

[Home](#) > [Code](#) > [Python](#)

Что такое пространства имен Python (и зачем они нужны?)



Monty Shokeen

Jun 1, 2017 • 8 min read



Русский ▾

Python

Coding Fundamentals

Russian (Русский) translation by [Ilya Nikov](#) (you can also [view the original English article](#))

Конфликты имен все время происходят в реальной жизни. Например, в каждой школе, в которой я когда-либо учился, было по крайней мере два ученика в моем классе с одним и тем же именем. Если кто-то вошел в класс и попросил ученика X, мы с энтузиазмом спросили: «О каком из них вы говорите? Есть два ученика по имени X. «После этого интересующийся человек предоставил бы нам фамилию, и мы указали бы ему верного X.

Всей этой путаницы с именами можно избежать, если у каждого было бы уникальное имя. Это не проблема в классе из 30 студентов. Тем не менее, становится все труднее придумать *уникальное, содержательное и легко запоминаемое* имя для каждого ребенка в школе, городе, стране или во всем мире. Другая проблема в предоставлении каждому ребенку уникального имени заключается в том, что процесс определения того, кто-то еще назвал своего ребенка, Мейси, Маки или Маси, может быть очень утомительным.

Подобный конфликт может также возникнуть в программировании. Когда вы пишете программу из 30 строк без внешних зависимостей, очень легко дать уникальные и значимые имена всем вашим переменным. Проблема возникает, когда в программе есть тысячи строк, и вы также загрузили некоторые внешние модули. В этом уроке вы узнаете о пространствах имен, их важности и разрешении области видимости в Python.

Что такое пространство имен?

Пространство имен - это в основном система, которая гарантирует, что все имена в программе уникальны и могут использоваться без каких-либо конфликтов. Возможно, вы уже знаете, что все в Python строки, списки, функции и т.д. - это объекты. Еще один интересный факт: Python реализует пространства имен как словари. Существует сопоставление «имя-объект» с именами в виде ключей и объектов в качестве значений. Несколько пространств имен могут использовать одно и то же имя и сопоставлять их с другим объектом. Вот несколько примеров пространств имен:

- **Локальное пространство имен:** это пространство имен содержит локальные имена внутри функции. Это пространство имен создается при вызове функции и продолжается до тех пор, пока функция не вернется.
- **Глобальное пространство имен:** это пространство имен, которое включает имена из различных импортированных модулей, которые вы используете в проекте. Оно создается, когда модуль включен в проект, и оно существует до завершения скрипта.
- **Встроенное пространство имен:** это пространство имен содержит встроенные функции и встроенные имена исключений.

В [математических модулях](#) Python на Envato Tuts + я написал о полезных математических функциях, доступных в разных модулях. Например, модули `math` и `cmath` имеют множество функций, которые являются общими для обоих из них, такие как `log10()`, `acos()`, `cos()`, `exp()` и т.д. Если вы используете оба этих модуля в одной и той же программе, единственный способ использовать эти функции с определенностью - это префикс имени модуля, например, `math.log10()` и `cmath.log10()`.

Что такое область?

Пространства имен помогают нам однозначно идентифицировать все имена внутри программы. Однако это не означает, что мы можем использовать имя переменной везде, где захотим. Имя также имеет область действия, определяющую части программы, в которых вы можете использовать это имя без использования префикса. Так же, как пространства имен, в программе также есть несколько областей. Вот список некоторых областей, которые могут существовать во время выполнения программы.

- Локальная область, которая является самой внутренней областью, которая содержит список локальных имен, доступных в текущей функции.
- Область всех закрывающих функций. Поиск имени начинается с ближайшей охватывающей области и перемещается наружу.
- Область уровня модуля, содержащая все глобальные имена из текущего модуля.
- Это область, которая содержит список всех встроенных имен. Поиск в этой области выполняется последним.

В следующих разделах этого руководства мы будем широко использовать встроенную [функцию `dir\(\)` Python](#), чтобы вернуть список имен в текущей локальной области. Это поможет вам более четко понять концепцию пространств имен и области.

Разрешение области

Как я упоминал в предыдущем разделе, поиск данного имени начинается с самой внутренней функции, а затем движется все выше и выше, пока программа не сможет сопоставить это имя с объектом. Если такое имя не найдено ни в одном из пространств имен, программа вызывает исключение *NameError*.

Прежде чем начать, попробуйте ввести `dir()` в IDLE или любой другой Python IDE.

```

1 | dir()
2 | # ['__builtins__', '__doc__', '__loader__', '__name__', '__package__', '_

```

Все эти имена, перечисленные в каталоге `dir()`, доступны в каждой программе Python. Для краткости я начну ссылаться на них как `'__builtins__'` `'...'` `'__spec__'` в остальных примерах.

Давайте посмотрим на результат функции `dir()` после определения переменной и функции.

```

1 | a_num = 10
2 | dir()
3 | # ['__builtins__' .... '__spec__', 'a_num']
4 |
5 | def some_func():
6 |     b_num = 11
7 |     print(dir())
8 |
9 | some_func()
10 | # ['b_num']
11 |
12 | dir()
13 | # ['__builtins__' ... '__spec__', 'a_num', 'some_func']
14 |

```

Функция `dir()` выводит только список имен внутри текущей области. Вот почему внутри области `some_func()` существует только одно имя, называемое `b_num`. Вызов `dir()` после определения `some_func()` добавляет его в список имен, доступных в глобальном пространстве имен.

Теперь давайте посмотрим список имен внутри некоторых вложенных функций. Код в этом блоке продолжается от предыдущего блока.

```

1 | def outer_func():
2 |     c_num = 12
3 |     def inner_func():
4 |         d_num = 13
5 |         print(dir(), ' - names in inner_func')
6 |     e_num = 14
7 |     inner_func()
8 |     print(dir(), ' - names in outer_func')
9 |
10 | outer_func()
11 | # ['d_num'] - names in inner_func
12 | # ['c_num', 'e_num', 'inner_func'] - names in outer_func

```

Вышеприведенный код определяет две переменные и функцию внутри области `external_func()`. Внутри `inner_func()` функция `dir()` только печатает имя `d_num`. Это кажется справедливым, поскольку `d_num` - единственная переменная, определенная там.

Если явно не указано `global`, переназначение глобального имени внутри локального пространства имен создает новую локальную переменную с тем же именем. Это видно из следующего кода.

```
1 | a_num = 10
2 | b_num = 11
3 |
4 | def outer_func():
5 |     global a_num
6 |     a_num = 15
7 |     b_num = 16
8 |     def inner_func():
9 |         global a_num
10 |        a_num = 20
11 |        b_num = 21
12 |        print('a_num inside inner_func :', a_num)
13 |        print('b_num inside inner_func :', b_num)
14 |     inner_func()
15 |     print('a_num inside outer_func :', a_num)
16 |     print('b_num inside outer_func :', b_num)
17 |
18 | outer_func()
19 | print('a_num outside all functions :', a_num)
20 | print('b_num outside all functions :', b_num)
21 |
22 | # a_num inside inner_func : 20
23 | # b_num inside inner_func : 21
24 |
25 | # a_num inside outer_func : 20
26 | # b_num inside outer_func : 16
27 |
28 | # a_num outside all functions : 20
29 | # b_num outside all functions : 11
```

Внутри и `external_func()` и `inner_func()`, `a_num` объявляется глобальной переменной. Мы просто устанавливаем другое значение для одной и той же глобальной переменной. Это связано с тем, что значение `a_num` во всех местоположениях равно 20. С другой стороны, каждая функция создает свою собственную переменную `b_num` с локальной областью, а функция `print()` печатает значение этой переменной с локальным охватом.

Правильный импорт модулей

Очень часто мы импортируем внешние модули в свои проекты для ускорения разработки. Существует три способа импорта модулей. В этом разделе вы узнаете обо всех этих методах, мы подробно обсудим их плюсы и минусы.

- `from module import *`: этот метод импорта модуля импортирует все имена из данного модуля непосредственно в текущее пространство имен. Возможно, у вас возникнет соблазн использовать этот метод, поскольку он позволяет вам напрямую использовать функцию без добавления имени модуля в качестве префикса. Тем не менее, он очень подвержен ошибкам, и вы также теряете возможность сказать, какой модуль фактически импортировал эту функцию. Вот пример использования этого метода:

```
1 | dir()
2 | # ['__builtins__' ... '__spec__']
3 |
4 | from math import *
5 | dir()
6 | # ['__builtins__' ... '__spec__', 'acos', 'acosh', 'asin', 'asinh',
7 | # 'atan', 'atan2', 'atanh', 'ceil', 'copysign', 'cos', 'cosh', 'degrees'
8 | # 'e', 'erf', 'erfc', 'exp', 'expm1', 'fabs', 'factorial', 'floor', 'fmo
9 | # 'frexp', 'fsum', 'gamma', 'gcd', 'hypot', 'inf', 'isclose', 'isfinite'
10 | # 'isinf', 'isnan', 'ldexp', 'lgamma', 'log', 'log10', 'log1p', 'log2',
11 | # 'modf', 'nan', 'pi', 'pow', 'radians', 'sin', 'sinh', 'sqrt', 'tan',
12 | # 'tanh', 'trunc']
13 |
14 | log10(125)
15 | # 2.0969100130080562
16 |
17 | from cmath import *
18 | dir()
19 | # ['__builtins__' ... '__spec__', 'acos', 'acosh', 'asin', 'asinh', 'ata
20 | # 'atan2', 'atanh', 'ceil', 'copysign', 'cos', 'cosh', 'degrees', 'e', '
21 | # 'erfc', 'exp', 'expm1', 'fabs', 'factorial', 'floor', 'fmod', 'frexp',
22 | # 'gamma', 'gcd', 'hypot', 'inf', 'isclose', 'isfinite', 'isinf', 'isnan
23 | # 'ldexp', 'lgamma', 'log', 'log10', 'log1p', 'log2', 'modf', 'nan', 'ph
24 | # 'pi', 'polar', 'pow', 'radians', 'rect', 'sin', 'sinh', 'sqrt', 'tan',
25 | # 'trunc']
26 |
27 | log10(125)
28 | # (2.0969100130080562+0j)
```

Если вы знакомы с модулями *math* и *cmath*, вы уже знаете, что существует несколько общих имен, которые определены в обоих этих модулях, но

применимы к реальным и сложным числам соответственно.

Поскольку мы импортировали модуль *cmath* после модуля *math*, он перезаписывает определения функций этих общих функций из модуля *math*. Вот почему первый `log10(125)` возвращает реальное число, а второй `log10(125)` возвращает комплексное число. Теперь у вас нет возможности использовать функцию `log10()` из математического модуля. Даже если вы попытались ввести `math.log10(125)`, вы получите исключение `NameError`, потому что `math` фактически не существует в пространстве имен.

Суть в том, что вы не должны использовать этот способ импорта функций из разных модулей, чтобы сэкономить несколько нажатий клавиш.

- `from module import nameA, nameB`: Если вы знаете, что используете только одно или два имени из модуля, вы можете импортировать их напрямую с помощью этого метода. Таким образом, вы можете писать код более кратко, сохраняя при этом минимум пространства имен. Однако имейте в виду, что вы по-прежнему не можете использовать какое-либо другое имя из модуля, используя `module.nameZ`. Любая функция с таким же именем в вашей программе также перезапишет определение этой функции, импортированной из модуля. Это сделает непригодную импортированную функцию. Вот пример использования этого метода:

```
1 | dir()
2 | # ['__builtins__' ... '__spec__']
3 |
4 | from math import log2, log10
5 | dir()
6 | # ['__builtins__' ... '__spec__', 'log10', 'log2']
7 |
8 | log10(125)
9 | # 2.0969100130080562
```

- `import module`: это самый безопасный и рекомендуемый способ импорта модуля. Единственным недостатком является то, что вам нужно будет префикс имени модуля ко всем именам, которые вы собираетесь использовать в программе. Тем не менее, вы сможете избежать загрязнения пространства имен, а также определить функции, имена которых соответствуют имени функций модуля.

```
1 | dir()
2 | # ['__builtins__' ... '__spec__']
3 |
4 | import math
5 | dir()
6 | # ['__builtins__' ... '__spec__', 'math']
7 |
8 | math.log10(125)
9 | # 2.0969100130080562
```

Заключение

Надеюсь, этот учебник поможет вам понять пространства имен и их важность. Теперь вы умеете определить область различных имен в программе и избежать потенциальных ошибок.

Кроме того, не стесняйтесь посмотреть, что у нас есть для продажи и [для изучения на рынке](#), и не стесняйтесь задавать любые вопросы и предоставлять свою ценную обратную связь, используя приведенный ниже канал.

В заключительном разделе статьи обсуждались различные способы импорта модулей в Python, их плюсы и минусы. Если у вас есть вопросы по этой теме, пожалуйста, дайте мне знать в комментариях.

Did you find this post useful?



Yes



No

Want a weekly email summary?

Subscribe below and we'll send you a weekly email summary of all new Code tutorials. Never miss out on learning about the next big thing.

Sign up



Monty Shokeen

Freelancer, Instructor

I am a full-stack developer who also loves to write tutorials. After trying out a bunch of things till my second year of college, I decided to work on my web development skills. Starting with just HTML and CSS, I kept moving forward and gained experience in PHP, JavaScript, and Python.

I usually spend my free time either working on some side projects or traveling around.

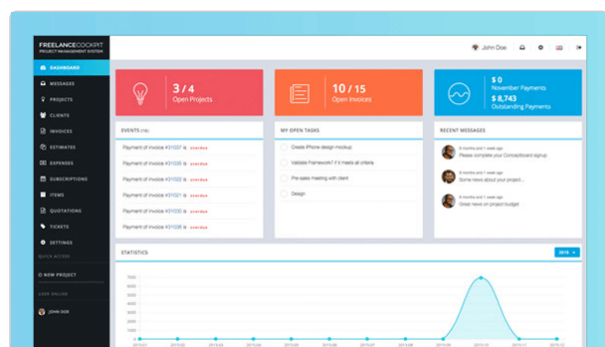
LOOKING FOR SOMETHING TO HELP KICK START YOUR NEXT PROJECT?

Envato Market has a range of items for sale to help get you started.



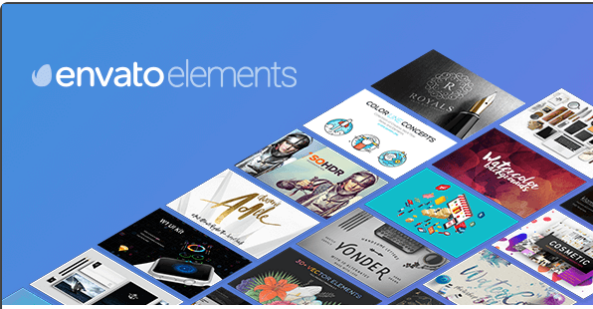
WordPress Plugins

From \$5



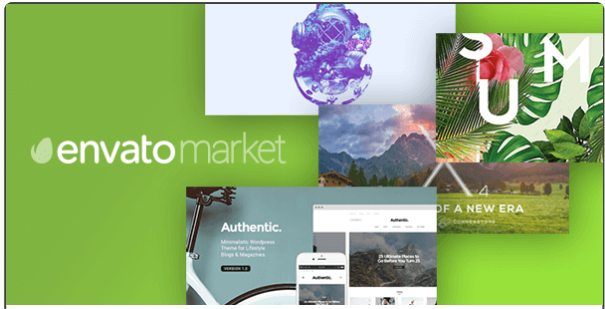
PHP Scripts

From \$5



**Unlimited Downloads
From \$16.50/month**

Get access to over one million creative assets on Envato Elements.



Over 9 Million Digital Assets

Everything you need for your next creative project.

QUICK LINKS - Explore popular categories

ENVATO TUTS+

[About Envato Tuts+](#)
[Terms of Use](#)
[Privacy](#)
[Cookies](#)
[Do not sell or share my
personal information](#)

HELP

[FAQ](#)
[Help Center](#)



tuts+

25,095
Tutorials

553
Courses

19,091
Translations



[Envato](#) [Envato Elements](#) [Envato Market](#) [Placeit by Envato](#) [All products](#) [Careers](#) [Sitemap](#)

© 2024 Envato Pty Ltd. Trademarks and brands are the property of their respective owners.

