Teach Python 1: Basics

Python Is…

- An interpreted language with dynamic typing and minimal syntax.
- A multi-paradigm language that supports procedural, object-oriented and functional programming. We are focusing on procedural programming only.
- A well-used language with a large community of support and many libraries available.
- A popular first language.
- A philosophy. (Type "import this" into the Python Shell for "The Zen of Python".)

The Shell

You can evaluate Python expressions and try out code snippets in the Python Shell. It's a sandbox for developing code and it's a great presentation tool when you're teaching.

The Python Shell will become available to you when you run your IDE. If you have not installed a third party IDE (I recommend Pyzo), you can run IDLE (the bundled Python IDE) or just run the python.exe program to get access to the Shell.

Expressions

The following operators work on numeric values (integer and floating point): `+`, `-`, `*`, `/`, `//`, `%`, `**`.

- For most operators, if both operands are integers the result will be an integer, and if one or both are floating point, the result will be floating point.
- The `/` always evaluates to a float, even if both operands are integers.[1]
- Double slash (`//`) is integer division, aka "floor division".[2]
- `**` is the exponentiation operator.
- `%` is the modulus operator.

The + operator is also used for string concatenation. Both operands ***must*** be strings.

> Use either double or single quotes for a string literal: `'a string'`, `"also a string"`
>
> Use three quote characters to create a multiline string: `"""This is an example of a multi-line string"""`

The * operator can work with a string and an integer operand to perform repeated concatenation.

The following operators work with values of any type to return a Boolean: `==`, `!=`, `>`, `<`, `>=`, `<=`.

> The Boolean literals are `True` and `False`.

The logical operators are: `and`, `or`, `not`.

---

[1] This is true for Python 3. In Python 2 if you use / with integer operands, it will perform integer division.

[2] This operator does not exist in Python 2.

Variables

You don't have to declare variables. Just assign to a new name to create a variable.

For example:

```
my_var = 234.5
```

Variables do not have types associated with them. A variable can hold an integer at one point in the program and then a string later.

All values are objects. Variables store references to objects. Even integers, Booleans, etc. are objects, (but they are immutable so they behave as if they were primitive).

Types

All types are object types. Use the built-in function `type` to find out the type (i.e. class) of an object. Basic types include: `int`, `float`, `bool`, `str` and `list`. These type names are also the names of type conversion functions.[3]

> For example: `int("45")` returns 45; `str(45)` returns `"45"`.

Use the built-in Boolean function `isinstance` to test the type of a variable or literal.

> For example: `isinstance(45, int)` returns `True` and `isinstance(45, float)` returns `False`.

The `int` type has no limits (e.g. try `2**1000` in the shell). The `float` type is a standard 64 bit double.

Basic I/O

In what follows, square brackets indicate optional parameters and three dots (…) indicate 0 or more parameters.

`print(..., [sep], [end])`[4]

> Takes 0 or more arguments and prints them to standard output, separated by spaces and followed by a carriage return. You can change the separator using the `sep` keyword argument and the end character by using the `end` keyword argument (more on keyword arguments later).
>
> For example: `print('test', '1', '2', '3',sep='',end='')` prints the string `'test123'` to standard output with no carriage return at the end.

[3] Despite the fact that all values in Python are objects and all types are classes, the language is still designed so that you can program in a purely imperative or procedural way. Many of the "functions" introduced here (e.g. the int function) are actually constructors for a class of the same name.

[4] Print is a function in Python 3, but in Python 2 it is an operator. The syntax is `print "Hello, World"`.

`input([prompt]) -> string`[5]

Returns a string from standard input (i.e. the keyboard) with an optional prompt. If you want a different type (like an `int` or a `float`) you can use the `int` and `float` classes to convert.

For example: `a = int(input('Enter your age: '))`

Compound Statements

Compound statements (like conditionals, loops, function declarations, etc.) consist of a header line, then a colon, then an indented block of statements.

The `if` Statement

```
if spam < 5:
    print(spam, '< 5!')
    spam = spam + 1
```

The While Statement

```
while spam < 5:
    print(spam, '< 5!')
    spam = spam + 1
```

The `if..else` Statement

```
if spam < 5:
    print(spam, '< 5!')
else:
    print(spam, '>= 5!')
```

Chaining with `elif`

```
if spam < eggs:
    print(spam, '<', eggs)
elif spam == eggs:
    print(spam, '=', eggs)
else:
    print(spam, '>', eggs)
```

Note that Python also contains a `break` statement that will terminate a loop early. For example:

```
while True:
    user_input = int(input('Enter an int or 0 to quit: '))
    if user_input == 0:
        break
    print(user_input ** 2)
```

Indentation Rules

The standard indent is 4 spaces, but you're free to indent however you like. However, all the statements in a block must be indented to the same level to avoid a syntax error.

Functions

Function definition uses the `def` keyword. No need to specify any types for parameters or for the return value. Returning a value is optional.

```
def spam(x, y):
    print('You called the spam function!')
    return x < y
```

[5] The equivalent Python 2 function is `raw_input`.

Commenting

Use # to create a single line comment tag.

A triple-quoted string at the top of a module or function is called a "docstring". Docstrings are the Python commenting standard (see https://www.python.org/dev/peps/pep-0257/). They are used to create the output for the `help` function.

Here's an example with comments in bold:

```
""" This is the docstring for my script. It's the first thing in the script
file. """

def my_function(x, y, z):
    """ This is the docstring for my_function. Put some brief
    explanation here. """
    pass

help(my_function)   # this will display the my_function docstring.
```

If you put the above script in a file called **example.py** and then `import example` into another script or into the shell, `help(example)` will display the docstrings.

Variable Scope

Python has function level scope. By default, function parameters and variables only live for the duration of that function call and are only accessible within that function.

Other compound statements do not create a scope. If you introduce a new variable in a `while` statement, that variable will continue to be accessible after the while statement has completed.

If you want to assign to a global variable from within a function, you have to declare it with the global keyword.

Here's an example…

```
x = 5
y = 6
print(x,y)
def test():
      global x
      x = 10[6]
      y = 20
      print(x,y)
test()
print(x,y)
```

and here's its output…

```
5 6
10 20
10 6
```

[6] Try removing this line to see what happens. Can you explain the result?

Modules

A module is a file of python code with a `.py` extension.

If you have a file named `my_module.py` with definitions for a function `my_function` and a global variable `my_var`, you can import those definitions into another module. Here's an example:

```
import my_module
my_module.my_function(my_module.my_var)
my_module.my_var = 10
```

Or you can import individual names directly into your namespace, like this:

```
from my_module import my_function
my_function(10)
```

Or you can import all the names in `my_module` directly into your namespace, like this:

```
from my_module import *
my_function(my_var)
my_var = 10
```

But this is frowned upon (see the last line of "The Zen of Python").

Appendix A: More Built-In Functions

In what follows, square brackets indicate optional parameters and three dots (…) indicate 0 or more parameters. Use `help('builtins')` for a really big list of all built-in functions and classes.

`round(number[, ndigits]) -> number`

Round a number to a given precision in decimal digits (default 0). This returns an `int` when called with one argument, otherwise the same type as the number. `ndigits` may be negative.

`len(object) -> integer`

Return the length of a string or a list.

`max(a, b, ...) -> value`

Return the value of the largest argument.

`min(a, b, ...) -> value`

Return the value of the smallest argument.

`format(value[, format_spec]) -> string`

Returns a formatted string for output. This works a lot like the C `printf` function. For example, `format(45.345, '.2f')` returns `'45.35'`. For more information, visit https://docs.python.org/release/3.2.5/library/string.html#formatspec.

`help(item)`

Use in the shell to get help on any defined name (e.g. `help(print)`). For a big list of predefined names, use `help('builtins')`.

`quit()`

Exits the current script. Also exits the shell on some IDEs.

Appendix B: More Math Functions

To get access to a bunch of other math functions and constants, import the math library:

```
import math
```

This gives you access to trigonometric and other functions, which you can access through the name of the module. For example, to convert a value stored in variable x to radians:

```
x = math.radians(x)
```

Use `help(math)` in the shell (after typing `import math`) to get the full list of functions.

Appendix C: Other Useful Functions

The module `random` contains a function called `randint` to generate random integers.

```
import random
print(random.randint(1,10))
```

The module `time` contains a `sleep` function and a `time` function.

```
import time
print('Back in 5 seconds…')
time.sleep(5)
print('I am back. UNIX time is now:', time.time())
```

Appendix D: Exceptions

Run-time errors are called "exceptions". Exceptions are "raised" and can be "handled" or left to crash the program. Here are some common exceptions you might see:

Exception Type	Example	Fix
TypeError	`x = 'my son is '+3.5`	`x = 'my son is' + str(3.5)`
NameError	`print(x)`	`x = 5` `print(x)`
ValueError	`x = int("34.2")`	`x = int("34")`
ZeroDivisionError	`x = 5 / 0`	`x = 5 / 2`

Exceptions can be handled using a `try.. except` statement, like this:

```
try:
    x = float(input('gimme a non-zero number'))
    y = 1 / x
    print('Inverse of', x, 'is', y)
except ValueError:
    print('That wasn't a number!')
except ZeroDivisionError:
    print('That wasn't non-zero!')
except:
    print('Something went wrong. Not sure what.')
```

Appendix E: More Support Materials

Think Python is a good, brief and free overview text:
http://greenteapress.com/thinkpython2/thinkpython2.pdf

University of Waterloo has some good introductory Python videos:
https://opencs.uwaterloo.ca/

Official Python documentation for the 3.2.5 release is here:
https://docs.python.org/release/3.2.5/

Sure you could do it that way… but *should* you? Python style guide is here:
https://www.python.org/dev/peps/pep-0008/

Python documentation conventions are here:
https://www.python.org/dev/peps/pep-0257/