Python Introduction

Prelude

Welcome to Google's Python online tutorial. It is based on the introductory Python course offered internally. Originally created during the Python 2.4 days, we've tried to keep the content universal and exercises relevant, even for newer releases. As mentioned on the setup page, this material covers Python 2. While we recommend "avoiding" Python 3 for now, recognize that it is the future, as all new features are only going there. The good news is that developers learning either version can pick up the other without too much difficulty. If you want to know more about choosing Python 2 vs. 3, check out this post.

We strongly recommend you follow along with the companion videos throughout the course, starting with the first one. If you're seeking a companion MOOC course, try the ones from Udacity and Coursera (intro to programming [beginners] or intro to Python), and if you're looking for a companion book to your learning, regardless of your Python skill level, check out these reading lists. Finally, if you're seeking self-paced online learning *without* watching videos, try the ones listed towards the end of this post — each feature learning content as well as a Python interactive interpreter you can practice with. What's this "interpreter" we mention? You'll find out in the next section!

Language Introduction

Python is a dynamic, interpreted (bytecode-compiled) language. There are no type declarations of variables, parameters, functions, or methods in source code. This makes the code short and flexible, and you lose the compile-time type checking of the source code. Python tracks the types of all values at runtime and flags code that does not make sense as it runs.

An excellent way to see how Python code works is to run the Python interpreter and type code right into it. If you ever have a question like, "What happens if I add an `int` to a `list`?" Just typing it into the Python interpreter is a fast and likely the best way to see what happens. (See below to see what really happens!)

```
$ python        ## Run the Python interpreter
```

```
Python 2.7.9 (default, Dec 30 2014, 03:41:42)
[GCC 4.1.2 20080704 (Red Hat 4.1.2-55)] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>> a = 6       ## set a variable in this interpreter session
>>> a           ## entering an expression prints its value
6
>>> a + 2
8
>>> a = 'hi'    ## 'a' can hold a string just as well
>>> a
'hi'
>>> len(a)      ## call the len() function on a string
2
>>> a + len(a)  ## try something that doesn't work
Traceback (most recent call last):
  File "", line 1, in
TypeError: cannot concatenate 'str' and 'int' objects
>>> a + str(len(a))  ## probably what you really wanted
'hi2'
>>> foo         ## try something else that doesn't work
Traceback (most recent call last):
  File "", line 1, in
NameError: name 'foo' is not defined
>>> ^D          ## type CTRL-d to exit (CTRL-z in Windows/DOS terminal)
```

As you can see above, it's easy to experiment with variables and operators. Also, the interpreter throws, or "raises" in Python parlance, a runtime error if the code tries to read a variable that has not been assigned a value. Like C++ and Java, Python is case sensitive so "a" and "A" are different variables. The end of a line marks the end of a statement, so unlike C++ and Java, Python does not require a semicolon at the end of each statement. Comments begin with a '#' and extend to the end of the line.

Python source code

Python source files use the ".py" extension and are called "modules." With a Python module `hello.py`, the easiest way to run it is with the shell command "python hello.py Alice" which calls the Python interpreter to execute the code in `hello.py`, passing it the command line argument "Alice". See the official docs page on all the different options you have when running Python from the command-line.

Here's a very simple `hello.py` program (notice that blocks of code are delimited strictly using

indentation rather than curly braces — more on this later!):

```
#!/usr/bin/env python

# import modules used here -- sys is a very standard one
import sys

# Gather our code in a main() function
def main():
    print 'Hello there', sys.argv[1]
    # Command line args are in sys.argv[1], sys.argv[2] ...
    # sys.argv[0] is the script name itself and can be ignored

# Standard boilerplate to call the main() function to begin
# the program.
if __name__ == '__main__':
    main()
```

Running this program from the command line looks like:

```
$ python hello.py Guido
Hello there Guido
$ ./hello.py Alice  ## without needing 'python' first (Unix)
Hello there Alice
```

Imports, Command-line arguments, and `len()`

The outermost statements in a Python file, or "module", do its one-time setup — those statements run from top to bottom the first time the module is imported somewhere, setting up its variables and functions. A Python module can be run directly — as above "python hello.py Bob" — or it can be imported and used by some other module. When a Python file is run directly, the special variable "__name__" is set to "__main__". Therefore, it's common to have the boilerplate `if __name__ ==...` shown above to call a main() function when the module is run directly, but not when the module is imported by some other module.

In a standard Python program, the list `sys.argv` contains the command-line arguments in the standard way with sys.argv[0] being the program itself, sys.argv[1] the first argument, and so on. If you know about `argc`, or the number of arguments, you can simply request this value from Python with `len(sys.argv)`, just like we did in the interactive interpreter code above when requesting the length of a string. In general, `len()` can tell you how long a string is, the number of elements

in lists and tuples (another array-like data structure), and the number of key-value pairs in a dictionary.

User-defined Functions

Functions in Python are defined like this:

```
# Defines a "repeat" function that takes 2 arguments.
def repeat(s, exclaim):
    """
    Returns the string 's' repeated 3 times.
    If exclaim is true, add exclamation marks.
    """

    result = s + s + s # can also use "s * 3" which is faster (Why?)
    if exclaim:
        result = result + '!!!'
    return result
```

Notice also how the lines that make up the function or if-statement are grouped by all having the same level of indentation. We also presented 2 different ways to repeat strings, using the + operator which is more user-friendly, but * also works because it's Python's "repeat" operator, meaning that `'-' * 10` gives `'----------'`, a neat way to create an onscreen "line." In the code comment, we hinted that * works faster than +, the reason being that * calculates the size of the resulting object once whereas with +, that calculation is made each time + is called. Both + and * are called "overloaded" operators because they mean different things for numbers vs. for strings (and other data types).

The `def` keyword defines the function with its parameters within parentheses and its code indented. The first line of a function can be a documentation string ("docstring") that describes what the function does. The docstring can be a single line, or a multi-line description as in the example above. (Yes, those are "triple quotes," a feature unique to Python!) Variables defined in the function are local to that function, so the "result" in the above function is separate from a "result" variable in another function. The `return` statement can take an argument, in which case that is the value returned to the caller.

Here is code that calls the above repeat() function, printing what it returns:

```
def main():
    print repeat('Yay', False)      ## YayYayYay
```

```
    print repeat('Woo Hoo', True)   ## Woo HooWoo HooWoo Hoo!!!
```

At run time, functions must be defined by the execution of a "def" before they are called. It's typical to def a main() function towards the bottom of the file with the functions it calls above it.

Indentation

One unusual Python feature is that the whitespace indentation of a piece of code affects its meaning. A logical block of statements such as the ones that make up a function should all have the same indentation, set in from the indentation of their parent function or "if" or whatever. If one of the lines in a group has a different indentation, it is flagged as a syntax error.

Python's use of whitespace feels a little strange at first, but it's logical and I found I got used to it very quickly. Avoid using TABs as they greatly complicate the indentation scheme (not to mention TABs may mean different things on different platforms). Set your editor to insert spaces instead of TABs for Python code.

A common question beginners ask is, "How many spaces should I indent?" According to the official Python style guide (PEP 8), you should indent with 4 spaces. (Fun fact: Google's internal style guideline dictates indenting by 2 spaces!)

Code Checked at Runtime

Python does very little checking at compile time, deferring almost all type, name, etc. checks on each line until that line runs. Suppose the above main() calls repeat() like this:

```
def main():
    if name == 'Guido':
        print repeeeet(name) + '!!!'
    else:
        print repeat(name)
```

The if-statement contains an obvious error, where the repeat() function is accidentally typed in as repeeeet(). The funny thing in Python ... this code compiles and runs fine so long as the name at runtime is not 'Guido'. Only when a run actually tries to execute the repeeeet() will it notice that there is no such function and raise an error. This just means that when you first run a Python program, some of the first errors you see will be simple typos like this. This is one area where languages with a more verbose type system, like Java, have an advantage ... they can catch such

errors at compile time (but of course you have to maintain all that type information ... it's a tradeoff).

Variable Names

Since Python variables don't have any type spelled out in the source code, it's extra helpful to give meaningful names to your variables to remind yourself of what's going on. So use "name" if it's a single name, and "names" if it's a list of names, and "tuples" if it's a list of tuples. Many basic Python errors result from forgetting what type of value is in each variable, so use your variable names (all you have really) to help keep things straight.

As far as actual naming goes, some languages prefer underscored_parts for variable names made up of "more than one word," but other languages prefer camelCasing. In general, Python prefers the underscore method but guides developers to defer to camelCasing if integrating into existing Python code that already uses that style. Readability counts. Read more in the section on naming conventions in PEP 8.

As you can guess, keywords like 'print' and 'while' cannot be used as variable names — you'll get a syntax error if you do. However, be careful not to use built-ins as variable names. For example, while 'str' and 'list' may seem like good names, you'd be overriding those system variables. Built-ins are not keywords and thus, are susceptible to inadvertent use by new Python developers.

More on Modules and their Namespaces

Suppose you've got a module "binky.py" which contains a "def foo()". The fully qualified name of that foo function is "binky.foo". In this way, various Python modules can name their functions and variables whatever they want, and the variable names won't conflict — module1.foo is different from module2.foo. In the Python vocabulary, we'd say that binky, module1, and module2 each have their own "namespaces," which as you can guess are variable name-to-object bindings.

For example, we have the standard "sys" module that contains some standard system facilities, like the argv list, and exit() function. With the statement "import sys" you can can then access the definitions in the sys module and makes them available by their fully-qualified name, e.g. sys.exit(). (Yes, 'sys' has a namespace too!)

```
import sys
```

```
# Now can refer to sys.xxx facilities
sys.exit(0)
```

There is another import form that looks like this: "from sys import argv, exit". That makes argv and exit() available by their short names; however, we recommend the original form with the fully-qualified names because it's a lot easier to determine where a function or attribute came from.

There are many modules and packages which are bundled with a standard installation of the Python interpreter, so you don't have do anything extra to use them. These are collectively known as the "Python Standard Library." Commonly used modules/packages include:

- sys — access to exit(), argv, stdin, stdout, ...
- re — regular expressions
- os — operating system interface, file system

You can find the documentation of all the Standard Library modules and packages at http://docs.python.org/library.

Online help, `help()`, and `dir()`

There are a variety ways to get help for Python.

- Do a Google search, starting with the word "python", like "python list" or "python string lowercase". The first hit is often the answer. This technique seems to work better for Python than it does for other languages for some reason.
- The official Python docs site — docs.python.org — has high quality docs. Nonetheless, I often find a Google search of a couple words to be quicker.
- There is also an official Tutor mailing list specifically designed for those who are new to Python and/or programming!
- Many questions (and answers) can be found on StackOverflow and Quora.
- Use the help() and dir() functions (see below).

Inside the Python interpreter, the help() function pulls up documentation strings for various modules, functions, and methods. These doc strings are similar to Java's javadoc. The dir() function tells you what the attributes of an object are. Below are some ways to call help() and dir() from the interpreter:

- `help(len)` — help string for the built-in `len()` function; note that it's "len" not "len()", which is a **call** to the function, which we don't want
- `help(sys)` — help string for the `sys` module (must do an `import sys` first)
- `dir(sys)` — `dir()` is like `help()` but just gives a quick list of its defined symbols, or "attributes"
- `help(sys.exit)` — help string for the `exit()` function in the `sys` module
- `help('xyz'.split)` — help string for the `split()` method for string objects. You can call `help()` with that object itself or an **example** of that object, plus its attribute. For example, calling `help('xyz'.split)` is the same as calling `help(str.split)`.
- `help(list)` — help string for `list` objects
- `dir(list)` — displays `list` object attributes, including its methods
- `help(list.append)` — help string for the `append()` method for `list` objects

Last updated February 4, 2015.