Функции

Что такое модуль?

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.

Что такое модуль?

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.

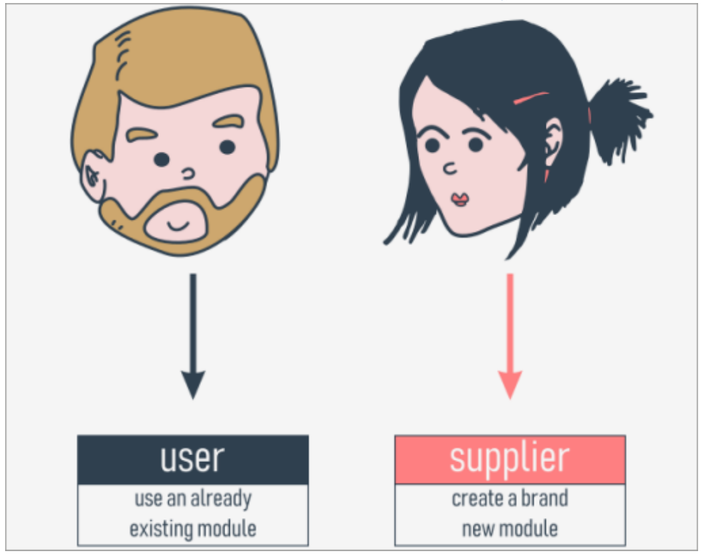
Как разделять элемент программного обеспечения на отдельные, но взаимодействующие части? В этом и вопрос. Ответом являются модули

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.

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».

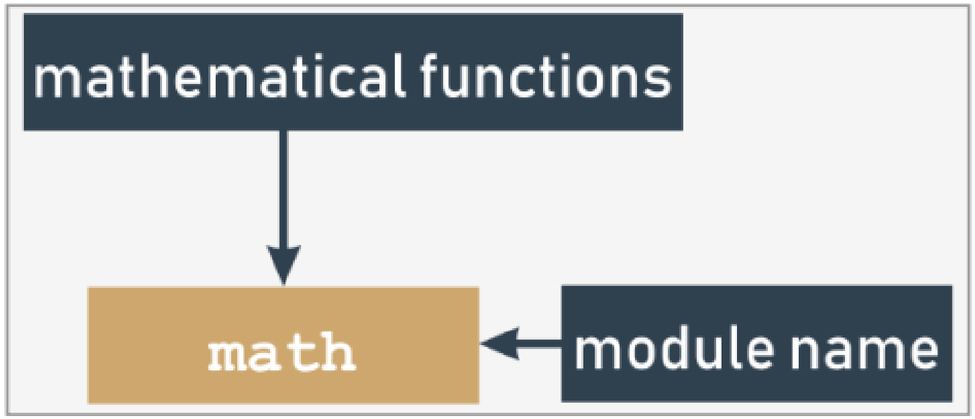


Все эти модули вместе со встроенными функциями образуют стандартную библиотеку Python — особую библиотеку, в которой модули играют роль книг (можно даже сказать, что папки играют роль полок). Если вы хотите взглянуть на полный список всех «томов», собранных в этой библиотеке, вы можете найти его здесь: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

Как использовать модули?

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.

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).

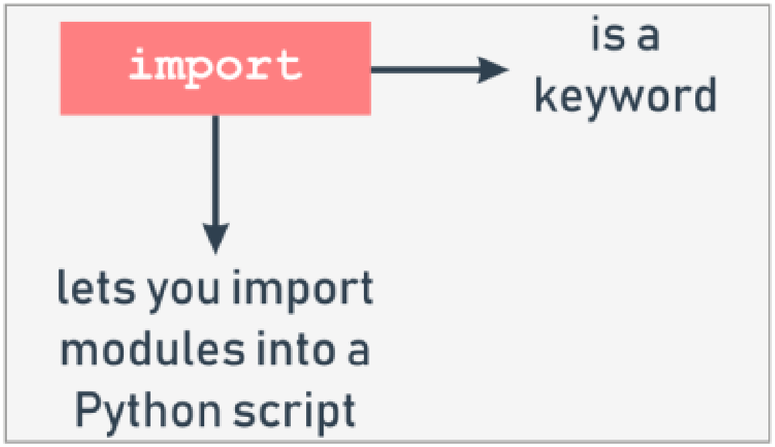
Предположим, вы хотите использовать две сущности, предоставляемые модулем math:

- 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

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



Импортирование модулей

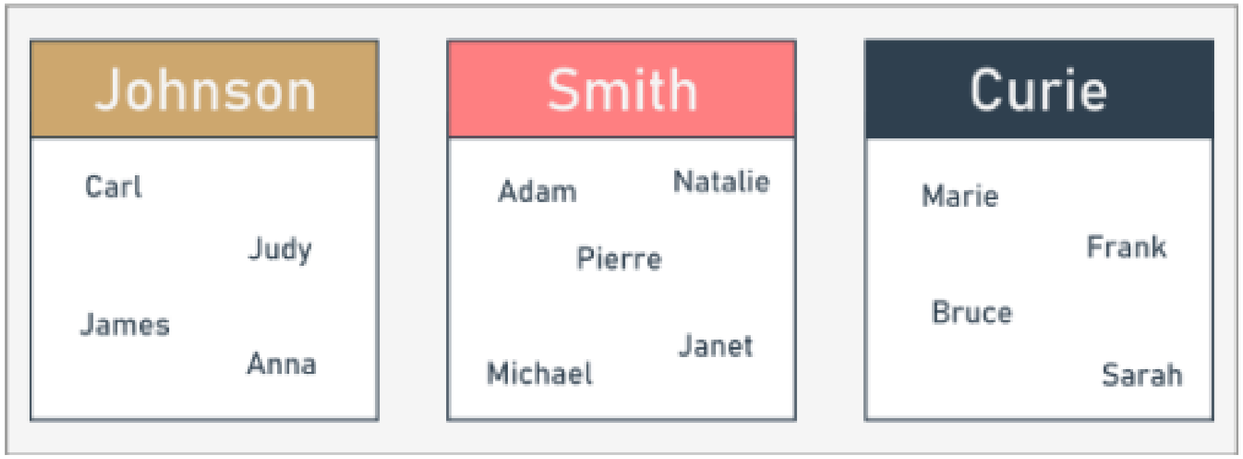
Инструкция может находиться где угодно в вашем коде, но она должна быть размещена перед первым использованием любой из сущностей модуля (*код будет работать при данных условиях, НО это нехорошо с точки зрения правил PEP-8, импорты необходимо производить в самом верху вашего файла с кодом*). Если вы хотите (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.

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



Импортирование модулей

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.

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

Импортирование модулей

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 пишите:

- 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.*

Импортирование модулей

Первый пример не будет очень сложным — мы просто хотим вывести значение sin(1/2π).

```
import math

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

```
>>> 1.0
```

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 посмотрим, как могут сосуществовать два пространства имен (ваше и модуля).

```
import math


def sin(x):
    if 2 * x == pi:
        return 0.9999999999
    else:
        return None


pi = 3.14
print(sin(pi / 2))
print(math.sin(math.pi / 2))
```

```
>>> 0.9999999999
1.0
```

Здесь мы определили наши собственные pi и sin.

Запускаем программу. Код выдает следующий результат:

0.9999999999

1.0

Как видим сущности не влияют друг на друга.

Импортирование модулей

Во втором методе синтаксис **import** точно указывает, какая сущность (или сущности) модуля будут применятся 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...).

Импортирование модулей

Посмотрим как это работает на практике.

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

```
>>> 1.0
```

Результат такой же как и в первом случае, так как на самом деле мы использовали те же сущности, что и раньше.

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

Посмотрим на код ниже и проанализируем его:

```
1   from math import sin, pi
2
3   print(sin(pi / 2))
4
5   pi = 3.14
6
7
8   def sin(x):
9       if 2 * x == pi:
10          return 0.9999999999
11      else:
12          return None
13
14
15  print(sin(pi / 2))
```

```
>>> 1.0
    0.9999999999
```

- *строка 01:* выполнить выборочный импорт;
- *строка 03:* использовать импортированные сущности и получить ожидаемый результат (1.0);
- строки с 05 по 12: переопределить значения pi и sin - фактически они заменяют первоначальные (импортированные) определения в пространстве имен кода;
- строка 13: получить 0.99999999, что подтверждает наши выводы.

Импортирование модулей

Сделаем еще один тест. Посмотрите на следующий код:

```
pi = 3.14

    1

def sin(x):
    if 2 * x == pi:
        return 0.99999999
    else:
        return None

print(sin(pi / 2)) 2

* from math import sin, pi 3

print(sin(pi / 2)) 4
```

```
>>> 0.99999999
1.0
```

мы изменили последовательность операций кода:

1. определить наши собственные pi и sin;
2. использовать их (на экране появляется 0.99999999);
3. выполнить импорт — импортированные символы заменяют свои предыдущие определения в пространстве имен;
4. получить в результате 1.0.

*однако как мы видим from у нас подчеркнуто волнистой чертой, почему? А потому что импорты в середине кода не соответствует PEP 8.

```
PEP 8: E402 module level import not at top of file
```

Импортирование модулей

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

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))
```

```
>>> 1.0
```

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

Импортирование модулей: ключевое слово as

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

Посмотрим как это будет работать:

```
from math import sin as sinus, pi as pi_num

print(sinus(pi_num / 2))
```

```
>>> 1.0
```

Данный метод может показаться немного странным но он работает!

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

Полезные модули

Работа со стандартными модулями

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, предоставленные 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.

Импортирование модулей: ключевое слово as

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")
```

```
>>>
__doc__     __loader__     __name__     __package__     __spec__     acos     acosh
asin     asinh     atan     atan2     atanh     cbrt     ceil     comb     copysign     cos     cosh
degrees     dist     e     erf     erfc     exp     exp2     expm1     fabs     factorial     floor
fmod     frexp     fsum     gamma     gcd     hypot     inf     isclose     isfinite     isinf     isnan     isqrt
lcm     ldexp     lgamma     log     log10     log1p     log2     modf     nan     nextafter     perm
pi     pow     prod     radians     remainder     sin     sinh     sqrt     sumprod     tan
tanh     tau     trunc     ulp
```

Обратите внимание имена начинающиеся с «__» вверху списка. Мы узнаем 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.

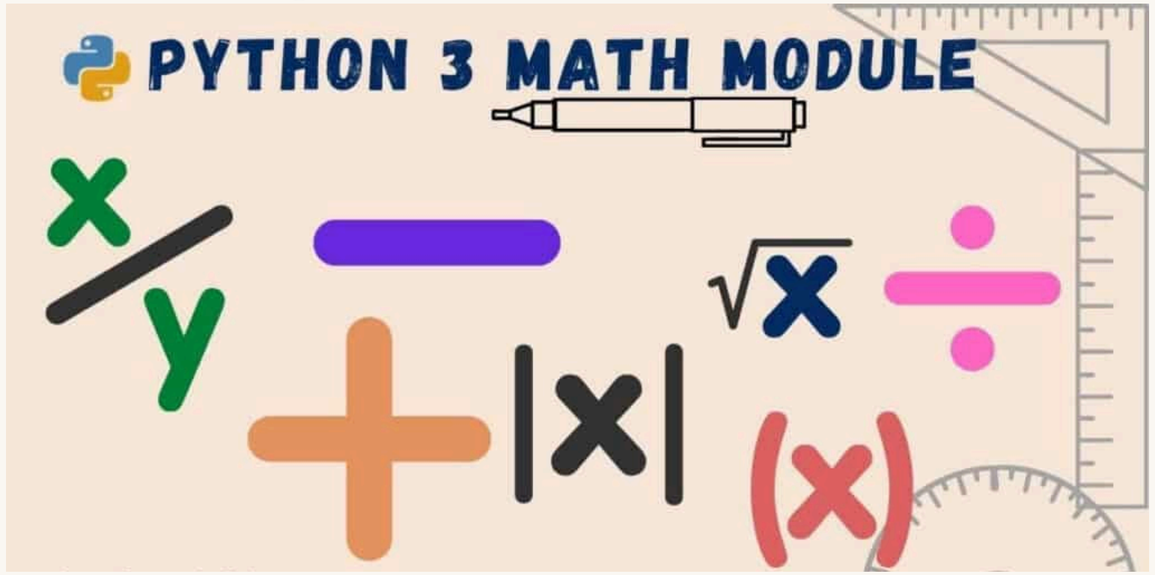
Использование функции dir() внутри кода может показаться не очень полезным — обычно вы хотите знать содержимое определенного модуля, прежде чем писать и запускать код.

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

```
Python Console>>> import math
>>> dir(math)
['__doc__', '__loader__', '__name__', '__package__', '__spec__', 'acos', 'acosh', 'asin', 'asinh', 'atan', 'atan2',
```

все содержимое, соответствует примеру выше.

Некоторые функции из модуля math



Модуль 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:

- **asin(x)** → арксинус **x**;
- **acos(x)** → арккосинус **x**;
- **atan(x)** → арктангенс **x**.

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

Модуль math

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).

Давайте используем эти функции в примере.

```
from math import pi, radians, degrees, sin, cos, tan, asin

ad = 90
ar = radians(ad)
ad = degrees(ar)

print(ad == 90.0)
print(ar == pi / 2)
print(sin(ar) / cos(ar) == tan(ar))
print(asin(sin(ar)) == ar)
```

```
>>> True
    True
    True
    True
```

результаты выполнения кода сообщает нам о верности данных тождеств

Модуль math

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** в степени **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** в степени **y** (обратите внимание на области определения, также **pow(x, y)** встроенная функция, и ее не нужно импортировать.)

Модуль math

Посмотрим как это можно использовать:

```
from math import e, exp, log

print(pow(e, 1))
print(pow(e, 1) == exp(log(e)))

print(pow(2, 2))
print(pow(2, 2) == exp(2 * log(2)))

print(log(e, e))
print(log(e, e) == exp(0))
```

```
>>>
2.718281828459045
True
4
True
1.0
True
```

Модуль math

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

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

```
import math
from math import ceil, floor, trunc

x = 3.4
y = 5.6
print("Верхнее округление", ceil(x), ceil(y))
print("Верхнее округление", ceil(-x), ceil(-y))
print("Нижнее  округление", floor(x), floor(y))
print("Нижнее округление", floor(-x), floor(-y))
print("Усеченное до целого числа", trunc(x), trunc(y))
print("Усеченное до целого числа", trunc(-x), trunc(-y))
print("Факториал", math.factorial(5))
print("Гипотенуза", math.hypot(3, 4))
```

>>>

```
Верхнее округление 4 6
Верхнее округление -3 -5
Нижнее  округление 3 5
Нижнее округление -4 -6
Усеченное до целого числа 3 5
Усеченное до целого числа -3 -5
Факториал 120
Гипотенуза 5.0
```

программа демонстрирует принципиальные различия между **ceil()**, **floor()** и **trunc()**. А также демонстрирует работу **factorial()** и **hypot()**

Библиотека random

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.

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.

Есть ли настоящая случайность в компьютерах?

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 - random, seed

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 <= x < 1.0).

```
from random import random

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

```
0.6507663155884139
0.6000048945762936
>>> 0.7657985459895116
0.7223429567406547
0.12669447200289452
```

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

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**.

```
from random import random, seed

seed(0)

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

```
0.8444218515250481
0.7579544029403025
>>> 0.420571580830845
0.25891675029296335
0.5112747213686085
```

Я много раз запускал этот код, и каждый раз получали именно такой результат. Это потому, что мы изменили предыдущую программу — по сути, мы удалили любые следы случайности из кода.
Из-за того, что начальное число всегда установлено на одно и то же значением, последовательность сгенерированных значений всегда выглядит одинаково.

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.

Некоторые функции из модуля random

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).

```
from random import randrange, randint

print(randrange(1), end=' ')
print(randrange(0, 1), end=' ')
print(randrange(0, 1, 1), end=' ')
print(randint(0, 1), end=' ')
```

```
>>> 0 0 0 1
```

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).

Предыдущие функции имеют один важный недостаток — они могут создавать повторяющиеся значения, даже если число последующих вызовов не превышает ширину указанного диапазона.

Библиотека random, метод .randint()

Для подключения данной библиотеки к тому месту где вы пишите код необходимо воспользоваться командой **import** (данная команда может импортировать не только встроенные или скачанные библиотеки но и ваши функции или классы которые могут быть написаны в отдельных файлах при этом код написанный в других файлах, будет исполнятся и в текущем при вызове данной функции, класса или его метода), в данном случае **import random.**

У данной библиотеки есть ряд встроенных методов и пожалуй самый популярный это метод **.randint().** Он используется для генерации некоего псевдослучайного целого числа в заданном нами диапазоне

```
import random

print(f"Вывод случайного числа: {random.randint(1, 10)}")
```

```
>>>  Вывод случайного числа: 8
```

Есть и 2й метод - не импортировать библиотеку целиком, а взять из нее только нужный нам метод, это как если бы мы взяли только какую-то полку:

```
from random import randint

print(f"Вывод случайного числа: {randint(1, 10)}")
```

```
>>>  Вывод случайного числа: 5
```

Необходимо помнить что в случае диапазона в методе **.randint()** обе границы включаются то есть в нашем примере от 1 по 10.

Библиотека random, метод .shuffle()

Метод **.shuffle()** используется для перемешивания данных списка или другой изменяемой последовательности. Метод **.shuffle()** перемешивает элементы списка на месте. Он работает так как когда вы перемешиваете карты в колоде.

```
import random

some_list = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
print(f"""
Список до перемешивания:
{some_list}""")

random.shuffle(some_list)
print(f"""
Перемешанный список:
{some_list}""")
```

>>>

```
Список до перемешивания:
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

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

Он используется просто для того чтобы перемешать нашу последовательность в некоем псевдослучайном порядке. Мы не перезаписываем список, мы изменяем порядок самого списка а не создаем его новую копию. При каждом запуске программы, цикле порядок элементов списка будет разный, но так как это псевдослучайность мы можем получить и тот же самый.

Библиотека random, метод .sample()

Метод **.sample()** используется когда нам нужно выбрать несколько элементов из заданной последовательности.
Метод **.sample()** возвращает список уникальных элементов (то есть он не может 2 раза взять 1 элемент), которые были выбраны из последовательности **population** (строка, список, словарь и т.д.)**.** Итоговое количество элементов зависит от значения переданного вторым аргументом.
Например если наша задача **перемешать буквы в строке** мы не можем воспользоваться методом **.shuffle**, т.к. **строка это неизменяемая последовательность**, поэтому **1м аргументом мы передаем строку, а вторым количество элементов этой строки (все или определенное количество)**, метод вернет нам список элементов строки. А затем при помощи метода **.join() мы соберем строку.**

```
import random


some_str = "Привет я строка"
items = random.sample(some_str, len(some_str)) >>> ['а', 'к', 'е', ' ', 'с', 'я', 'и', 'р', 'т', 'о', 'р', 'П', 'т', 'в', ' ']
print(items)                                       аке сяиторПтв
result = ''.join(items)
print(result)
```

Библиотека random, метод .choice()

Метод random.choice() используется, когда вам нужно получить один случайный элемент из последовательности. Например, подарок, загаданное слово или участника.

```
import random

some_str = "Привет я строка"
item = random.choice(some_str)

print(item)

some_list = ["яблоки", "груши", "мандарины"]
item = random.choice(some_list)

print(item)
```

```
>>>
а
груши
>>>
и
мандарины
```

Как видно из примера, действительно мы получили случайный элемент как из строки так и из списка

Применение на практике

Нам нужно получить некий случайный подарок из списка применив все 4 метода

```python
# randint, shuffle, sample, choice
import random

gifts = ["барбариски", "мячик", "машинка", "конструктор"]
index = random.randint(0, len(gifts) - 1)
gift = gifts[index]
print(f"Вы получаете: {gift}!")

random.shuffle(gifts)
gift = gifts[0]
print(f"Вы получаете: {gift}!")

gift_list = random.sample(gifts, 1)
print(f"Вы получаете: {gift_list}!")
# т.к. .sample() выводит список может понадобится
# дополнительная обработка данных
gift = random.sample(gifts, 1)[0]
print(f"Вы получаете: {gift}!")

gift = random.choice(gifts)
print(f"Вы получаете: {gift}!")
```

```
>>>
Вы получаете: барбариски!
Вы получаете: мячик!
Вы получаете: ['барбариски']!
Вы получаете: барбариски!
Вы получаете: машинка!
```

Как узнать где вы?

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.

- 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.

Как узнать где вы?

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.

Модуль platform. Некоторые функции из модуля platform

И именно для этого существует модуль, предоставляющий некоторые средства, позволяющие вам узнать, где вы находитесь и какие компоненты работают для вас. Модуль называется платформой. Рассмотрим некоторые функции, которые он вам предоставляет.

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

Вот как ее можно вызвать:

platform(aliased = False, terse = False)

- **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).

Модуль platform функция platform()

И именно для этого 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.

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

Вот как ее можно вызвать:

platform(aliased = False, terse = False)

- **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).

Функции platform(), machine(), processor(), system(), version()

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.

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

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

```
from platform import (platform, machine,
                      processor,
                      system, version)


print(platform())
print(platform(True))
print(platform(False, True))
print(machine())
print(processor())
print(system())
print(version())
```

```
Windows-11-10.0.22631-SP0
Windows-11-10.0.22631-SP0
Windows-11
>>> AMD64
AMD64 Family 25 Model 33 Stepping 2, AuthenticAMD
Windows
10.0.22631
```

```
macOS-14.4.1-arm64-arm-64bit
macOS-14.4.1-arm64-arm-64bit
macOS-14.4.1
>>> arm64
arm
Darwin
Darwin Kernel Version 23.4.0: Fri Mar 15 00:12:41 PDT 2024; root:xnu-10063.1
```

Вот результат запуска на двух разных ПК, как с разными ОС так и с разными архитектурами.

📌 python_implementation() , python_version_tuple()

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.

Вот пример запуска на моем ПК.

```
from platform import python_implementation, python_version_tuple

print(python_implementation())
print(python_version_tuple())
print('.'.join(python_version_tuple()))
```

```
CPython
>>> ('3', '12', '2')
3.12.2
```

В результате выполнения мы получим используемую нами на данный момент версию Python

Каталог модулей Python

Модули 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 (да даже тут! *_*).

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.