдонима выполняется вместе с импортом модуля и требует следующей формы инструкции импорта:

```
import module as alias
```

«Модуль» идентифицирует исходное имя модуля, а «псевдоним» — это имя, которое вы хотите использовать вместо исходного

Примечание: **as** это ключевое слово.

Если вам нужно изменить слово **math**, вы можете ввести собственное имя, как в примере:


```
import math as m
print(m.sin(m.pi/2))
```

Примечание: после успешного выполнения импорта с псевдонимом исходное имя модуля становится недоступным и не должно использоваться

В свою очередь, когда вы используете вариант `from module import name` и вам нужно изменить имя сущности, вы создаете псевдоним для сущности. Это приведет к замене имени на выбранный вами псевдоним

Вот как это делается:

```
from module import name as alias
```

Как и ранее, исходное имя (не псевдоним) становится недоступным.

Фраза `name as alias` может повторяться — используйте запятые для разделения подобных фраз, например:

```
from module import n as a, m as b, o as c
```

Пример может показаться немного странным, но он работает:

```
from math import pi as PI, sin as sine
print(sine(PI/2))
```

Теперь вы знакомы с основами использования модулей. Давайте посмотрим на некоторые модули и некоторые их полезные сущности.

Полезные модули

Работа со стандартными модулями

Прежде чем мы начнем изучать некоторые стандартные модули Python, мы хотим представить вам функцию `dir()`. Она не имеет ничего общего с командой `dir`, которую вы знаете из консолей Windows и Unix, поскольку `dir()` не показывает содержимое каталога/папки на диске, но без всяких сомнений она делает нечто действительно похожее — она способна раскрыть все имена, представленные через определенный модуль.

Есть одно условие: модуль должен быть предварительно импортирован целиком (т.е. с помощью инструкции `import module` — `from module` недостаточно).

Функция возвращает отсортированный по алфавиту список, содержащий имена всех сущностей, доступных в модуле, идентифицируемых по имени, переданному функции в качестве аргумента:

```
dir(module)
```

***Примечание:** если имя модуля было псевдонимом, вы должны использовать псевдоним, а не оригинальное имя.*

Использование функции внутри обычного скрипта не имеет особого смысла, но все же возможно.

Например, вы можете запустить следующий код для вывода имен всех сущностей в модуле `math`:

```
import math
for name in dir(math):
    print(name, end="\t")
```

Пример кода должен выдать следующий результат:

```
→
__doc__  __loader__  __name__  __package__
asinh    atan    atan2    atanh    ceil    copysign
exp      expm1    fabs     factorial floor    fmod    →
isinf    isnan    ldexp    lgamma   log     log10   loglp
sinh     sqrt     tan      tanh     trunc

__doc__  __loader__  __name__  __package__
asinh    atan    atan2    atanh    ceil    copysign
exp      expm1    fabs     factorial floor    fmod
isinf    isnan    ldexp    lgamma   log     log10   loglp
sinh     sqrt     tan      tanh     trunc
```

Вы заметили имена начинающиеся с «__» вверху списка? Мы расскажем вам больше о них, когда будем говорить о проблемах, связанных с написанием ваших собственных модулей. Некоторые из имен могут оживить воспоминания из уроков математики, и у вас, вероятно, не возникнет проблем с угадыванием их значения.

Использование функции `dir()` внутри кода может показаться не очень полезным — обычно вы хотите знать содержимое определенного модуля, прежде чем писать и запускать код.

К счастью, вы можете выполнить функцию непосредственно в консоли Python (IDLE), без необходимости писать и запускать отдельный скрипт

Вот как это можно сделать:

```
import math
dir(math)
```

Вы увидите нечто, похожее на это:

```

Python 3.6.1 (v3.6.1:69c0db5, Mar 21 2017, 17:54:52) [MSC v.1900 32 bit (Intel)]
> on win32
Type "copyright", "credits" or "license()" for more information.
>>> import math
>>> dir(math)
['_doc_', '_loader_', '_name_', '_package_', '_spec_', 'acos', 'acosh',
 'asin', 'asinh', 'atan', 'atan2', 'atanh', 'ceil', 'copysign', 'cos', 'cosh',
 'degrees', 'e', 'erf', 'erfc', 'exp', 'expm1', 'fabs', 'factorial', 'floor',
 'fmod', 'frexp', 'fsum', 'gamma', 'gcd', 'hypot', 'inf', 'isclose', 'isfinite',
 'isinf', 'isnan', 'ldexp', 'lgamma', 'log', 'log10', 'loglp', 'log2', 'modf', 'nan',
 'pi', 'pow', 'radians', 'sin', 'sinh', 'sqrt', 'tan', 'tanh', 'tau', 'trunc']
>>>
  
```

Некоторые функции из модуля math

Давайте начнем с краткого разбора некоторых функций, предоставляемых модулем **math**.

Мы выбрали их произвольно, но это не значит, что функции, которые мы здесь не упомянули, менее значимы. Вы можете самостоятельно погрузиться в изучение модулей — у нас не хватит места или времени, чтобы подробно обо всем здесь поговорить.

Первая группа функций **math** связана с тригонометрией:

- $\sin(x)$ → синус x ;
- $\cos(x)$ → косинус x ;
- $\tan(x)$ → тангенс x .

Все эти функции принимают один аргумент (значение угла, выраженное в радианах) и возвращают соответствующий результат (будьте осторожны с **tan()** — не все аргументы принимаются).

Конечно, есть и их обратные версии:

- $\text{asin}(x)$ → арксинус x ;
- $\text{acos}(x)$ → арккосинус x ;
- $\text{atan}(x)$ → арктангенс x .

Эти функции принимают один аргумент (обратите внимание на области определения) и возвращают значение угла в радианах.

Чтобы эффективно работать с угловыми измерениями, модуль `math` предоставляет вам следующие сущности:

- `pi` → константа со значением, приближенным к π ;
- `radians(x)` → функция, которая преобразует `x` из градусов в радианы;
- `degrees(x)` → действие в обратном направлении (от радианов к градусам).

Посмотрите на код ниже.

```
1 from math import pi, radians, degrees, sin, cos, tan, asin
2
3 ad = 90
4 ar = radians(ad)
5 ad = degrees(ar)
6
7 print(ad == 90.)
8 print(ar == pi / 2.)
9 print(sin(ar) / cos(ar) == tan(ar))
10 print(asin(sin(ar)) == ar)
```

Пример кода не очень сложный, но можете ли вы предсказать результат?

Помимо круговых функций (перечисленных выше) модуль `math` также содержит набор их гиперболических аналогов:

- `sinh(x)` → гиперболический синус;
- `cosh(x)` → гиперболический косинус;
- `tanh(x)` → гиперболический тангенс;
- `asinh(x)` → гиперболический арксинус;
- `acosh(x)` → гиперболический арккосинус;
- `atanh(x)` → гиперболический арктангенс.

Другая группа функций `math` состоит из функций, связанных с возведением в степень:

- `e` → константа со значением, которое приближено к числу Эйлера (`e`);
- `exp(x)` → нахождение значения e^x ;
- `log(x)` → натуральный логарифм `x`;
- `log(x, b)` → логарифм `x` по основанию `b`;
- `log10(x)` → десятичный логарифм `x` (более точный, чем `log(x, 10)`);
- `log2(x)` → двоичный логарифм `x` (более точный, чем `log(x, 2)`);
- `pow(x, y)` → нахождение значения x^y (обратите внимание на области определения)

Это встроенная функция, и ее не нужно импортировать.

Посмотрите на код ниже. Можете ли вы предсказать его вывод?

```
1 from math import e, exp, log
2
3 print(pow(e, 1) == exp(log(e)))
4 print(pow(2, 2) == exp(2 * log(2)))
5 print(log(e, e) == exp(0))
```

Последняя группа состоит из некоторых функций общего назначения, таких как:

- `ceil(x)` → верхнее округление `x` (наименьшее целое число, больше или равное `x`);
- `floor(x)` → нижнее округление `x` (наибольшее целое число, меньше или равное `x`);

- `trunc(x)` → значение `x`, усеченное до целого числа (будьте осторожны — оно не эквивалентно ни верхнему ни нижнему округление);
- `factorial(x)` → возвращает `x!` (`x` должен быть целым и не отрицательным);
- `hypot(x, y)` → возвращает длину гипотенузы прямоугольного треугольника с длинами катетов, равными `x` и `y` (аналогично `sqrt(pow(x, 2) + pow(y, 2))`), но более точно).

Посмотрите на код ниже.

```

1 from math import ceil, floor, trunc
2
3 x = 1.4
4 y = 2.6
5
6 print(floor(x), floor(y))
7 print(floor(-x), floor(-y))
8 print(ceil(x), ceil(y))
9 print(ceil(-x), ceil(-y))
10 print(trunc(x), trunc(y))
11 print(trunc(-x), trunc(-y))

```

Внимательно проанализируйте программу.

Она демонстрирует принципиальные различия между `ceil()`, `floor()` и `trunc()`.

Запустите программу и проверьте ее вывод.

Есть ли настоящая случайность в компьютерах?

Другой модуль, о котором стоит упомянуть, это модуль с именем `random`.

Он предоставляет некоторые механизмы, позволяющие вам работать с псевдослучайными числами.

Обратите внимание на префикс *псевдо* — числа, сгенерированные модулями, могут выглядеть случайными в том смысле, что вы не можете предсказать их последующие значения, но не забывайте, что все они рассчитываются с использованием очень усовершенствованных алгоритмов.



Рисунок 6

Алгоритмы не случайны — они детерминированные и предсказуемы. Только те физические процессы, которые полностью выходят из-под нашего контроля (например, интенсивность космического излучения), могут быть использованы в качестве источника фактических случайных данных. Данные, полученные с помощью детерминированных компьютеров, ни в коем случае не могут быть случайными.

Генератор случайных чисел принимает значение, называемое начальным числом (*seed*), обрабатывает его как входное значение, вычисляет «случайное» число на основе этого (метод зависит от выбранного алгоритма) и создает новое начальное значение.

Длина цикла, в котором все начальные значения уникальны, может быть очень большой, но она не бесконечна — рано или поздно начальные значения начнут повторяться, и генерируемые значения также повторятся. Это нормально. Это особенность, а не ошибка или баг.

Исходное начальное значение, заданное во время запуска программы, определяет порядок появления сгенерированных значений.

Случайный фактор процесса может быть увеличен путем задания начального числа, взятого из текущего времени — это может гарантировать, что каждый запуск программы будет начинаться с другого начального значения (следовательно, он будет использовать разные случайные числа).

К счастью, такая инициализация выполняется Python во время импорта модуля.

Некоторые функции из модуля `random`

Самая общая функция с именем `random()` (не путать с именем модуля) создает число с плавающей запятой x из диапазона $(0.0, 1.0)$ — другими словами: $(0.0 \leq x < 1.0)$.

Приведенный ниже пример создаст пять псевдослучайных значений — поскольку их значения определяются текущим (довольно непредсказуемым) начальным значением, их невозможно угадать.

```
1 from random import random
2
3 for i in range(5):
4     print(random())
```

Запустите программу.

Функция `seed()` может напрямую устанавливать начальное число генератора. Мы покажем вам два ее варианта:

- `seed()` — устанавливает начальное число на текущее время;
- `seed(int_value)` — устанавливает начальное число на целочисленное значение `int_value`.

Мы изменили предыдущую программу — по сути, мы удалили любые следы случайности из кода:

```
from random import random, seed
seed(0)

for i in range(5):
    print(random())
```

Из-за того, что начальное число всегда установлено на одно и то же значением, последовательность сгенерированных значений всегда выглядит одинаково.

Вот результат выполнения кода:

```
0.844421851525
0.75795440294
0.420571580831
0.258916750293
0.511274721369
```

А что у вас?

Примечание: *ваши значения могут немного отличаться от наших, если ваша система использует более точную или менее точную вещественную*

арифметику, но разница будет заметна довольно далеко от десятичной точки.

Если вам нужны целочисленные случайные значения, лучше подойдет одна из следующих функций:

- `randrange(end);`
- `randrange(beg, end);`
- `randrange(beg, end, step);`
- `randint(left, right).`

Вызов первых трех будут генерировать целое число, взятое (псевдослучайно) из диапазона (соответственно):

- `range(end);`
- `range(beg, end);`
- `range(beg, end, step).`

Обратите внимание на неявное правостороннее исключение! Последняя функция является эквивалентом `randrange(left, right+1)` — она генерирует целочисленное значение `i`, которое попадает в диапазон `[left, right]` (без исключения с правой стороны).

Посмотрите на код ниже.

```
1 from random import randrange, randint
2
3 print(randrange(1), end=' ')
4 print(randrange(0, 1), end=' ')
5 print(randrange(0, 1, 1), end=' ')
6 print(randint(0, 1))
```

Эта программа, следовательно, выведет строку, состоящую из трех нулей и либо с нулем, либо с единицей на четвертом месте.

Предыдущие функции имеют один важный недостаток — они могут создавать повторяющиеся значения, даже

если число последующих вызовов не превышает ширину указанного диапазона.

Посмотрите на код ниже.

```
1 from random import randint
2
3 for i in range(10):
4     print(randint(1, 10), end=',')
```

Скорее всего, программа выводит ряд чисел, в которых некоторые элементы не являются уникальными.

Вот что мы получили в одном из запусков:

```
9, 4, 5, 4, 5, 8, 9, 4, 8, 4,
```

Как видите, это не очень хороший инструмент для генерации чисел в лотерее. К счастью, есть лучшее решение, чем написание собственного кода для проверки уникальности «вытянутых» чисел.

Эта функция названа очень наглядно — **choice**:

- **choice(sequence);**
- **sample(sequence, elements_to_choose=1).**

Первый вариант выбирает «случайный» элемент из входной последовательности и возвращает его.

Второй создает список (образец), состоящий из элемента **elements_to_choose** (по умолчанию 1), «вытянутого» из входной последовательности.

Другими словами, функция выбирает некоторые входные элементы, возвращая список с выбором. Элементы в образце расположены в случайном порядке.

Примечание: *elements_to_choose не должно быть больше длины входной последовательности.*

Посмотрите на код ниже:

```
from random import choice, sample
lst = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
print(choice(lst))
print(sample(lst, 5))
print(sample(lst, 10))
```

Опять же, вывод программы не предсказуем. Наши результаты выглядели следующим образом:

```
4
[3, 1, 8, 9, 10]
[10, 8, 5, 1, 6, 4, 3, 9, 7, 2]
```

Как узнать где вы?

Иногда бывает необходимо узнать информацию, не связанную с Python. Например, вам может понадобиться узнать местоположение вашей программы в более широкой среде компьютера.

Представьте себе среду вашей программы в виде пирамиды, состоящей из нескольких слоев или платформ.



Рисунок 7

Слои выглядят так:

- ваш (работающий) код расположен вверху;
- Python (точнее — его среда выполнения) находится прямо под ним;
- следующий слой пирамиды заполнен ОС (операционной системой) — среда Python предоставляет некоторые из своих функций, используя сервисы операционной системы; Python, хотя и очень мощный, но не всемогущий — он вынужден использовать много помощников, если он собирается обрабатывать файлы или связываться с физическими устройствами;
- самый нижний уровень — это аппаратное обеспечение: процессор (или процессоры), сетевые интерфейсы, устройства интерфейса пользователя (мыши, клавиатуры и т.д.) и все другие аппараты, необходимые для работы компьютера; ОС знает, как управлять им, и использует множество приемов, чтобы привести все части в единый механизм.

Это означает, что некоторые из ваших действий (или, скорее, вашей программы) должны пройти долгий путь для успешного выполнения — представьте, что:

- ваш код хочет создать файл, поэтому он вызывает одну из функций Python;
- Python принимает заказ, реорганизует его в соответствии с требованиями локальной ОС (это все равно, что поставить отметку «утверждено» по вашему запросу) и отправляет его дальше (это может напоминать вам цепочку команд);

- ОС проверяет, является ли запрос обоснованным и действительным (например, соответствует ли имя файла некоторым правилам синтаксиса), и пытается создать файл; такая операция, казалось бы, очень простая, но она атомарная — она состоит из множества незначительных шагов, предпринятых...
- аппаратным обеспечением, которое отвечает за активацию устройств хранения (жесткий диск, твердотельные устройства и т.д.) для удовлетворения потребностей ОС.

Обычно вы не в курсе всей этой суеты — вы хотите, чтобы файл был создан, и это все.

Но иногда вы хотите узнать больше — например, имя ОС, на которой установлен Python, и некоторые характеристики, описывающие оборудование, на котором установлена ОС.

Существует модуль, предоставляющий некоторые средства, позволяющие вам узнать, где вы находитесь и какие компоненты работают для вас. Модуль называется платформой. Мы покажем вам некоторые функции, которые он вам предоставляет.

Некоторые функции из модуля `platform`

Модуль `platform` позволяет получить доступ к базовым данным платформы, т.е. к оборудованию, операционной системе и информации о версии интерпретатора.

Существует функция, которая с легкостью может показать вам все нижележащие слои, называемая также `platform`. Она просто возвращает строку, описываю-

щую среду; таким образом, ее вывод скорее адресован людям, а не автоматизированной обработке (вы скоро это увидите).

Вот как вы можете вызвать ее:

```
platform(aliased = False, terse = False)
```

А теперь:

- **aliased** → если установлено значение **True** (или любое ненулевое значение) то может привести к тому, что функция представит альтернативные имена нижележащих слоев вместо общих;
- **terse** → если установлено значение **True** (или любое ненулевое значение) это может убедить функцию представить более короткую форму результата (если это возможно).

Мы запустили нашу программу

```
1 from platform import platform
2
3 print(platform())
4 print(platform(1))
5 print(platform(0, 1))
```

используя три разные платформы — вот что мы получили:

Intel x86 + Windows ® Vista (32 bit):

Windows-Vista-6.0.6002-SP2

Windows-Vista-6.0.6002-SP2

Windows-Vista

Intel x86 + Gentoo Linux (64 bit):

Linux-3.18.62-g6-x86_64-Intel-R-_Core-TM-_i3-2330M_
CPU_@_2.20GHz-with-gentoo-2.3


```
Linux-3.18.62-g6-x86_64-Intel-R-_Core-TM-_i3-2330M_
CPU_@_2.20GHz-with-gentoo-2.3
Linux-3.18.62-g6-x86_64-Intel-R-_Core-TM-_i3-2330M_
CPU_@_2.20GHz-with-glibc2.3.4
```

Raspberry PI2 + Raspbian Linux (32 bit):

```
Linux-4.4.0-1-rpi2-armv7l-with-debian-9.0
Linux-4.4.0-1-rpi2-armv7l-with-debian-9.0
Linux-4.4.0-1-rpi2-armv7l-with-glibc2.9
```

Вы также можете запустить пример программы в IDLE на локальном компьютере, чтобы проверить, какой вывод вы получите.

Иногда вам может понадобиться общее имя процессора, на котором ваша ОС работает вместе с Python и вашим кодом — функция с именем `machine()` скажет вам об этом. Как и ранее, функция возвращает строку.

Опять же, мы запустили пример программы

```
1 from platform import machine
2
3 print(machine())
```

на трех разных платформах:

Intel x86 + Windows ® Vista (32 bit):

```
x86
```

Intel x86 + Gentoo Linux (64 bit):

```
x86_64
```

Raspberry PI2 + Raspbian Linux (32 bit):

```
armv7l
```

Функция `processor()` возвращает строку, заполненную реальным именем процессора (если это возможно).

Еще раз, мы запустили пример программы

```
1 from platform import processor
2
3 print(processor())
```

на трех разных платформах:

Intel x86 + Windows ® Vista (32 bit):

x86

Intel x86 + Gentoo Linux (64 bit):

Intel(R) Core(TM) i3-2330M CPU @ 2.20GHz

Raspberry PI2 + Raspbian Linux (32 bit):

armv7l

Функция с именем `system()` возвращает общее имя ОС в виде строки.

```
1 from platform import system
2
3 print(system())
```

Наши примеры платформ представили себя так:

Intel x86 + Windows ® Vista (32 bit):

Windows

Intel x86 + Gentoo Linux (64 bit):

Linux

Raspberry PI2 + Raspbian Linux (32 bit):

Linux

Версия ОС предоставляется в виде строки функцией `version()`.

Запустите код и проверьте его вывод.

```
1 from platform import version
2
3 print(version())
```

Вот что мы получили:

```
Intel x86 + Windows ® Vista (32 bit):
6.0.6002

Intel x86 + Gentoo Linux (64 bit):
#1 SMP PREEMPT Fri Jul 21 22:44:37 CEST 2017

Raspberry PI2 + Raspbian Linux (32 bit):
#1 SMP Debian 4.4.6-1+rpil4 (2016-05-05)
```

Если вам нужно знать, в какой версии Python работает ваш код, вы можете проверить это с помощью ряда выделенных функций — вот две из них:

- `python_implementation()` → возвращает строку, обозначающую реализацию Python (ожидайте здесь CPython, если только вы не решите использовать какую-либо неканоническую ветку Python)
- `python_version_tuple()` → возвращает трехэлементный кортеж, заполненный:
 - основной частью версии Python;
 - дополнительной частью;
 - номером версии исправлений.

Наш пример программы:

```
1 from platform import python_implementation, python_version_tuple
2
3 print(python_implementation())
4
5 for atr in python_version_tuple():
6     print(atr)
```

выдает следующий результат:

```
CPython  
3  
6  
4
```

Весьма вероятно, что ваша версия Python будет другой.

Каталог модулей Python

Мы здесь рассмотрели только основные модули Python. Модули Python составляют свою собственную вселенную, в которой сам Python является всего лишь галактикой, и мы бы рискнули сказать, что глубокое изучение этих модулей может занять значительно больше времени, чем знакомство с «чистым» Python.

Более того, сообщество Python по всему миру создает и поддерживает сотни дополнительных модулей, используемых в очень специализированных приложениях, таких как генетика, психология или даже астрология.

Эти модули не распространяются (и не будут распространяться) вместе с Python или по официальным каналам, что делает вселенную Python более обширной, почти бесконечной.

Вы можете прочитать обо всех стандартных модулях Python здесь: <https://docs.python.org/3/py-modindex.html>.

Не волнуйтесь — вам не понадобятся все эти модули. Многие из них очень специфичны.

Все, что вам нужно сделать, это найти модули, которые вы хотите, и научиться их использовать. Это просто.

Python » English 3.7.2rc1 Documentation »

Python Module Index

[_](#) | [a](#) | [b](#) | [c](#) | [d](#) | [e](#) | [f](#) | [g](#) | [h](#) | [i](#) | [j](#) | [k](#) | [l](#) | [m](#) | [n](#) | [o](#) | [p](#) | [q](#) | [r](#) | [s](#) | [t](#) | [u](#) | [v](#) | [w](#) | [x](#) | [z](#)

_	
__future__	<i>Future statement definitions</i>
__main__	<i>The environment where the top-level script is run.</i>
_dummy_thread	<i>Drop-in replacement for the <code>_thread</code> module.</i>
_thread	<i>Low-level threading API.</i>
a	
abc	<i>Abstract base classes according to PEP 3119.</i>
aifc	<i>Read and write audio files in AIFF or AIFC format.</i>
argparse	<i>Command-line option and argument parsing library.</i>
array	<i>Space efficient arrays of uniformly typed numeric values.</i>
ast	<i>Abstract Syntax Tree classes and manipulation.</i>
asynchat	<i>Support for asynchronous command/response protocols.</i>
asyncio	<i>Asynchronous I/O.</i>
asyncore	<i>A base class for developing asynchronous socket handling services.</i>
atexit	<i>Register and execute cleanup functions.</i>
audioop	<i>Manipulate raw audio data.</i>
b	
base64	<i>RFC 3548: Base16, Base32, Base64 Data Encodings; Base85 and Ascii85</i>

В следующем разделе мы рассмотрим кое-что еще. Мы хотим показать вам, как написать свой собственный модуль.

Модули и пакеты

Что такое пакет?

Написание собственных модулей мало чем отличается от написания обычных скриптов.

Есть некоторые конкретные аспекты, о которых вы должны знать, но это определенно не ракетостроение. Вы увидите это достаточно скоро.

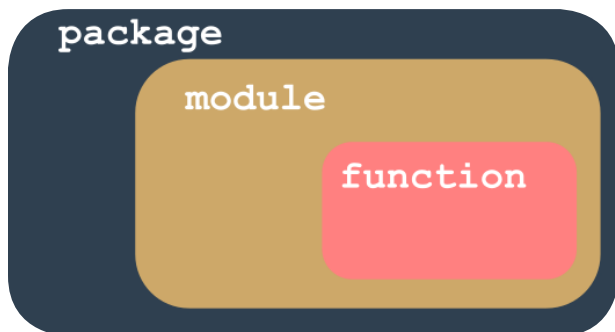


Рисунок 8

Давайте подведем итог некоторым важным вопросам:

- **Модуль** — это своего рода контейнер, заполненный функциями — вы можете упаковать столько функций, сколько хотите, в один модуль и распространять его по всему миру;
- Конечно, как правило, лучше не смешивать функции с различными областями применения в одном модуле (как в библиотеке — никто не ожидает, что научные работы будут помещены в раздел комиксов), поэтому тщательно сгруппируйте свои функции

и назовите модуль, содержащий их, четким и интуитивно понятным способом (например, не давайте имя `arcade_games` модулю, содержащему функции, предназначенные для разделения и форматирования жестких дисков);

- Создание множества модулей может привести к беспорядку — рано или поздно вы захотите сгруппировать свои модули точно так же, как вы ранее сгруппировали функции — есть ли более общий контейнер, чем модуль?
- Да, есть — это пакет; в мире модулей пакет играет ту же роль, что и папка/каталог в мире файлов.

Ваш первый модуль

В этом разделе вы будете работать на вашей машине локально. Давайте начнем с нуля, вот так:

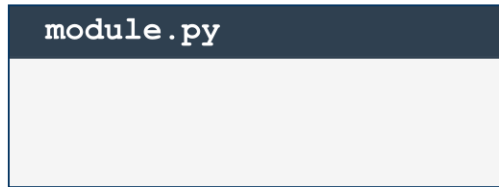


Рисунок 9

Вам нужно два файла, чтобы повторить эти эксперименты. Одним из них будет сам модуль. Сейчас он пустой. Не волнуйтесь, вы будете заполнять его реальным кодом.

Мы назвали файл `module.py`. Не очень креативно, но просто и понятно.

Второй файл содержит код с использованием нового модуля. Его зовут `main.py`.

Его содержимое пока очень краткое:

```
main.py
import module
```

Рисунок 10

Примечание: оба файла должны находиться в одной папке. Мы настоятельно рекомендуем вам создать новую папку для обоих файлов. В таком случае вам будет легче выполнять некоторые действия.

Запустите IDLE и запустите файл `main.py`. Что вы видите? Вы ничего не должны видеть. Это означает, что Python успешно импортировал содержимое файла `module.py`. Неважно, что модуль пока пуст. Самый первый шаг был сделан, но прежде чем вы перейдете к следующему шагу, мы хотим, чтобы вы взглянули на папку, в которой существуют оба файла.

Вы замечаете что-то интересное? Появилась новая подпапка? Ее имя `__pycache__`. Загляните внутрь. Что вы видите? Внутри есть файл с именем (примерно) `module.cpython-ху.pyс`, где `х` и `у` — цифры, производные от вашей версии Python (например, они будут 3 и 4, если вы используете Python 3.4).

Имя файла совпадает с именем вашего модуля (здесь — `module`). Часть после первой точки говорит, какая реализация Python создала файл (здесь — CPython) и номер версии. Последняя часть (`pyс`) происходит от слов *Python* и *compiled*.

Вы можете заглянуть внутрь файла — содержимое совершенно нечитаемое для людей. Так и должно быть,

поскольку файл предназначен только для использования самим Python.

Когда Python импортирует модуль в первый раз, он переводит его содержимое в некую скомпилированную форму. Файл не содержит машинного кода — это внутренний полускомпилированный код Python, готовый к выполнению интерпретатором Python. Поскольку такой файл не требует большого количества проверок, необходимых для чистого исходного файла, выполнение запускается быстрее и также происходит быстрее.

Благодаря этому каждый последующий импорт будет выполняться быстрее, чем интерпретация исходного текста с нуля. Python может проверить, был ли изменен исходный файл модуля (в этом случае файл *рус* будет перестроен) или нет (когда файл *рус* может быть запущен сразу). Поскольку этот процесс полностью автоматический и прозрачный, вам не нужно об этом думать.

Ваш первый модуль: продолжение

Теперь мы добавили кое-что в файл модуля:

```
module.py
```

```
print("I like to be a module.")
```

Рисунок 11

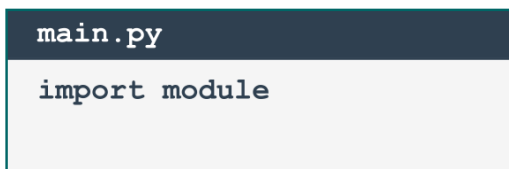
Можете ли вы заметить разницу между модулем и скриптом? Пока ее нет.

Этот файл можно запустить как любой другой скрипт. Попробуйте сами.

Что происходит? Вы должны увидеть следующую строку внутри вашей консоли:

```
I like to be a module.
```

Вернемся к файлу `main.py`:



```
main.py
import module
```

Рисунок 12

Запустить его. Что вы видите? Надеюсь, вы видите что-то вроде этого:

```
I like to be a module.
```

Что это на самом деле означает?

Когда модуль импортируется, его содержимое неявно выполняется Python. Это дает модулю возможность инициализировать некоторые из его внутренних аспектов (например, он может назначать некоторым переменным важные значения).

Примечание: инициализация происходит только один раз, когда происходит первый импорт, поэтому назначения, выполненные модулем, не повторяются без необходимости.

Представьте себе следующий контекст:

- есть модуль с именем `mod1`;
- есть модуль с именем `mod2`, который содержит инструкцию `import mod1`;

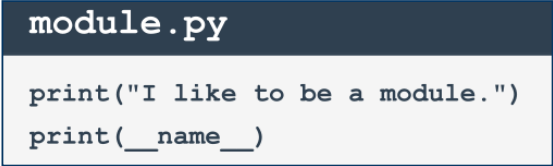
- есть основной файл, содержащий инструкции `import mod1` и `import mod2`.

На первый взгляд вы можете подумать, что `mod1` будет импортирован дважды — к счастью, происходит только первый импорт. Python запоминает импортированные модули и молча пропускает все последующие импорты.

Python может сделать гораздо больше. Он также создает переменную с именем `__name__`.

Более того, каждый исходный файл использует свою собственную, отдельную версию переменной — она не распространяется между модулями.

Мы покажем вам, как его использовать. Немного изменим модуль:

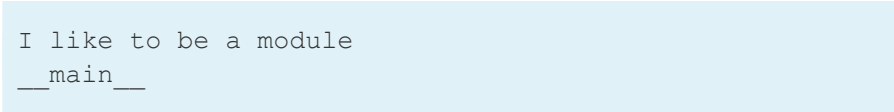


```
module.py

print("I like to be a module.")
print(__name__)
```

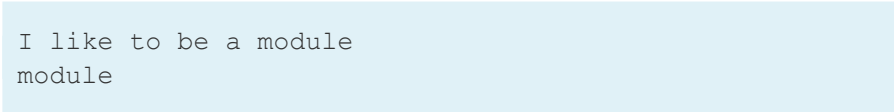
Рисунок 13

Теперь запускаем файл `module.py`. Вы должны увидеть следующие строки:



```
I like to be a module
__main__
```

Теперь запускаем файл `main.py`. И? Вы видите то же, что и мы?



```
I like to be a module
module
```

Мы можем сказать, что:

- Когда вы запускаете файл напрямую, его переменная `__name__` устанавливается в `__main__`;
- когда файл импортируется как модуль, его переменной `__name__` присваивается имя файла (исключая.py)

Вот как вы можете использовать переменную `__main__` для определения контекста, в котором был активирован ваш код:

`module.py`

```
if __name__ == "__main__":  
    print("I prefer to be a module")  
else:  
    print("I like to be a module")
```

Рисунок 14

Однако есть более умный способ использования переменной. Если вы напишете модуль, заполненный множеством сложных функций, вы можете использовать его, чтобы разместить ряд тестов для проверки должной работы функций.

Каждый раз, когда вы изменяете любую из этих функций, вы можете просто запустить модуль, чтобы убедиться, что ваши изменения не испортили код. Эти тесты будут пропущены, когда код будет импортирован как модуль.

Этот модуль будет содержать две простые функции, и если вы хотите узнать, сколько раз эти функции были вызваны, вам потребуется счетчик, инициализирован-

ный в ноль при импорте модуля. Вы можете сделать это следующим образом:

module.py

```
counter = 0

if __name__ == "__main__":
    print("I prefer to be a module")
else:
    print("I like to be a module")
```

Рисунок 15

Введение такой переменной абсолютно верное решение, но может вызвать важные побочные эффекты, о которых вы должны знать.

Взгляните на измененный файл [main.py](#):

main.py

```
import module
print(module.counter)
```

Рисунок 16

Как видите, основной файл пытается получить доступ к переменной счетчика модуля. Разве это допустимо? Да, именно так. Можно использовать? Это может быть очень полезным. Это безопасно? Это зависит от ситуации — если вы доверяете пользователям вашего модуля, проблем нет; однако вы можете не захотеть, чтобы остальной мир видел вашу личную/приватную переменную.

В отличие от многих других языков программирования, Python не позволяет скрыть такие переменные от глаз пользователей модуля. Вы можете только сообщить

своим пользователям, что это ваша переменная, что они могут ее прочитать, но они не должны изменять ее ни при каких обстоятельствах.

Это делается путем добавления к имени переменной `_` (одно подчеркивание) или `__` (два подчеркивания), но помните, что это всего лишь соглашение. Пользователи вашего модуля могут следовать ему или нет.

Мы, конечно, будем следовать соглашению. Теперь давайте поместим две функции в модуль — они будут оценивать сумму и произведение чисел, собранных в списке.

Кроме того, давайте добавим туда несколько декораций и удалим все лишние остатки.

Модуль готов:

module.py

```
#!/usr/bin/env python3

""" module.py - an example of Python module """

__counter = 0

def sum1(list):
    global __counter
    __counter += 1
    sum = 0
    for el in list:
        sum += el
    return sum

def prod1(list):
    global __counter
    __counter += 1
    prod = 1
    for el in list:
        prod *= el
    return prod
```

Рисунок 17а

```

if __name__ == "__main__":
    print("I prefer to be a module, but I can do some tests for you")
    l = [i+1 for i in range(5)]
    print(suml(l) == 15)
    print(prodl(l) == 120)

```

Рисунок 17b

Вероятно, несколько элементов нуждаются в некотором объяснении:

- строка, начинающаяся с `#!`, имеет много имен — в английском она может называться `shabang`, `shebang`, `hash bang`, `poundbang` или даже `hashpling`, в русском же это только шебанг (даже не спрашивайте почему). Само название здесь ничего не значит, значение имеет лишь ее роль. С точки зрения Python, это просто комментарий, который начинается с `#`. Для Unix и Unix-подобных ОС (включая MacOS) такая строка инструктирует ОС, как выполнять содержимое файла (другими словами, какую программу необходимо запустить для интерпретации текста). В некоторых средах (особенно в тех, которые связаны с веб-серверами) отсутствие этой строки вызовет проблемы;
- строка (возможно, многострочная), помещаемая перед любыми инструкциями модуля (включая импорт), называется строкой документации (`doc-string`) и должна кратко объяснить назначение и содержание модуля;
- функции, определенные внутри модуля (`suml()` и `prodl()`) доступны для импорта;
- мы использовали переменную `__name__`, чтобы определить, когда файл запускается автономно, и воспользовались этой возможностью, чтобы выполнить несколько простых тестов.

Теперь можно использовать новый модуль — это один из способов:

```
main.py

from module import sum1, prod1

zeroes = [0 for i in range(5)]
ones = [1 for i in range(5)]
print(sum1(zeroes))
print(prod1(ones))
```

Рисунок 18

Пришло время усложнить этот пример — мы предположили, что основной файл Python находится в той же папке/каталоге, что и импортируемый модуль.

Давайте оставим это предположение и проведем следующий мысленный эксперимент:

- мы используем ОС Windows® (это предположение важно, так как от него зависит форма имени файла)
- основной скрипт Python находится в *C:\Users\user\py\progs* и называется *main.py*.
- модуль для импорта находится в *C:\Users\user\py\modules*.

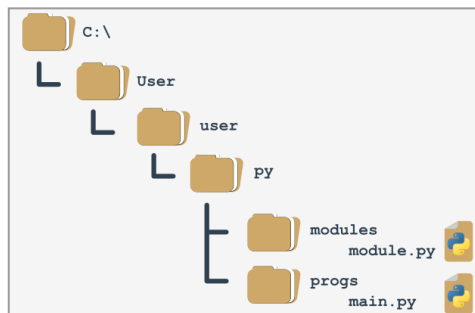


Рисунок 19

Как с этим бороться?

Чтобы ответить на этот вопрос, нам нужно поговорить о том, как Python ищет модули. Существует специальная переменная (фактически список), в которой хранятся все местоположения (папки/каталоги), в которых выполняется поиск, чтобы найти модуль, который был запрошен инструкцией импорта.

Python просматривает эти папки в том порядке, в котором они перечислены в списке — если модуль не может быть найден ни в одном из этих каталогов, импорт завершается неудачно. В противном случае будет принята во внимание первая папка, содержащая модуль с нужным именем (если в любой из оставшихся папок содержится модуль с таким именем, она будет проигнорирована).

Переменная называется `path`, и она доступна через модуль с именем `sys`. Вот как вы можете проверить ее обычное значение:

Мы запустили код в папке `C:\User\user` и получили:

```
C:\Users\user C:\Users\user\AppData\Local\Programs\
Python\Python36-32\python36.zip C:\Users\user\
AppData\Local\Programs\Python\Python36-32\DLLs C:\
Users\user\AppData\Local\Programs\Python\Python36-32\
lib C:\Users\user\AppData\Local\Programs\Python\
Python36-32 C:\Users\user\AppData\Local\Programs\
Python\Python36-32\lib\site-packages
```

Примечание: папка, в которой начинается выполнение, указана в элементе первого пути.

Еще раз обратите внимание: в качестве одного из элементов пути указан zip-файл — это не ошибка. Python

может обрабатывать zip-файлы как обычные папки — это может сэкономить много места.

Можете ли вы выяснить, как мы можем решить эту проблему?

Вы можете решить эту проблему, добавив папку, содержащую модуль, в переменную пути (она полностью изменяема).

Одно из нескольких возможных решений выглядит следующим образом:

```
main.py

from sys import path

path.append('..\modules')

import module

zeroes = [0 for i in range(5)]
ones = [1 for i in range(5)]
print(module.sum1(zeroes))
print(module.prod1(ones))
```

Рисунок 20

Примечание:

- Мы дублировали `\` внутри имени папки — знаете почему? Ответ: поскольку обратная косая черта используется для экранирования других символов — если вы хотите получить только обратную косую черту, вы должны экранировать ее.
- Мы использовали относительное имя папки — это будет работать, если вы запускаете файл

`main.py` непосредственно из его домашней папки, и не будет работать, если текущий каталог не соответствует относительному пути; вы всегда можете использовать абсолютный путь, например так: `path.append('C:\\Users\\user\\py\\modules')`

- Мы использовали метод `append()` — по сути, новый путь будет занимать последний элемент в списке путей; если вам не нравится эта идея, вы можете использовать вместо нее `insert()`.

Ваш первый пакет

Представьте себе, что в недалеком будущем вы и ваши коллеги напишите большое количество функций Python.

Ваша команда решает сгруппировать функции в отдельные модули, и это будет окончательным результатом упорядочивания:

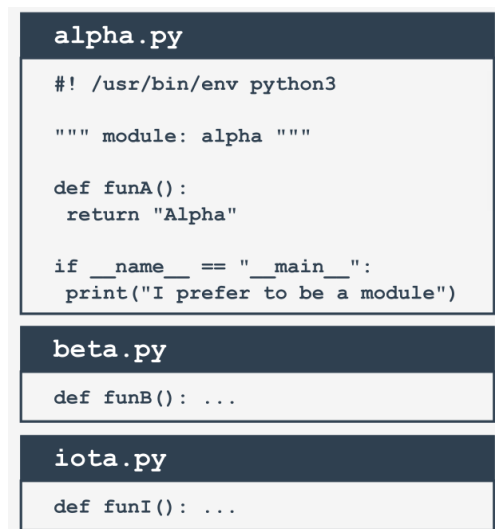


Рисунок 21а

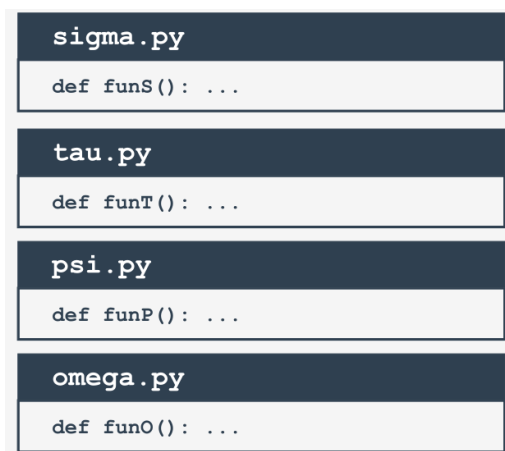


Рисунок 21b

Примечание: мы представили весь контент только для модуля *отега* — предположим, что все модули выглядят одинаково (они содержат одну функцию с именем *funX*, где *X* — первая буква имени модуля).

Внезапно кто-то замечает, что эти модули формируют свою собственную иерархию, поэтому помещать их все в линейную структуру не будет хорошей идеей.

После некоторого обсуждения команда приходит к выводу, что модули должны быть сгруппированы. Все участники согласны с тем, что следующая древовидная структура прекрасно отражает взаимоотношения между модулями. Вот как выглядит дерево в данный момент (рис. 23).

Такая структура почти является пакетом (в смысле Python). Ему не хватает мелких деталей, чтобы обеспечивать функциональность и оперативность. Мы сделаем это далее.

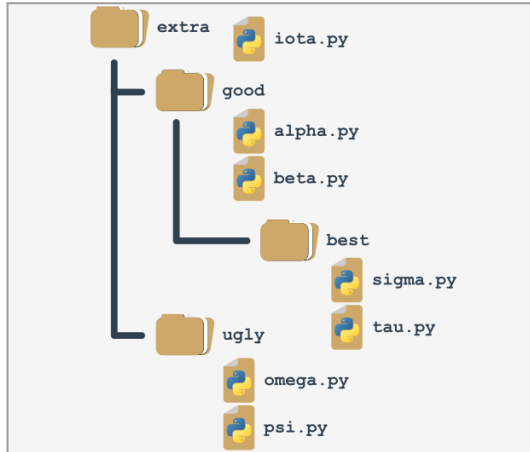


Рисунок 23

Если вы предполагаете, что `extra` — это имя только что созданного пакета (представьте, что это корень пакета), это налагает правило именования, которое позволяет вам четко называть каждую сущность из дерева.

Например:

- расположение функции `funT()` из пакета `tau` может быть описано как: `extra.good.best.tau.funT()`;
- функция, помеченная как: `extra.ugly.psi.funP()`;
- идет от модуля `psi`, хранящегося в подпакете `ugly` пакета `extra`.

Есть два вопроса требующих ответа:

- как вы преобразуете такое дерево (на самом деле, поддерево) в реальный пакет Python (другими словами, как вы убеждаете Python в том, что такое дерево — это не просто кучка остаточных файлов, а набор модулей)?
- куда вы помещаете поддерево, чтобы обеспечить его доступность для Python?

На первый вопрос есть неожиданный ответ: пакеты, как и модули, могут требовать инициализации.

Инициализация модуля выполняется с помощью несвязанного кода (не являющегося частью какой-либо функции), расположенного внутри файла модуля. Поскольку пакет не является файлом, этот метод бесполезен для инициализации пакетов.



Рисунок 24

Вместо этого вам нужно использовать другой прием — Python ожидает, что в папке пакета есть файл с очень уникальным именем: `__init__.py`. Содержимое файла выполняется при импорте любого из модулей пакета. Если вы не хотите никаких особых инициализаций, вы можете оставить файл пустым, но вы не должны его пропускать.

Давайте рассмотрим это снизу вверх (рис. 24):

- группа **ugly** содержит два модуля: **psi** и **omega**;
- группа **best** содержит два модуля: **sigma** и **tau**;
- группа **good** содержит два модуля (**alpha** и **beta**) и одну подгруппу (**best**)
- группа **extra** содержит две подгруппы (**good** и **bad**) и один модуль (**iota**)

Выглядит плохо? Вовсе нет — проанализируйте структуру. Это похоже на структуру каталогов, не так ли? (рис. 25).

Наличие файла `__init__.py`, наконец, составляет пакет:

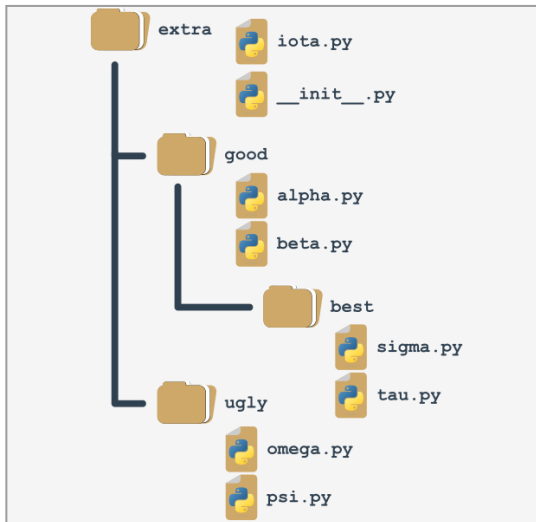


Рисунок 25

Примечание: не только корневая папка может содержать файл `__init__.py` — вы также можете поместить его в любую из ее подпапок (подпакетов). Это может быть полезно, если некоторые из подпакетов требуют индивидуальной обработки и специальных видов инициализации.

Теперь пришло время ответить на второй вопрос — ответ прост: где угодно. Вам нужно только убедиться, что Python знает о местонахождении пакета. Вы уже знаете, как это сделать.

Вы готовы использовать свой первый пакет.

Предположим, что рабочая среда выглядит следующим образом:

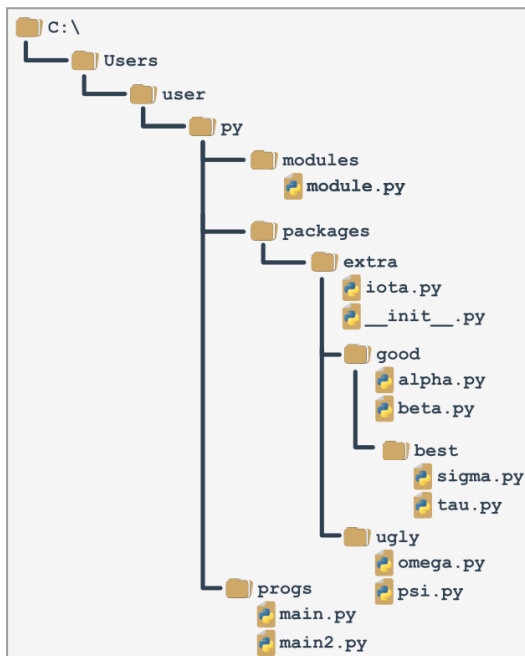


Рисунок 26

Мы подготовили zip-файл, содержащий все файлы из ветки пакетов (он прикреплен к *.pdf* файлу этого урока, для доступа к нему урок необходимо открыть в программе Adobe Acrobat Reader). Вы можете скачать его и использовать для своих собственных экспериментов, но не забудьте распаковать его в папку, представленную на схеме, иначе он не будет доступен для кода из основного файла.

Вы будете продолжать свои эксперименты, используя файл [main2.py](#).

Мы собираемся получить доступ к функции [funI\(\)](#) из модуля [iota](#) в верхней части пакета [extra](#). Нам придется использовать квалифицированные имена пакетов (свяжите это с именами папок и подпапок — соглашения очень похожи).

Вот как это сделать:

```
main2.py

from sys import path

path.append('..\packages')

import extra.iota

print(extra.iota.funI())
```

Рисунок 27

Примечание:

- мы изменили переменную *path*, чтобы сделать ее доступной для Python;
- *import* не указывает непосредственно на модуль, но указывает полный путь от вершины пакета;

- Замена `import extra.iota` на `import iota` вызовет ошибку.
- Также допустим следующий вариант:

main2.py

```
from sys import path

path.append('..\packages')

from extra.iota import funI

print(funI())
```

Рисунок 28

Обратите внимание на квалифицированное имя модуля `iota`.

Теперь давайте дойдем до самого конца дерева — вот как получить доступ к модулям `sigma` и `tau`.

main2.py

```
from sys import path

path.append('..\packages')

import extra.good.best.sigma
from extra.good.best.tau import funT

print(extra.good.best.sigma.funS())
print(funT())
```

Рисунок 29

Вы можете сделать свою жизнь проще, используя псевдонимы:

main2.py

```
from sys import path

path.append('..\packages')

import extra.good.best.sigma as sig
import extra.good.alpha as alp

print(sig.funS())
print(alp.funA())
```

Рисунок 30

Давайте предположим, что мы заархивировали весь подкаталог, начиная с папки **extra** (включая ее), и давайте получим файл с именем **extrapack.zip**. Затем мы помещаем файл в папку **packages**.

Теперь мы можем использовать zip-файл в роли пакетов:

main2.py

```
from sys import path

path.append('..\packages')

import extra.good.best.sigma as sig
import extra.good.alpha as alp

print(sig.funS())
print(alp.funA())
```

Рисунок 31

Если вы хотите провести свои собственные эксперименты с пакетом, который мы создали, вы можете ска-

чать его (файл *Extrapack.ZIP* прикреплен к *.pdf* файлу этого урока).

Теперь вы можете создавать модули и объединять их в пакеты. Пришло время начать совершенно другое обсуждение — об ошибках, сбоях и поломка.

Ошибки — программист

Ошибки, сбои и другие неприятности

«Всё, что может пойти не так, пойдет не так» — это закон Мерфи, и он работает везде и всегда. Выполнение вашего кода тоже может пойти не так. Если это возможно, так и будет.

Посмотрите код ниже.

```
1 import math
2
3 x = float(input("Enter x: "))
4 y = math.sqrt(x)
5
6 print("The square root of", x, "equals to", y)
```

Есть по крайней мере два возможных варианта того, что может пойти не так. Вы видите их?

- Поскольку пользователь может ввести совершенно произвольную строку символов, нет гарантии, что строка может быть преобразована в значение с плавающей запятой — это первая уязвимость кода;
- во-вторых, функция `sqrt()` завершается ошибкой, если получает отрицательный аргумент.

Вы можете получить одно из следующих сообщений об ошибке. Что-то вроде этого:

```
Enter x: Abracadabra
Traceback (most recent call last):
  File "sqrt.py", line 3, in
    x = float(input("Enter x: "))
ValueError: could not convert string to float:'Abracadabra'
```

Или что-то подобное:

```
Enter x: -1
Traceback (most recent call last):
  File "sqrt.py", line 4, in
    y = math.sqrt(x)
ValueError: math domain error
```

Можете ли вы защитить себя от таких неожиданно-стей? Конечно да, более того, вы должны сделать это, чтобы считаться хорошим программистом.

Исключения

Каждый раз, когда ваш код пытается что-то сделать неправильное/глупое/безответственное/безумное/ неосуществимое, Python делает две вещи:

- останавливает вашу программу;
- создает особый вид данных, называемый исключением.

Оба эти действия называются генерацией исключения. Мы можем сказать, что Python всегда генерирует исключение (или что исключение было сгенерировано), когда он не знает, что делать с вашим кодом.

Что произойдет дальше?

- сгенерированное исключение ожидает, что кто-то или что-то заметит его и примет меры;
- если никто или ничто не принимает меры относительно сгенерированного исключения, программа будет принудительно завершена, и вы увидите сообщение об ошибке, отправленное на консоль Python;
- в противном случае, если приняты соответствующие меры и исключение обработано должным образом,

приостановленная программа может быть возобновлена и ее выполнение может быть продолжено.

Python предоставляет эффективные инструменты, которые позволяют вам прослеживать исключения, идентифицировать их и эффективно их обрабатывать. Это возможно благодаря тому, что все потенциальные исключения имеют свои однозначные имена, поэтому вы можете классифицировать их и реагировать соответствующим образом.

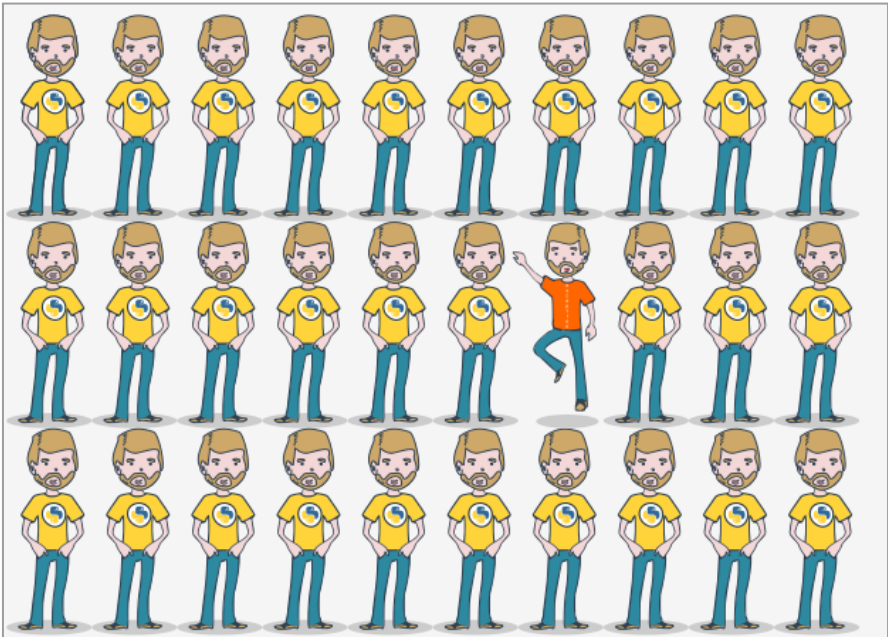


Рисунок 32

Вы уже знаете некоторые имена исключений. Посмотрите на следующее сообщение диагностики:

```
ValueError: math domain error
```

Слово, выделенное красным, — это просто имя исключения. Давайте познакомимся с некоторыми другими исключениями.

Посмотрите на код ниже.

```
1 value = 1  
2 value /= 0
```

Запустите (заведомо неверную) программу. В ответ вы увидите следующее сообщение:

```
Traceback (most recent call last):  
File "div.py", line 2, in  
value /= 0  
ZeroDivisionError: division by zero
```

Эта ошибка исключения называется **ZeroDivisionError**. Посмотрите на код ниже.

```
1 list = []  
2 x = list[0]
```

Что будет, когда вы запустите его? В ответ вы увидите следующее сообщение:

```
Traceback (most recent call last):  
File "lst.py", line 2, in  
x = list[0]  
IndexError: list index out of range
```

Это ошибка **IndexError**.

Как же обрабатывать исключения? Решением этой проблемы является ключевое слово **try**.

Какой алгоритм действий используется:

- во-первых, вы пытаетесь что-то сделать;
- затем проверяете, все ли прошло хорошо.

Но не лучше ли сначала проверить все обстоятельства, а потом, если это безопасно совершать действие?

Как в этом примере.

```

1 firstNumber = int(input("Enter the first number: "))
2 secondNumber = int(input("Enter the second number: "))
3
4 if secondNumber != 0:
5     print(firstNumber / secondNumber)
6 else:
7     print("This operation cannot be done.")
8
9 print("THE END.")

```

Этот способ может показаться наиболее естественным и понятным, но на самом деле это не облегчает программирование. Все эти проверки могут сделать код раздутым и нечитабельным. Python предпочитает другой подход. Посмотрите на код ниже:

```

1 firstNumber = int(input("Enter the first number: "))
2 secondNumber = int(input("Enter the second number: "))
3
4 try:
5     print(firstNumber / secondNumber)
6 except:
7     print("This operation cannot be done.")
8
9 print("THE END.")

```

Это любимый подход Python.

Примечание:

- ключевое слово *try* начинает блок кода, который может работать или не работать правильно;

- затем Python пытается выполнить рискованное действие; в случае неудачи возникает исключение, и Python начинает искать решение;
- ключевое слово `except` запускает фрагмент кода, который будет выполнен, если что-то внутри блока `try` пойдет не так — если исключение возникнет внутри предыдущего блока `try`, то оно здесь не будет выполнено, поэтому код, расположенный после ключевого слова исключения, должен обеспечить надлежащее реагирование на сгенерированное исключение;
- возврат к предыдущему уровню вложения завершает раздел `try-except`.

Запустите код представленный выше и проверьте его поведение.

Давайте подведем итоги:

```
try:
:
:
except:
:
:
```

- на первом этапе Python пытается выполнить все инструкции, помещенные между операторами `try:` и `except:` ;
- если с выполнением все в порядке и все инструкции выполняются успешно, выполнение переходит к точке после последней строки блока `except:`, и выполнение блока считается завершенным;
- если что-то пойдет не так внутри блока `try:` и `except:`, выполнение сразу же выйдет из блока в первую инструк-

цию, расположенную после ключевого слова `except:`; это означает, что некоторые инструкции из блока могут быть просто пропущены.

Посмотрите на код ниже.

```
1 try:
2     print("1")
3     x = 1 / 0
4     print("2")
5 except:
6     print("Oh dear, something went wrong...")
7
8 print("3")
```

Это поможет вам понять этот механизм.

Результат выполнения:

```
1
Oh dear, something went wrong...
3
```

Примечание: инструкция `print(«2»)` была *потеряна в процессе*.

У этого подхода есть один важный недостаток — если есть вероятность, что в ветке `except:` может быть пропущено более одного исключения, у вас могут возникнуть проблемы с выяснением того, что на самом деле произошло.

Как в нашем коде ниже.

```
1 try:
2     x = int(input("Enter a number: "))
3     y = 1 / x
4 except:
5     print("Oh dear, something went wrong...")
6
7 print("THE END.")
```

Запустите его и посмотрите, что произойдет. Сообщение: «*Oh dear, something went wrong...*», которое появляется в консоли, ничего не говорит о причине, в то время как есть две возможные причины исключения:

- нецелочисленные данные, введенные пользователем;
- целочисленное значение, равное 0, присвоенное переменной `x`.

Существует два способа решения проблемы:

- построить два последовательных блока `try-except`, по одному для каждой возможной причины исключения (легко, но приведет к неблагоприятному увеличению кода)
- использовать более продвинутый вариант инструкции.

Посмотрите на это:

```
try:
:
except exc1:
:
except exc2:
:
except:
:
```

Вот как это работает:

- если ветка `try` генерирует исключение `exc1`, оно будет обрабатываться блоком `except exc1;`
- аналогично, если ветка `try` генерирует исключение `exc2`, оно будет обрабатываться блоком `except exc2;`
- если ветка `try` генерирует любое другое исключение, оно будет обработано безымянным блоком `except`.

Давайте перейдем к следующей части курса и увидим его в действии.

Посмотрите на код ниже.

```

1 ▾ try:
2     x = int(input("Enter a number: "))
3     y = 1 / x
4     print(y)
5 ▾ except ZeroDivisionError:
6     print("You cannot divide by zero, sorry.")
7 ▾ except ValueError:
8     print("You must enter an integer value.")
9 ▾ except:
10    print("Oh dear, something went wrong...")
11
12 print("THE END.")

```

Наше решение здесь.

Код при запуске выдает один из следующих четырех вариантов вывода:

- если вы введете правильное ненулевое целочисленное значение (например, 5), он выдает:

```

0.2
THE END.

```

- Если вы вводите 0, он выдает:

```

You cannot divide by zero, sorry.
THE END.

```

- если вы введете любую нецелочисленную строку, вы увидите:

```

You must enter an integer value.
THE END.

```

- (локально на вашем компьютере), если вы нажмете **Ctrl-C**, пока программа ожидает ввода пользователя (что вызывает исключение с именем **KeyboardInterrupt**), программа сообщает:

```
Oh dear, something went wrong...
THE END.
```

Не забывайте, что:

- поиск по веткам **except** осуществляется в том же порядке, в котором они появляются в коде;
- вы не должны использовать более одной ветки исключения с определенным именем исключения;
- количество разных веток **except** произвольно — единственное условие: если вы используете **try**, вы должны поставить после него хотя бы один **except** (именованный или нет);
- ключевое слово **except** не должно использоваться без предшествующего **try**;
- если выполнена какая-либо из веток **except**, никакие другие ветки не будут просмотрены;
- если ни одна из указанных веток **except** не соответствует сгенерированному исключению, исключение остается необработанным (мы скоро обсудим это);
- если существует неназванная ветка **except** (одна без имени исключения), она должна быть указана как последняя.

```
try:
:
```

```
except exc1:
:
except exc2:
:
except:
:
```

Давайте продолжим эксперименты. Посмотрите на код ниже.

```
1 ▾ try:
2     x = int(input("Enter a number: "))
3     y = 1 / x
4     print(y)
5 ▾ except ValueError:
6     print("You must enter an integer value.")
7 ▾ except:
8     print("Oh dear, something went wrong...")
9
10 print("THE END.")
```

Мы изменили предыдущую программу — мы удалили ветку `ZeroDivisionError`.

Что происходит теперь, если пользователь вводит `0` в качестве значения?

Поскольку нет выделенных ветвей для деления на ноль, сгенерированное исключение попадает в общую (неозначенную) ветку; это означает, что в этом случае программа выдаст:

```
Oh dear, something went wrong...
THE END.
```

Попробуйте сами.

Давайте подпортим код еще раз. Посмотрите на программу ниже.

```
1 try:
2     x = int(input("Enter a number: "))
3     y = 1 / x
4     print(y)
5 except ValueError:
6     print("You must enter an integer value.")
7
8 print("THE END.")
```

На этот раз мы удалили неназванную ветку.

Пользователь вводит 0 еще раз и:

- сгенерированное исключение не будет обработано `ValueError` — оно не имеет к этому никакого отношения;
- поскольку нет другой ветки, вы должны увидеть это сообщение:

```
Traceback (most recent call last):
File "exc.py", line 3, in
y = 1 / x
ZeroDivisionError: division by zero
```

Вы узнали многое об обработке исключений в Python. В следующем разделе мы сосредоточимся на встроенных исключениях Python и их иерархиях.

Анатомия исключений

Исключения

Python 3 определяет 63 встроенных исключения, и все они образуют древовидную иерархию, хотя дерево немного странное, поскольку его корень расположен сверху.

Некоторые из встроенных исключений носят более общий характер (они включают в себя другие исключения), в то время как другие являются полностью конкретными (они представляют только себя). Можно сказать, что чем ближе к корню находится исключение, тем оно более общее (абстрактное). В свою очередь, исключения, расположенные на концах ветвей (мы можем назвать их листьями), являются конкретными.

Посмотрите на рисунок:

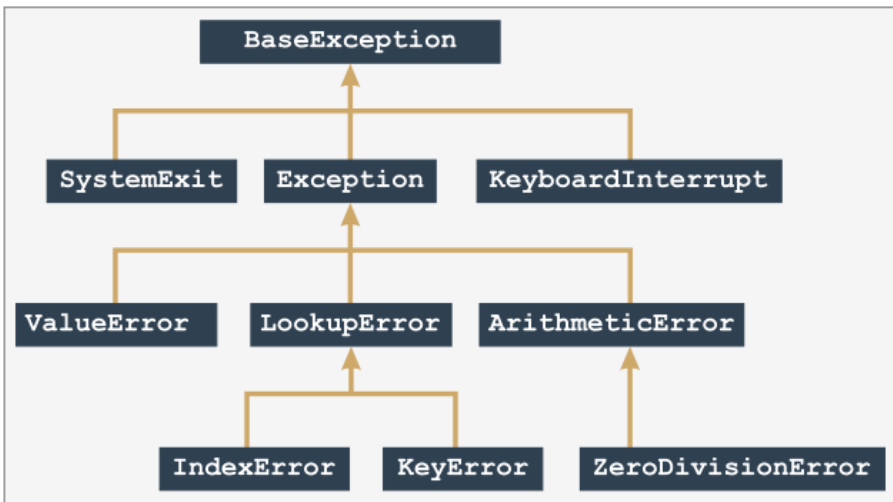


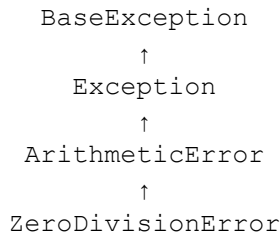
Рисунок 33

Он показывает небольшой раздел полного дерева исключений. Давайте начнем изучение дерева с листа `ZeroDivisionError`.

Примечание:

- *`ZeroDivisionError` является специальным случаем более общего класса исключений с именем `ArithmeticError`;*
- *`ArithmeticError` является специальным случаем более общего класса исключений с именем `Exception`;*
- *`Exception` является специальным случаем более общего класса с именем `BaseException`;*

Мы можем описать это следующим образом (обратите внимание на направление стрелок — они всегда указывают на более общую сущность):



Мы собираемся показать вам, как работает это обобщение. Давайте начнем с очень простого. Посмотрите на код ниже.

```

1 try:
2     y = 1 / 0
3 except ZeroDivisionError:
4     print("Ooopsss...")
5
6 print("THE END.")
  
```

Результат, который мы ожидаем увидеть, выглядит следующим образом:

```
Ooopssss... THE END.
```

Теперь посмотрите следующий код:

```
try:
    y = 1 / 0

except ArithmeticError:
    print("Ooopssss...")

print("THE END.")
```

Что-то изменилось — мы заменили `ZeroDivisionError` на `ArithmeticError`.

Вы уже знаете, что `ArithmeticError` — это общий класс, включающий (среди прочего) исключение `ZeroDivisionError`. Таким образом, вывод кода остается неизменным. Проверьте это.

Это также означает, что замена имени исключения на `Exception` или `BaseException` не изменит поведение программы.

Подведем итоги:

- каждое сгенерированное исключение попадает в первую соответствующую ветку;
- соответствующая ветка не обязательно должна точно указывать одно и то же исключение — достаточно, чтобы исключение было более общим (более абстрактным), чем сгенерированное.

Посмотрите на код.

```
1 try:
2     y = 1 / 0
3 except ZeroDivisionError:
4     print("Zero Division!")
5 except ArithmeticError:
6     print("Arithmetic problem!")
7
8 print("THE END.")
```

Что здесь произойдет?

Первая подходящая ветвь — та, которая содержит `ZeroDivisionError`. Это означает, что консоль покажет:

```
Zero division!
THE END.
```

Изменится ли что-нибудь, если мы поменяемся местами ветки `except`? Как здесь, ниже:

```
try:
    y = 1 / 0
except ArithmeticError:
    print("Arithmetic problem!")
except ZeroDivisionError:
    print("Zero Division!")

print("THE END.")
```

Изменение радикально — вывод кода теперь:

```
Arithmetic problem!
THE END.
```

Почему так, если сгенерированное исключение такое же, как и ранее?

Исключение то же самое, но более общее исключение теперь перечислено первым — оно также будет охватывать все деления на ноль. Это также означает, что нет никаких шансов, что какое-либо исключение попадет в ветку `ZeroDivisionError`. Эта ветка теперь совершенно недостижима.

Запомните:

- порядок веток имеет значение!
- не ставьте более общие исключения перед более конкретными;
- это сделает последний недостижимым и бесполезным;
- кроме того, это сделает ваш код беспорядочным и непоследовательным;
- Python не будет генерировать никаких сообщений об ошибках по этой проблеме.

Если вы хотите обрабатывать два или более исключений одним и тем же способом, вы можете использовать следующий синтаксис:

```
try:
:

except (exc1, exc2):
:
```

Вы просто должны поместить все задействованные имена исключений в список через запятую и не забыть скобки.

Если исключение возникает внутри функции, оно может быть обработано:

- внутри функции;
- вне функции;

Давайте начнем с первого варианта:

```

1 def badFun(n):
2     try:
3         return 1 / n
4     except ArithmeticError:
5         print("Arithmetic Problem!")
6         return None
7
8 badFun(0)
9
10 print("THE END.")

```

Исключение `ZeroDivisionError` (являющееся специальным случаем класса исключений `ArithmeticError`) вызывается внутри функции `badfun()`, и оно не покидает функцию — сама функция сама об этом заботится.

Программа выводит:

```

Arithmetic problem!
THE END.

```

Также есть возможность позволить исключению распространяться за пределы функции. Давайте это проверим:

```

def badFun(n):
    return 1 / n
try:
    badFun(0)

```

```
except ArithmeticError:
    print("What happened? An exception was raised!")

print("THE END.")
```

Проблема должна быть решена инициатором (или инициатором инициатора и т.д.).

Программа выводит:

```
What happened? An exception was raised!
THE END.
```

***Примечание:** сгенерированное исключение может пересекать границы функций и модулей и проходить по цепочке вызовов в поисках соответствующей конструкции `except`, способной его обработать.*

Если такой конструкции нет, исключение остается не-обработанным, и Python решает проблему стандартным способом — прекращая работу вашего кода и отправляя сообщение диагностики.

Теперь мы собираемся приостановить обсуждение этого вопроса, поскольку хотим представить вам совершенно новую инструкцию Python.

Инструкция `raise` генерирует указанное исключение с именем `exc`, как если бы оно было вызвано обычным (естественным) способом:

```
raise exc
```

***Примечание:** `raise` — это ключевое слово.*

Инструкция позволяет вам:

- имитировать генерацию реальных исключений (например, для проверки вашей стратегии обработки)
- частично обработать исключение и возложить на другую часть кода ответственность за завершение обработки (разделение задач).

Посмотрите на код ниже.

```
1 ▾ def badFun(n):  
2     raise ZeroDivisionError  
3  
4 ▾ try:  
5     badFun(0)  
6 ▾ except ArithmeticError:  
7     print("What happened? An error?")  
8  
9     print("THE END.")
```

Вот как вы можете использовать это на практике. Вывод программы остается неизменным.

Таким образом, вы можете протестировать свою процедуру обработки исключений, не заставляя код делать глупости.

Инструкция `raise` также может быть использована следующим образом (обратите внимание на отсутствие названия исключения):

```
raise
```

Есть одно серьезное ограничение: инструкция `raise` в таком виде может использоваться только внутри ветви `except`; использование его в любом другом контексте приводит к ошибке.

Инструкция немедленно повторит генерацию того же исключения, что обрабатывается в настоящий момент времени.

Благодаря этому вы можете распределять обработку исключений по различным частям кода.

Посмотрите на код ниже.

```
1 ▾ def badFun(n):  
2 ▾     try:  
3 ▾         return n / 0  
4 ▾     except:  
5 ▾         print("I did it again!")  
6 ▾         raise  
7  
8 ▾ try:  
9     badFun(0)  
10 ▾ except ArithmeticError:  
11     print("I see!")  
12  
13 print("THE END.")
```

Запустите его — мы увидим это в действии.

`ZeroDivisionError` генерируется дважды:

- сначала внутри части кода `try` (это вызвано фактическим делением на ноль)
- а потом внутри части `except` с помощью инструкции `raise`.

По сути, код выводит:

```
I did it again!  
I see!  
THE END.
```

Сейчас отличный момент, чтобы показать вам еще одну инструкцию Python с именем `assert`. Это ключевое слово.

```
assert expression
```

Как она работает?

- оценивает выражение;
- если выражение оценивается как `True`, или ненулевое числовое значение, или непустая строка, или любое другое значение, отличное от `None`, она больше ничего не будет делать;
- в противном случае она автоматически и сразу генерирует исключение с именем `AssertionError` (в этом случае мы говорим, что утверждение не выполнено).

Как ее можно использовать?

- Возможно вы захотите поместить ее в свой код там, где вы хотите гарантировать отсутствие явно неверных данных, и где вы не совсем уверены, что данные были тщательно проверены ранее (например, внутри функции, используемой кем-то другим)
- генерация исключения `AssertionError` защищает ваш код от получения недопустимых результатов и четко показывает характер ошибки;
- утверждения не отменяют исключения и не проверяют данные — они являются их дополнениями.

Если исключения и проверка данных подобны осторожному вождению, утверждение может играть роль подушки безопасности.

Давайте посмотрим на инструкцию `assert` в действии. Взгляните на код ниже

```
1 import math
2
3 x = float(input("Enter a number: "))
4 assert x >= 0.0
5
6 x = math.sqrt(x)
7
8 print(x)
```

Запустить его.

Программа работает без ошибок, если вы введете допустимое числовое значение, большее или равное нулю; в противном случае она останавливается и выдает следующее сообщение:

```
Traceback (most recent call last):
  File ".main.py", line 4, in
    assert x >= 0.0
AssertionError
```

Полезные исключения

Встроенные исключения

Мы собираемся показать вам краткий список самых полезных исключений. Хотя может показаться странным называть «полезной» вещь или явление, которое является видимым признаком отказа или неудачи, как вы знаете, человеку свойственно ошибаться, и если что-то может пойти не так, оно пойдет не так.

Исключения такие же рутинные и обычные, как и любой другой аспект жизни программиста. Для каждого исключения мы покажем вам:

- его имя;
- его расположение в дереве исключений;
- краткое описание;
- краткий фрагмент кода, показывающий обстоятельства, при которых может возникнуть исключение.

Есть много других исключений для изучения — у нас просто не хватит места, чтобы просмотреть их все здесь.

ArithmeticError

Расположение: `BaseException` ← `Exception` ← `ArithmeticError`.

Описание: абстрактное исключение, включающее в себя все исключения, вызванные арифметическими операциями, такими как деление на ноль или недопустимая область значений аргумента

AssertionError

Расположение: `BaseException` \leftarrow `Exception` \leftarrow `AssertionError`.

Описание: конкретное исключение, сгенерированное инструкцией `assert`, когда ее аргумент оценивается как `False`, `None`, `0`, или пустая строка.

Код:

```
from math import tan, radians
angle = int(input('Enter integral angle in degrees: '))

# мы должны быть уверены, что угол != 90 + k * 180
assert angle % 180 != 90
print(tan(radians(angle)))
```

BaseException

Расположение: `BaseException`.

Описание: самое общее (абстрактное) из всех исключений Python — все остальные исключения включены в это; можно сказать, что следующие две ветки `except` эквивалентны: `except:` и `except BaseException:`.

IndexError

Расположение: `BaseException` \leftarrow `Exception` \leftarrow `LookupError` \leftarrow `IndexError`.

Описание: конкретное исключение, которое генерируется при попытке доступа к несуществующему элементу последовательности (например, элементу списка).

Код:

```
# код показывает необычный способ
# выхода из цикла
```

```
list = [1, 2, 3, 4, 5]
ix = 0
doit = True

while doit:
    try:
        print(list[ix])
        ix += 1
    except IndexError:
        doit = False
print('Done')
```

KeyboardInterrupt

Расположение: `BaseException` ← `KeyboardInterrupt`.

Описание: конкретное исключение, генерируемое, когда пользователь использует сочетание клавиш, предназначенное для прекращения выполнения программы (`Ctrl-C` в большинстве ОС); если обработка этого исключения не приводит к завершению программы, программа продолжает выполнение.

Примечание: это исключение не является производным от класса `Exception`. Запустите программу в IDLE.

Код:

```
# выполнение этого кода нельзя остановить
# нажатие Ctrl-C
from time import sleep
seconds = 0
while True:
    try:
        print(seconds)
```

```
seconds += 1
sleep(1)
except KeyboardInterrupt:
    print("Don't do that!")
```

LookupError

Расположение: `BaseException` \leftarrow `Exception` \leftarrow `LookupError`.

Описание: абстрактное исключение, включая все исключения, вызванные ошибками, возникшими из-за недопустимых ссылок на разные коллекции (списки, словари, кортежи и т.д.).

MemoryError

Расположение: `BaseException` \leftarrow `Exception` \leftarrow `MemoryError`.

Описание: конкретное исключение, генерируемое, когда операция не может быть завершена из-за недостатка свободной памяти.

Код:

```
# этот код вызывает исключение MemoryError
# внимание: выполнение этого кода может быть
# критичным для вашей ОС
# не запускайте его в производственной среде!
string = 'x'
try:
    while True:
        string = string + string
        print(len(string))
except MemoryError:
    print('This is not funny!')
```

OverflowError

Расположение: `BaseException` \leftarrow `Exception` \leftarrow `ArithmeticError` \leftarrow `OverflowError`.

Описание: конкретное исключение, генерируемое, когда операция выдает слишком большое число для успешного хранения.

Код:

```
# код выводит последовательные
# значения exp(k), k = 1, 2, 4, 8, 16, ...
from math import exp
ex = 1
try:
    while True:
        print(exp(ex))
        ex *= 2
except OverflowError:
    print('The number is too big.')
```

ImportError

Расположение: `BaseException` \leftarrow `Exception` \leftarrow `StandardError` \leftarrow `ImportError`.

Описание: конкретное исключение, генерируемое при сбое операции импорта.

Код:

```
# один из этих импортов не сработает — какой?
try:
    import math
    import time
    import abracadabra
except:
    print('One of your imports has failed.')
```


KeyError

Расположение: `BaseException` \leftarrow `Exception` \leftarrow `LookupError` \leftarrow `KeyError`.

Описание: конкретное исключение, генерируемое при попытке доступа к несуществующему элементу коллекции (например, элементу словаря).

Код:

```
# как злоупотреблять словарем
# и как с этим бороться
dict = { 'a' : 'b', 'b' : 'c', 'c' : 'd' }
ch = 'a'
try:
    while True:
        ch = dict[ch]
        print(ch)
except KeyError:
    print('No such key:', ch)
```

Мы пока завершаем рассмотрение исключений, но мы вернемся к ним при обсуждении объектно-ориентированного программирования на Python. Вы можете использовать их для защиты своего кода от несчастных случаев, но вы также должны научиться вникать в них, исследуя информацию, которую они несут.

Исключения на самом деле являются объектами — однако мы ничего не можем вам рассказать об этом аспекте, пока не представим вам классы, объекты и тому подобное.

В настоящее время, если вы хотите больше узнать об исключениях самостоятельно, загляните в Стандартную библиотеку Python по адресу <https://docs.python.org/3.6/library/exceptions.html>.



Урок 6.

Модули, пакеты, исключения

© Компьютерная Академия «Шаг», www.itstep.org

Все права на охраняемые авторским правом фото-, аудио- и видеопроизведения, фрагменты которых использованы в материале, принадлежат их законным владельцам. Фрагменты произведений используются в иллюстративных целях в объёме, оправданном поставленной задачей, в рамках учебного процесса и в учебных целях, в соответствии со ст. 1274 ч. 4 ГК РФ и ст. 21 и 23 Закона Украины «Про авторське право і суміжні права». Объём и способ цитируемых произведений соответствует принятым нормам, не наносит ущерба нормальному использованию объектов авторского права и не ущемляет законные интересы автора и правообладателей. Цитируемые фрагменты произведений на момент использования не могут быть заменены альтернативными, не охраняемыми авторским правом аналогами, и как таковые соответствуют критериям добросовестного использования и честного использования.

Все права защищены. Полное или частичное копирование материалов запрещено. Согласование использования произведений или их фрагментов производится с авторами и правообладателями. Согласованное использование материалов возможно только при указании источника.

Ответственность за несанкционированное копирование и коммерческое использование материалов определяется действующим законодательством Украины.