

# A Byte of Python

"A Byte of Python" is a free book on programming using the Python language. It serves as a tutorial or guide to the Python language for a beginner audience. If all you know about computers is how to save text files, then this is the book for you.

## For Python version 3

This book will teach you to use Python version 3. There will also be guidance for you to adapt to the older and more common Python version 2 in the book.

For details on differences between Python 2 and Python 3, see:

- [The future of Python 2](#)
- [Porting Python 2 Code to Python 3](#)
- [Writing code that runs under both Python2 and 3](#)
- [Supporting Python 3: An in-depth guide](#)

<https://wiki.python.org/moin/PortingToPy3k/BilingualQuickRef>

# Table of Contents

Introduction	1.1
Dedication	1.2
Preface	1.3
About Python	1.4
Installation	1.5
First Steps	1.6
Basics	1.7
Operators and Expressions	1.8
Control flow	1.9
Functions	1.10
Modules	1.11
Data Structures	1.12
Problem Solving	1.13
Object Oriented Programming	1.14
Input and Output	1.15
Exceptions	1.16
Standard Library	1.17
More	1.18
What Next	1.19
Appendix: FLOSS	1.20
Appendix: About	1.21
Appendix: Revision History	1.22
Appendix: Translations	1.23
Appendix: Translation How-to	1.24
Feedback	1.25

# Functions

## Local Variables

Example (save as `function_local.py`):

```
x = 50

def func(x):
    print('x is', x)
    x = 2
    print('Changed local x to', x)

func(x)
print('x is still', x)
```

Output:

```
$ python function_local.py
x is 50
Changed local x to 2
x is still 50
```

## global Variables

Example (save as `function_global.py`):

```
x = 50

def func():
    global x

    print('x is', x)
    x = 2
    print('Changed global x to', x)

func()
print('Value of x is', x)
```

Output:

```
$ python function_global.py
x is 50
Changed global x to 2
Value of x is 2
```

## Default Argument Values

Example (save as `function_default.py`):

```
def say(message, times=1):  
    print(message * times)  
  
say('Hello')  
say('World', 5)
```

Output:

```
$ python function_default.py  
Hello  
WorldWorldWorldWorldWorld
```

## Keyword Arguments

Example (save as `function_keyword.py`):

```
def func(a, b=5, c=10):  
    print('a is', a, 'and b is', b, 'and c is', c)  
  
func(3, 7)  
func(25, c=24)  
func(c=50, a=100)
```

Output:

```
$ python function_keyword.py  
a is 3 and b is 7 and c is 10  
a is 25 and b is 5 and c is 24  
a is 100 and b is 5 and c is 50
```

# Functions

## VarArgs parameters

Example (save as `function_varargs.py`):

```
def total(a=5, *numbers, **phonebook):
    print('a', a)

    #iterate through all the items in tuple
    for single_item in numbers:
        print('single_item', single_item)

    #iterate through all the items in dictionary
    for first_part, second_part in phonebook.items():
        print(first_part, second_part)

print(total(10, 1, 2, 3, Jack=1123, John=2231, Inge=1560))
```

Output:

```
$ python function_varargs.py
a 10
single_item 1
single_item 2
single_item 3
Inge 1560
John 2231
Jack 1123
None
```

## return statement

Example (save as `function_return.py`):

```
def maximum(x, y):
    if x > y:
        return x
    elif x == y:
        return 'The numbers are equal'
    else:
        return y

print(maximum(2, 3))
```

Output:

```
$ python function_return.py
3
```

`.__doc__`

## DocStrings

Example (save as `function_docstring.py`):

```
def print_max(x, y):
    '''Prints the maximum of two numbers.

    The two values must be integers.'''
    # convert to integers, if possible
    x = int(x)
    y = int(y)

    if x > y:
        print(x, 'is maximum')
    else:
        print(y, 'is maximum')

print_max(3, 5)
print(print_max.__doc__)
```

Output:

```
$ python function_docstring.py
5 is maximum
Prints the maximum of two numbers.

    The two values must be integers.
```

## sys.argv

Example (save as `module_using_sys.py`):

```
import sys

print('The command line arguments are:')
for i in sys.argv:
    print(i)

print('\n\nThe PYTHONPATH is', sys.path, '\n')
```

Output:

```
$ python module_using_sys.py we are arguments
The command line arguments are:
module_using_sys.py
we
are
arguments

The PYTHONPATH is ['/tmp/py',
# many entries here, not shown here
'/Library/Python/2.7/site-packages',
'/usr/local/lib/python2.7/site-packages']
```



## Byte-compiled .pyc files

Importing a module is a relatively costly affair, so Python does some tricks to make it faster. One way is to create *byte-compiled* files with the extension `.pyc` which is an intermediate form that Python transforms the program into (remember the [introduction section](#) on how Python works?). This `.pyc` file is useful when you import the module the next time from a different program - it will be much faster since a portion of the processing required in importing a module is already done. Also, these byte-compiled files are platform-independent.

NOTE: These `.pyc` files are usually created in the same directory as the corresponding `.py` files. If Python does not have permission to write to files in that directory, then the `.pyc` files will *not* be created.

## from..import statement

If you want to directly import the `argv` variable into your program (to avoid typing the `sys.` everytime for it), then you can use the `from sys import argv` statement.

WARNING: In general, *avoid* using the `from..import` statement, use the `import` statement instead. This is because your program will avoid name clashes and will be more readable.

Example:

```
from math import sqrt
print("Square root of 16 is", sqrt(16))
```



Example (save as `module_using_name.py`):

```
if __name__ == '__main__':  
    print('This program is being run by itself')  
else:  
    print('I am being imported from another module')
```

Output:

```
$ python module_using_name.py  
This program is being run by itself  
  
$ python  
>>> import module_using_name  
I am being imported from another module  
>>>
```

Every module has a name and statements in a module can find out the name of their module. This is handy for the particular purpose of figuring out whether the module is being run standalone or being imported. As mentioned previously, when a module is imported for the first time, the code it contains gets executed. We can use this to make the module behave in different ways depending on whether it is being used by itself or being imported from another module. This can be achieved using the `__name__` attribute of the module.

## Making Your Own Modules

Example (save as `mymodule.py`):

```
def say_hi():  
    print('Hi, this is mymodule speaking.')  
__version__ = '0.1'
```

Another module (save as `mymodule_demo.py`):

```
import mymodule  
  
mymodule.say_hi()  
print('Version', mymodule.__version__)
```

Output:

```
$ python mymodule_demo.py  
Hi, this is mymodule speaking.  
Version 0.1
```

Another module (save as `mymodule_demo.py`):

```
from mymodule import say_hi, __version__  
  
say_hi()  
print('Version', __version__)
```

Output:

```
$ python mymodule_demo.py  
Hi, this is mymodule speaking.  
Version 0.1
```

```
$ python
>>> import sys

# get names of attributes in sys module
>>> dir(sys)
['__displayhook__', '__doc__',
'argv', 'builtin_module_names',
'version', 'version_info']
# only few entries shown here

# get names of attributes for current module
>>> dir()
['__builtins__', '__doc__',
'__name__', '__package__']

# create a new variable 'a'
>>> a = 5

>>> dir()
['__builtins__', '__doc__', '__name__', '__package__', 'a']

# delete/remove a name
>>> del a

>>> dir()
['__builtins__', '__doc__', '__name__', '__package__']
```

## Packages

This is how you would structure the folders:

```
- <some folder present in the sys.path>/  
  - world/  
    - __init__.py  
    - asia/  
      - __init__.py  
      - india/  
        - __init__.py  
        - foo.py  
    - africa/  
      - __init__.py  
      - madagascar/  
        - __init__.py  
        - bar.py
```