👋 Enjoying Exercism? We need your help to survive… Please support us if you can!

Learn Discover Contribute More Insiders

Tracks / Python / Syllabus / Basics

Ba

Basics in

1 exercise

You've mastered Basics in Python. Explore more concepts →

About Basics

Python is a **dynamic and strongly typed** programming language. It employs both **duck typing** and **gradual typing**, via **type hints**. Imperative, declarative (e.g., functional), and object-oriented programming *styles* are all supported, but internally **everything in Python is an object**.

Python puts a strong emphasis on code readability and (*similar to Haskell*) uses **significant indentation** to denote function, method, and class definitions.

Python was created by Guido van Rossum and first released in 1991. The **Python Software Foundation** manages and directs resources for Python and CPython development and receives

proposals for changes to the language from **members** of the community via **Python Enhancement Proposals or PEPs**.

Complete documentation for the current release can be found at **docs.python.org**.

- **Python Tutorial**
- **Python Library Reference**
- **Python Language Reference**
- **Python HOW TOs**
- **Python FAQs**
- **Python Glossary of Terms**

This first concept introduces 4 major Python language features:

1. Name Assignment (*variables and constants*),
2. Functions (*the `def` keyword and the `return` keyword*),
3. Comments, and
4. Docstrings.

> **Note**
>
> In general, content, tests, and analyzer tooling for the Python track follow the style conventions outlined in **PEP 8** and **PEP 257** for Python code style, with the additional

(strong) suggestion that there be no single letter variable names.

The **zen of Python (PEP 20)** and **What is Pythonic?** lay out additional philosophies.

On the Python track, **variables** are always written in `snake_case`, and constants in `SCREAMING_SNAKE_CASE`

Name Assignment (Variables & Constants)

In Python, there are no keywords used in creating variables or constants. Instead, programmers can bind ***names*** (also called *variables*) to any type of object using the assignment `=` operator: `<name> = <value>`. A name can be reassigned (or re-bound) to different values (different object types) over its lifetime.

For example, `my_first_variable` can be re-assigned many times using `=`, and can refer to different object types with each re-assignment:

```
>>> my_first_variable = 1  # my_first_variable bound to an integer object of va
>>> my_first_variable = 2  # my_first_variable re-assigned to integer value 2.

>>> print(type(my_first_variable))
<class 'int'>

>>> print(my_first_variable)
2
```

```
>>> my_first_variable = "Now, I'm a string." # You may re-bind a name to a dif
>>> print(type(my_first_variable))
<class 'str'>

>>> print(my_first_variable)
"Now, I'm a string."  # Strings can be declared using single or double quote ma

import collections
>>> my_first_variable = collections.Counter([1,1,2,3,3,3,4,5,6,7]) # Now my_fir
>>> print(type(my_first_variable))
<class 'collections.Counter'>

>>> print(my_first_variable)
>>> Counter({3: 3, 1: 2, 2: 1, 4: 1, 5: 1, 6: 1, 7: 1})
```

Constants

Constants are names meant to be assigned only once in a program. They should be defined at a **module** (file) level, and are typically visible to all functions and classes in the program. Using `SCREAMING_SNAKE_CASE` signals that the name should not be re-assigned, or its value mutated.

```
# All caps signal that this is intended as a constant.
MY_FIRST_CONSTANT = 16
```

```
# Re-assignment will be allowed by the compiler & interpreter,
# but this is VERY strongly discouraged.
# Please don't do this, it could create problems in your program!
MY_FIRST_CONSTANT = "Some other value"
```

Functions

In Python, units of functionality are encapsulated in ***functions.***, which are themselves **objects** (*it's **turtles all the way down***).

Functions can be executed by themselves, passed as arguments to other functions, nested, or bound to a class. When functions are bound to a **class** name, they're referred to as **methods**. Related functions and classes (*with their methods*) can be grouped together in the same file or module, and imported in part or in whole for use in other programs.

The `def` keyword begins a **function definition**. Each function can have zero or more formal **parameters** in `()` parenthesis, followed by a `:` colon. Statements for the *body* of the function begin on the line following `def` and must be *indented in a block*.

```
# The body of a function is indented by 2 spaces, & prints the sum of the numbe
def add_two_numbers(number_one, number_two):
  total = number_one + number_two
  print(total)

>>> add_two_numbers(3, 4)
7
```

```
# Inconsistent indentation in your code blocks will raise an error.
>>> def add_three_numbers_misformatted(number_one, number_two, number_three):
...      result = number_one + number_two + number_three   # This was indented b
...     print(result)     #this was only indented by 3 spaces
...
...
  File "<stdin>", line 3
    print(result)
    ^
IndentationError: unindent does not match any outer indentation level
```

Functions *explicitly* return a value or object via the `return` keyword. Functions that do not have an *explicit* `return` expression will *implicitly* return `None`.

```
# Function definition on first line.
def add_two_numbers(number_one, number_two):
  result = number_one + number_two
  return result  # Returns the sum of the numbers.

>>> add_two_numbers(3, 4)
7
```

```
# This function will return None.
def add_two_numbers(number_one, number_two):
  result = number_one + number_two

>>> print(add_two_numbers(5, 7))
None
```

Calling Functions

Functions are ***called*** or invoked using their name followed by `()`. Dot (`.`) notation is used for calling functions defined inside a class or module.

```
>>> def number_to_the_power_of(number_one, number_two):
        return number_one ** number_two
...

>>> number_to_the_power_of(3,3) # Invoking the function with the arguments 3 ar
27



# A mis-match between the number of parameters and the number of arguments will
>>> number_to_the_power_of(4,)
...
Traceback (most recent call last):
```

```
  File "<stdin>", line 1, in <module>
TypeError: number_to_the_power_of() missing 1 required positional argument: 'nu


# Calling methods or functions in classes and modules.
>>> start_text = "my silly sentence for examples."
>>> str.upper(start_text)  # Calling the upper() method for the built-in str cl
"MY SILLY SENTENCE FOR EXAMPLES."

# Importing the math module
import math

>>> math.pow(2,4)  # Calling the pow() function from the math module
>>> 16.0
```

Comments

Comments in Python start with a `#` that is not part of a string, and end at line termination. Unlike many other programming languages, Python **does not support** multi-line comment marks. Each line of a comment block must start with the `#` character.

Comments are ignored by the interpreter:

```
# This is a single line comment.
```

```
x = "foo"  # This is an in-line comment.

# This is a multi-line
# comment block over multiple lines --
# these should be used sparingly.
```

Docstrings

The first statement of a function body can optionally be a ***docstring***, which concisely summarizes the function or object's purpose. Docstrings are declared using triple double quotes (""") indented at the same level as the code block:

```
# An example from PEP257 of a multi-line docstring.
def complex(real=0.0, imag=0.0):
    """Form a complex number.

    Keyword arguments:
    real -- the real part (default 0.0)
    imag -- the imaginary part (default 0.0)
    """

    if imag == 0.0 and real == 0.0:
```

```
        return complex_zero
```

Docstrings are read by automated documentation tools and are returned by calling the special attribute `.__doc__` on the function, method, or class name. They are recommended for programs of any size where documentation is needed, and their conventions are laid out in **PEP257**.

Docstrings can also function as **lightweight unit tests**, which can be read and run by PyTest, or by importing the `doctest` module. Testing and `doctest` will be covered in a later concept.

```
# An example on a user-defined function.
>>> def number_to_the_power_of(number_one, number_two):
        """Raise a number to an arbitrary power.

        :param number_one: int the base number.
        :param number_two: int the power to raise the base number to.
        :return: int - number raised to power of second number

        Takes number_one and raises it to the power of number_two, returning th
        """

        return number_one ** number_two
...
```

```
# Calling the .__doc__ attribute of the function and printing the result.
>>> print(number_to_the_power_of.__doc__)
Raise a number to an arbitrary power.

    :param number_one: int the base number.
    :param number_two: int the power to raise the base number to.
    :return: int - number raised to power of second number

    Takes number_one and raises it to the power of number_two, returning the re



# Printing the __doc__ attribute for the built-in type: str.
>>> print(str.__doc__)
str(object='') -> str
str(bytes_or_buffer[, encoding[, errors]]) -> str

Create a new string object from the given object. If encoding or
errors is specified, then the object must expose a data buffer
that will be decoded using the given encoding and error handler.
Otherwise, returns the result of object.__str__() (if defined)
or repr(object).
```

```
encoding defaults to sys.getdefaultencoding().
errors defaults to 'strict'.
```

Learn More

- Reuven Lerner: Understanding Python Assignment
- Sentdex (YouTube): Python 3 Programming Tutorial - Functions
- Real Python: Commenting vs Documenting Code.
- Python Morsels: Everything is an Object
- Eli Bendersky: Python internals: how callables work
- dynamic typing and strong typing
- type hints
- significant indentation
- DigitalOcean: How to Write Doctests in Python.
- Ned Batchelder: Is Python Interpreted or Compiled? Yes.

1 author
2 contributors

Edit via GitHub