Intro to Robotics with Raspberry Pi!

Section 1. General Concepts – Programming

Outline

Intro to Programming in Python

Motivation

Why learn to code?

Why Python?

Who uses Python?

Getting Started

Raspberry Pi, Linux, Python interpreter, and

Integrated Development Environment (IDLE)

Using Python Interactively

Python as a calculator

Basic data types, assignment, and variables

Interactive programs: input and output

Control of flow (choice and loops)

Compound data types (lists and dictionaries)

Functions

Built-in modules

Writing and Running Python Scripts

Working with Python scripts

Custom modules

Reading and writing files

Applications

Plotting data with *matplotlib*

Running a simple web application with Flask

Motivation

Why learn to code?

You've probably heard many reasons to learn programming: make money, create apps/websites, build critical-thinking skills, put it on your resume…

But *really*, why should you?

- Programming is a **tool** that allows you to tackle and (hopefully) solve many kinds of problems.
- We're surrounded by **data** (*digital age*), programming allows to interact with data in meaningful and efficient ways (e.g., fantasy football).
- Allows for **automating** repetitive tasks (e.g., search and replace multiple docs, renaming photos).
- **Teaches** how to learn (master Google searches) and practice precise, disciplined, and abstract thinking.

Why Python?

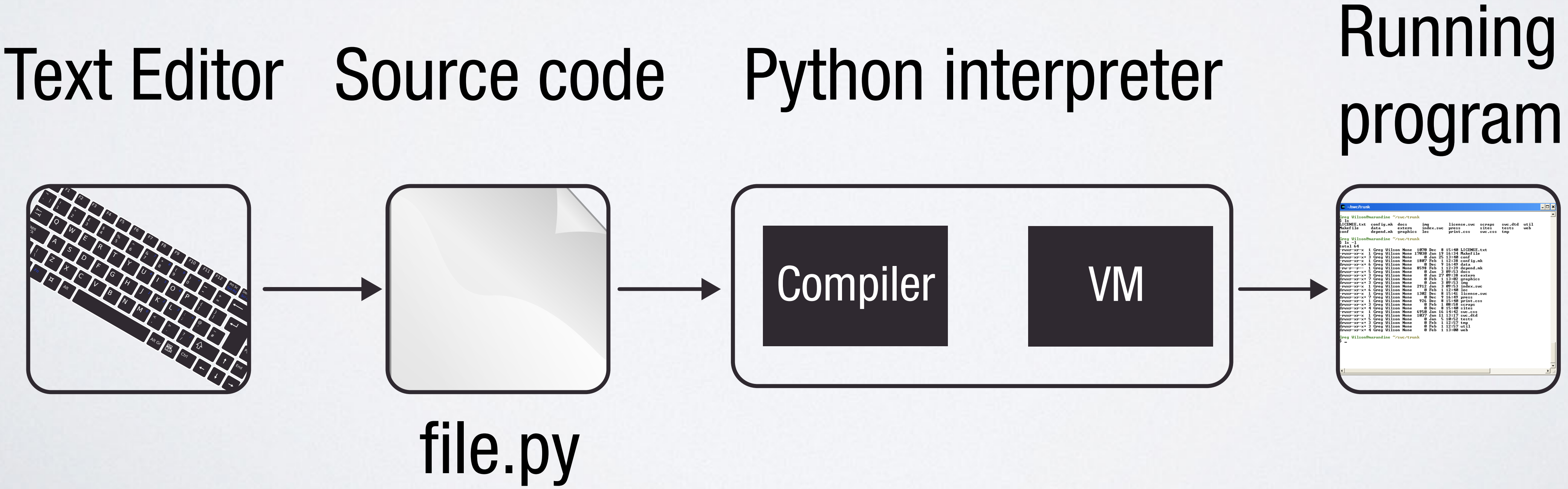
Python is a *general purpose* programming language created by Guido Van Rossum.

- One of the **easiest** languages to learn/teach (friendly syntax).
- Wealth of **free tools** to start developing in Python.
- Large community **support** (Q&A, extensive collection of libraries).
- Concepts applicable to **other programming languages** (C#, Perl, …)
- Ranked **top eight** most popular programming languages in the world.

The Python Programming Language

Language Features

- Python is an **interpreted** language (no need to compile).
- Python is **dynamically typed** (no need to declare data types).
- Statement grouping is done by **indentation** instead of beginning and ending brackets (readability!)
- Paired with a full-featured scripting **interpreter**.



Text Editor

Source code

Python interpreter

Running program

Compiler

VM

file.py

The Python Programming Language

Python usage in the ‘real’ world

- Web/desktop applications, games, analyzing and visualizing data…

Instagram

Task Queue & Push Notifications

Battlefield 2

Score Keeping & Team Balancing

NASA – LDSD

Aerodynamic Modeling & Simulation

Blender

UI, Add-ons, Import/ Export Tools

Getting Started with Python

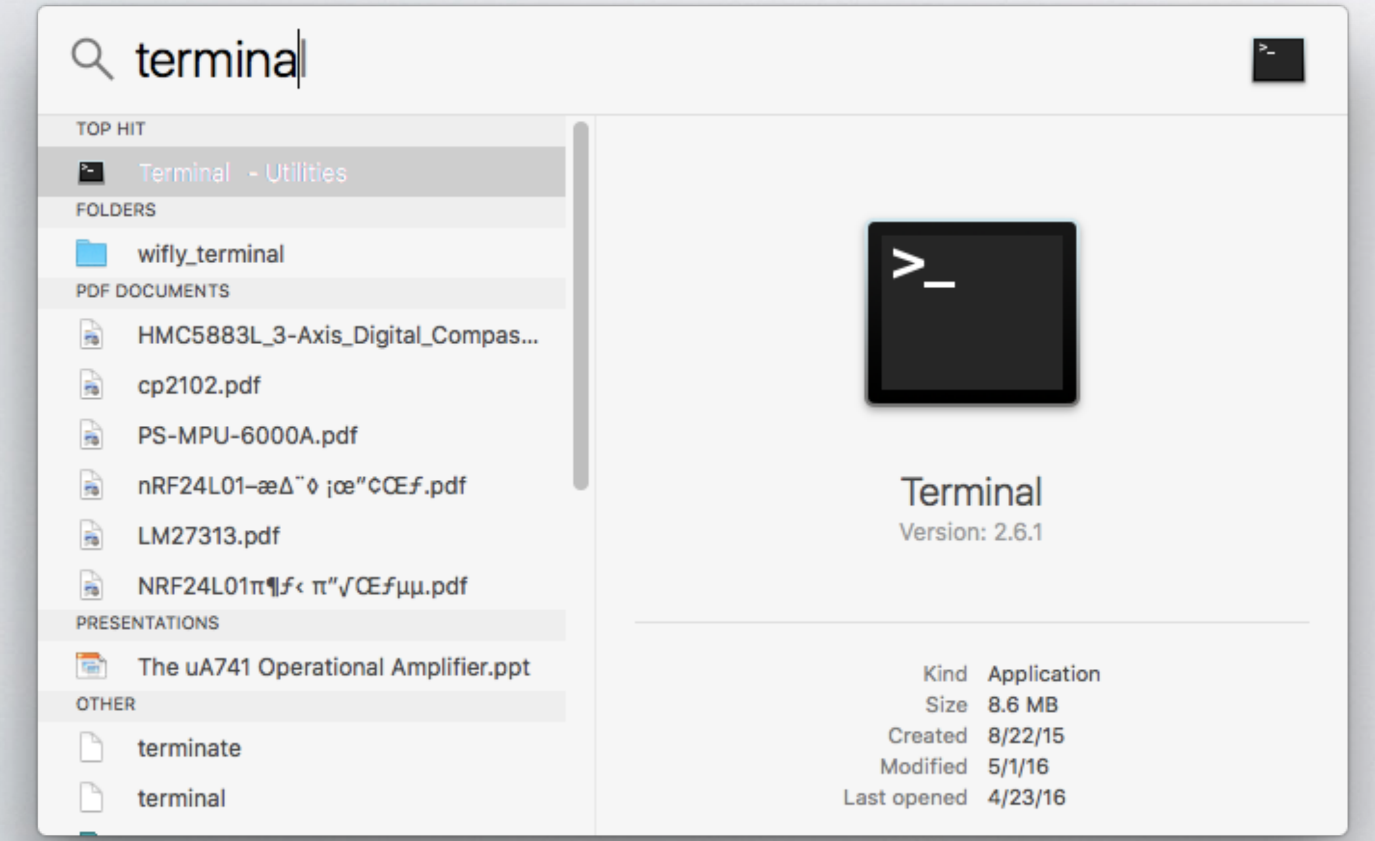
“A little less conversation, a little more action please…”

The Python Interpreter

- The **interpreter** is the program that allows you to run ‘unpacked’ Python code on your computer.
- It can be run in **interactive** (calculator) mode by issuing the command:

```
python
```

- Open the **Terminal.app** and try it!



terminal

TOP HIT

Terminal - Utilities

FOLDERS

wifly_terminal

PDF DOCUMENTS

HMC5883L_3-Axis_Digital_Compas...

cp2102.pdf

PS-MPU-6000A.pdf

nRF24L01–æΔ¨◊ ¡œ”¢Œƒ.pdf

LM27313.pdf

NRF24L01π¶ƒ‹ π”√Œƒµµ.pdf

PRESENTATIONS

The uA741 Operational Amplifier.ppt

OTHER

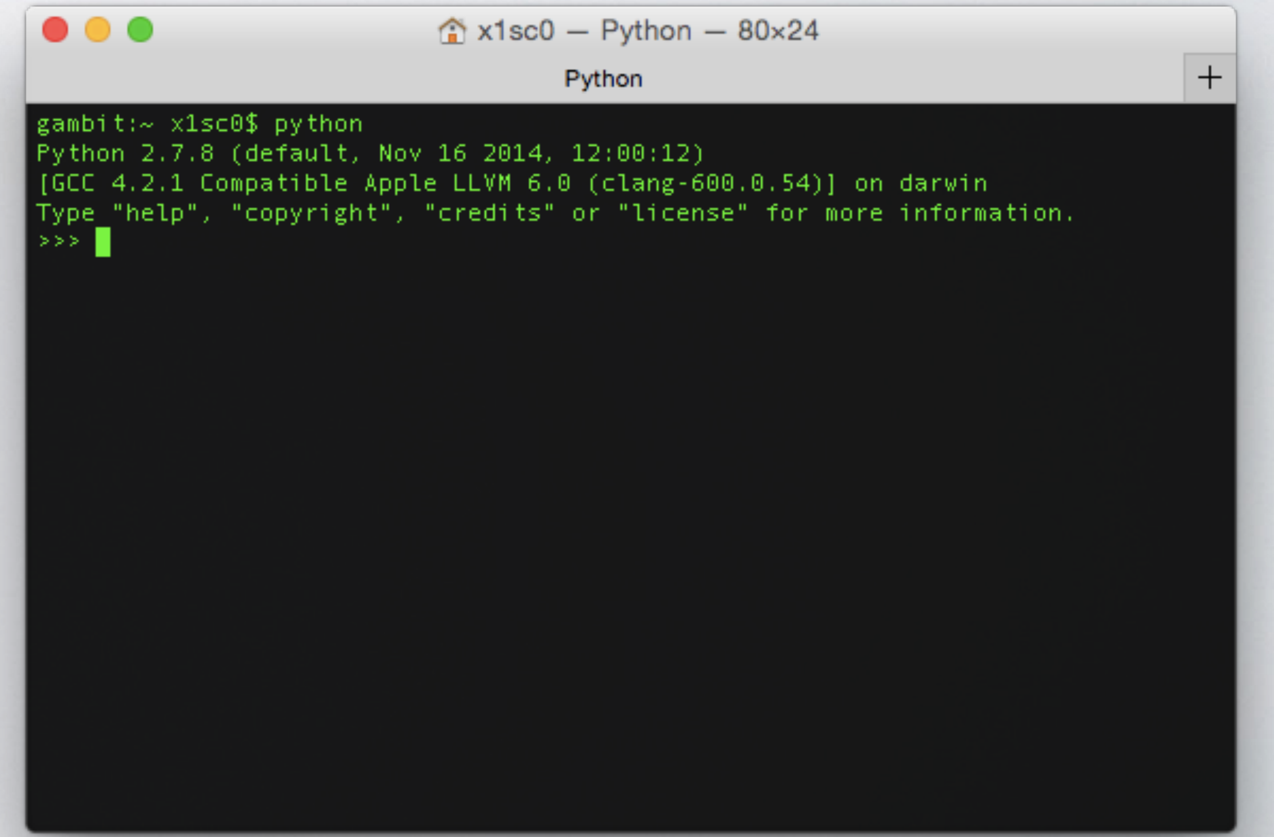
terminate

terminal

Terminal

Version: 2.6.1

Kind	Application
Size	8.6 MB
Created	8/22/15
Modified	5/1/16
Last opened	4/23/16



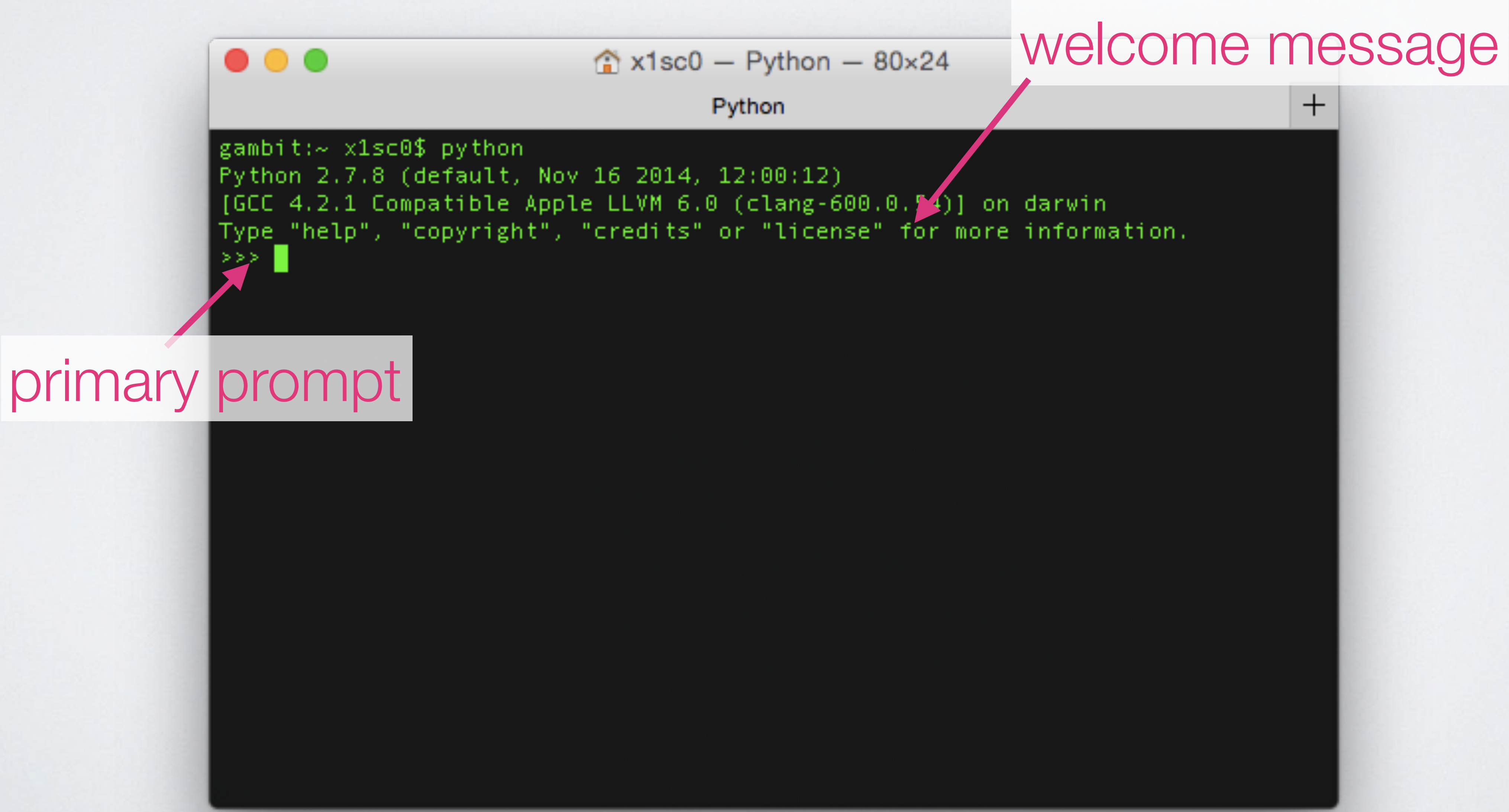
x1sc0 — Python — 80×24

Python

```
gambit:~ x1sc0$ python
Python 2.7.8 (default, Nov 16 2014, 12:00:12)
[GCC 4.2.1 Compatible Apple LLVM 6.0 (clang-600.0.54)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> 
```

The Python Interpreter

- In interactive mode the interpreter prompts for the next command with the primary prompt (**>>>**).
- For continuation lines the interpreter prompts w/ secondary prompt (**...**).
- The interpreter prints a welcome message (version number and copyright)



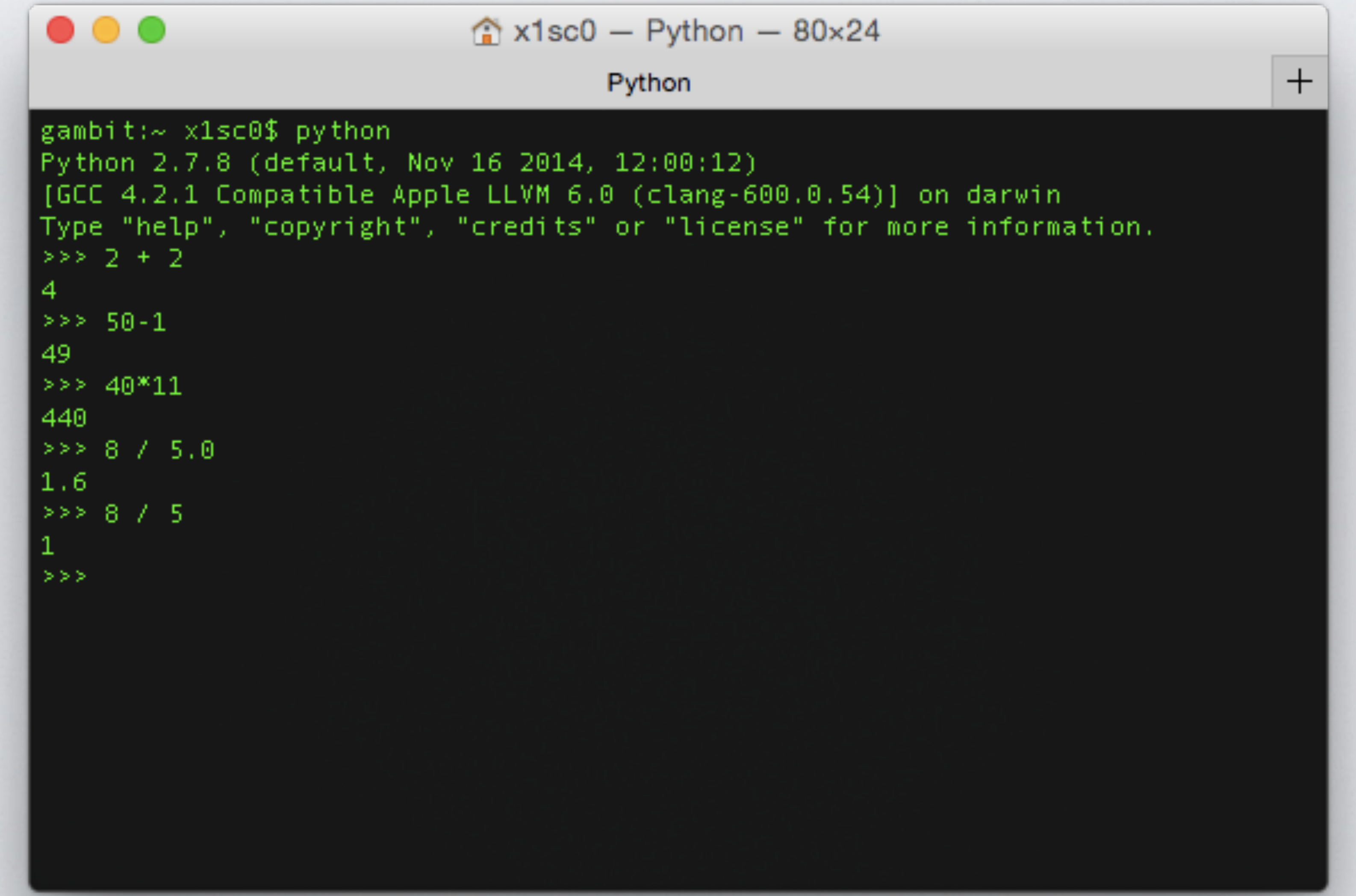
```
gambit:~ x1sc0$ python
Python 2.7.8 (default, Nov 16 2014, 12:00:12)
[GCC 4.2.1 Compatible Apple LLVM 6.0 (clang-600.0.54)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> 
```

welcome message

primary prompt

Numbers

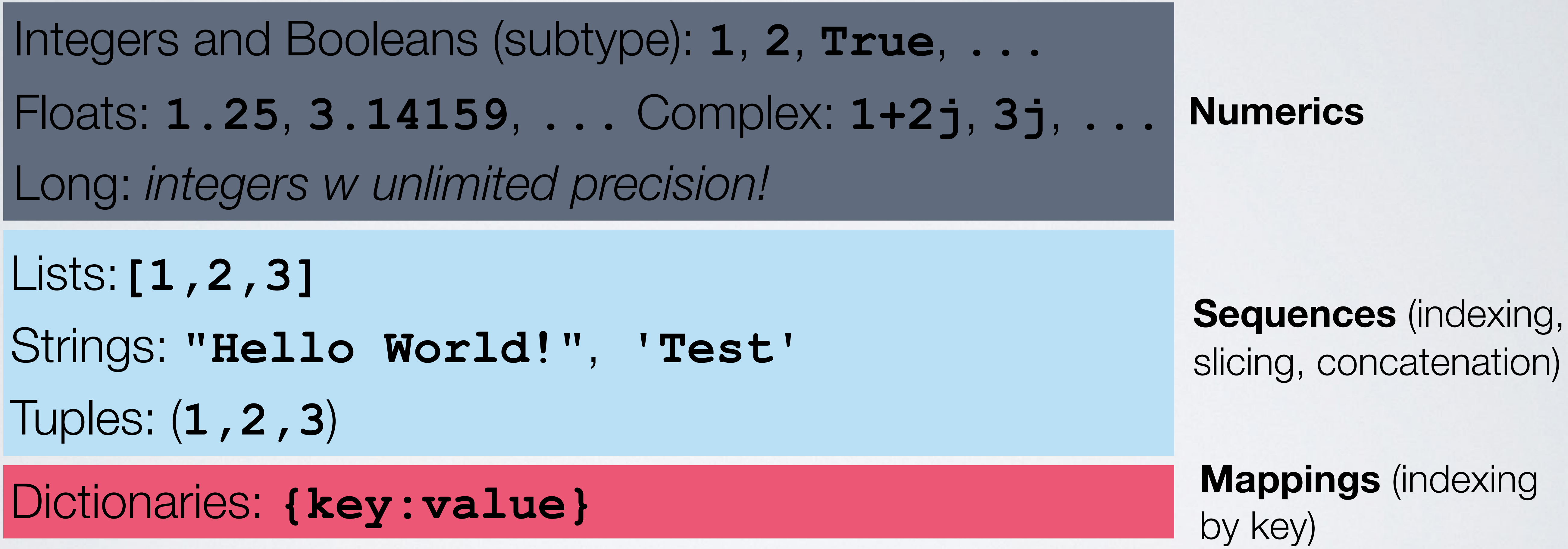
- In interactive mode Python behaves as a calculator.
- It ignores whitespace except for indentation.
- We need to be careful with operations between different **data types** (e.g., adding a string and a number).



```
gambit:~ x1sc0$ python
Python 2.7.8 (default, Nov 16 2014, 12:00:12)
[GCC 4.2.1 Compatible Apple LLVM 6.0 (clang-600.0.54)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> 2 + 2
4
>>> 50-1
49
>>> 40*11
440
>>> 8 / 5.0
1.6
>>> 8 / 5
1
>>>
```

Built-In Data Types

- Disclaimer: everything in Python is an object, and almost everything has attributes and methods.
- The principal built-in types are:



Integers and Booleans (subtype): `1`, `2`, `True`, `...`

Floats: `1.25`, `3.14159`, `...` Complex: `1+2j`, `3j`, `...`

Long: *integers w unlimited precision!*

Numerics

Lists: `[1,2,3]`

Strings: `"Hello World!"`, `'Test'`

Tuples: (`1,2,3`)

Sequences (indexing, slicing, concatenation)

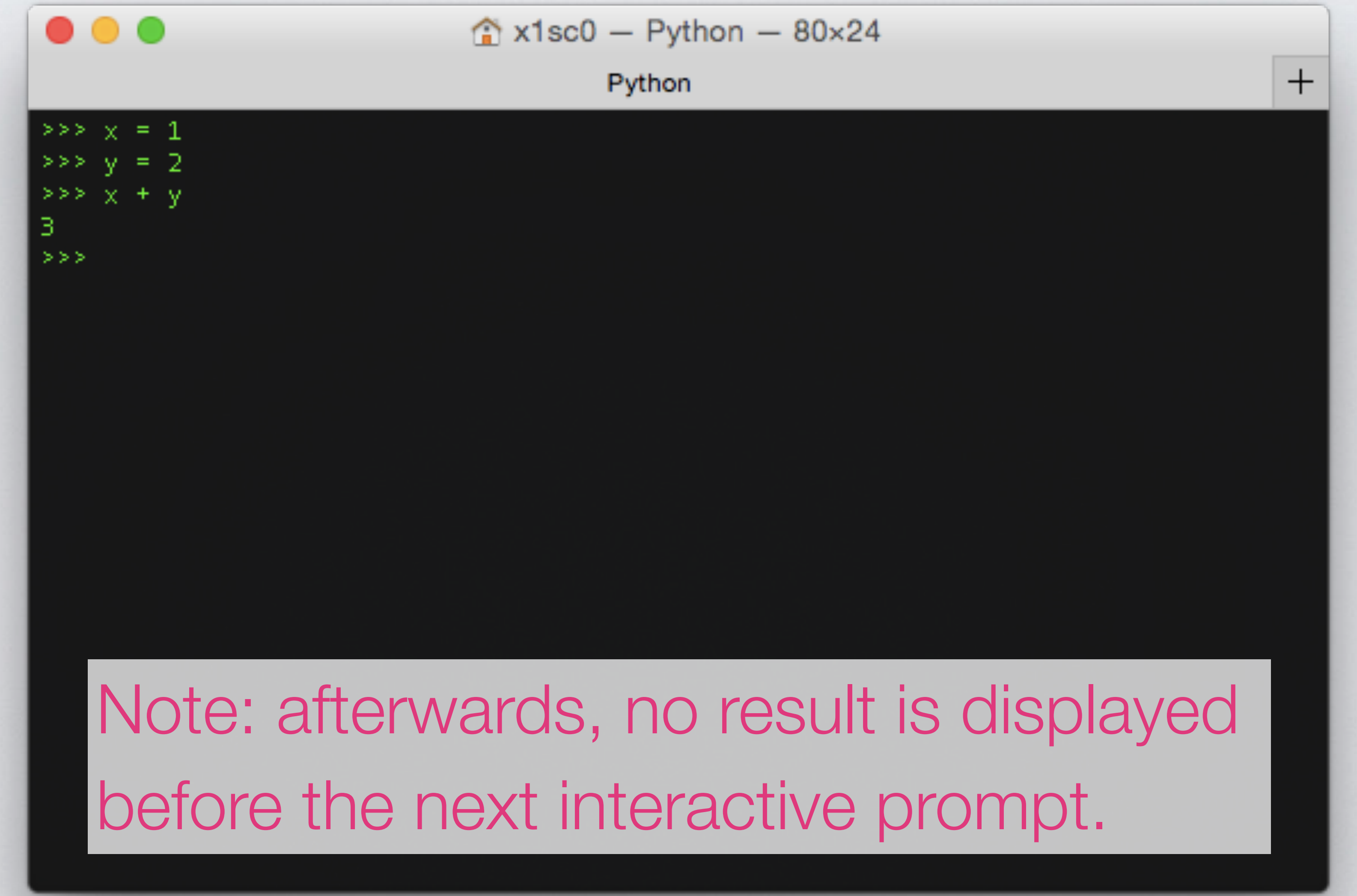
Dictionaries: `{key:value}`

Mappings (indexing by key)

- Other built-in types include **sets** and **files**.
- Additional types exist for representing things like **dates**, and can be imported from modules (libraries).

Assignment and Variables

- Variables can store values and we tend to use them similarly than in *math*!
- The equal sign (**=**) is used to assign a value to a variable.
- Variables are quite powerful, they can **store** any data type!



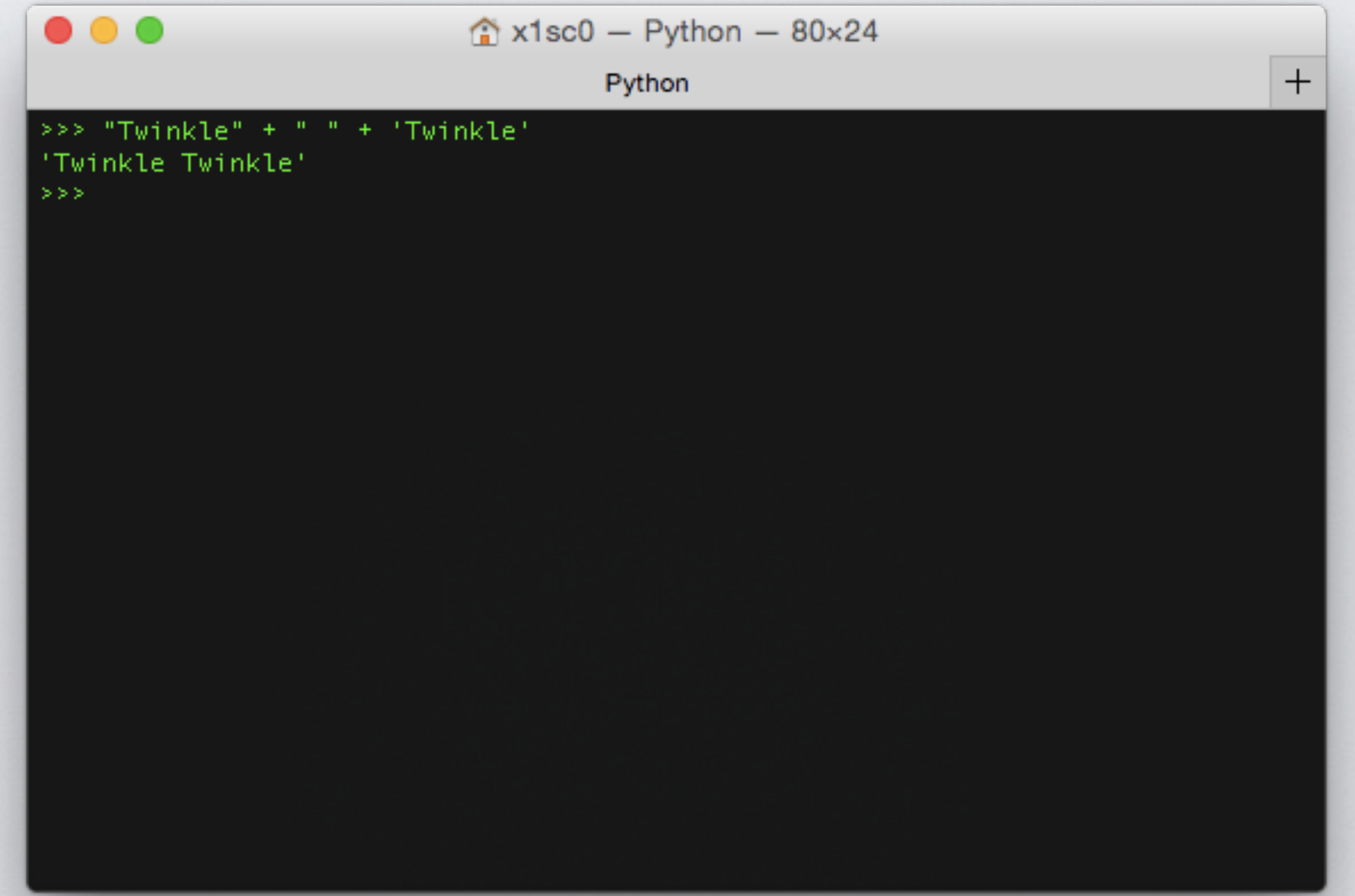
```
>>> x = 1
>>> y = 2
>>> x + y
3
>>>
```

Note: afterwards, no result is displayed before the next interactive prompt.

Working with Strings

- In computer programming, a string is traditionally a **sequence** of characters.

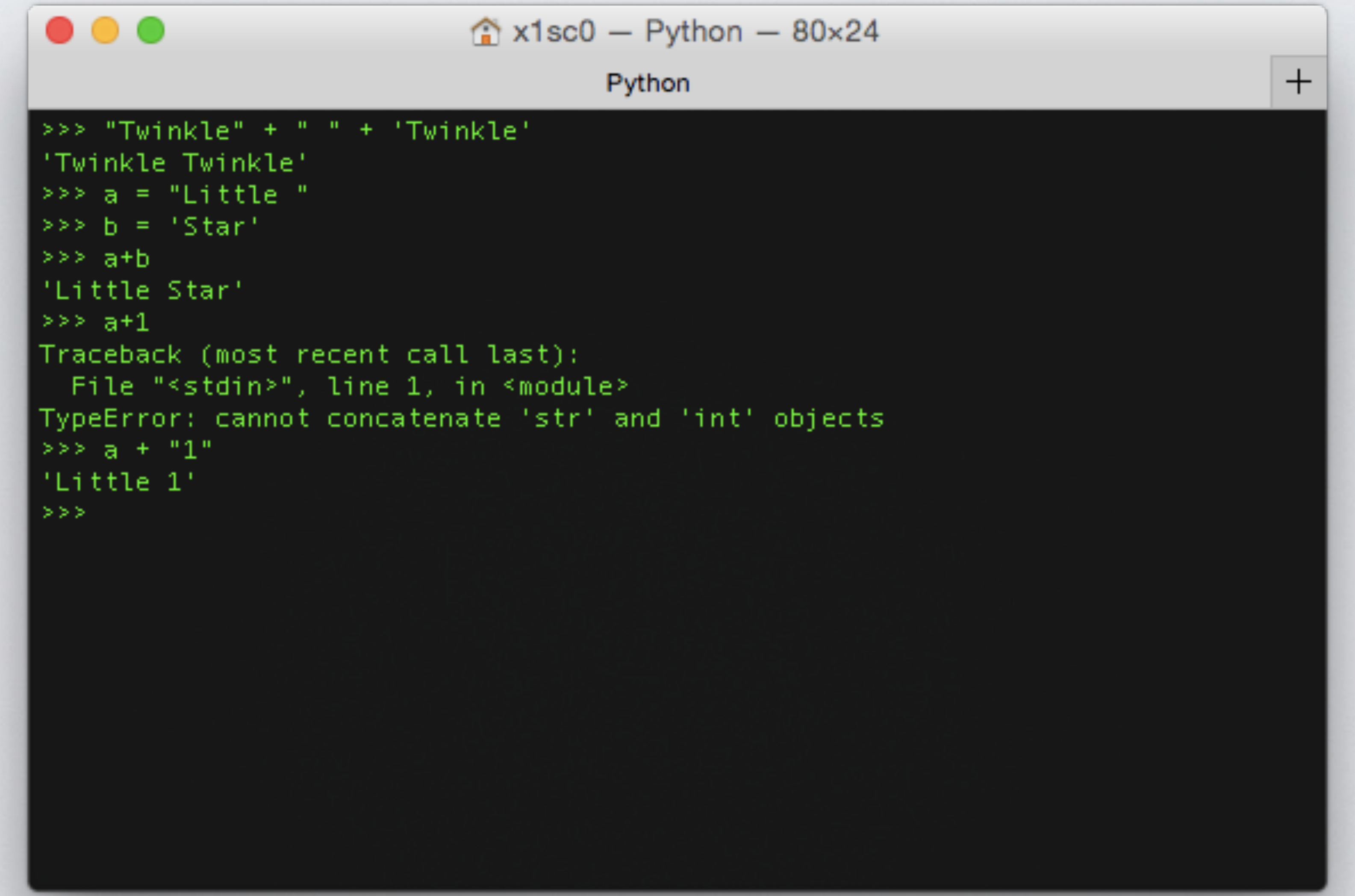
- In Python, strings are enclosed in either single (`'...'`) or double quotes (`"..."`) with the same result.



```
>>> "Twinkle" + " " + 'Twinkle'
'Twinkle Twinkle'
>>>
```

Working with Strings

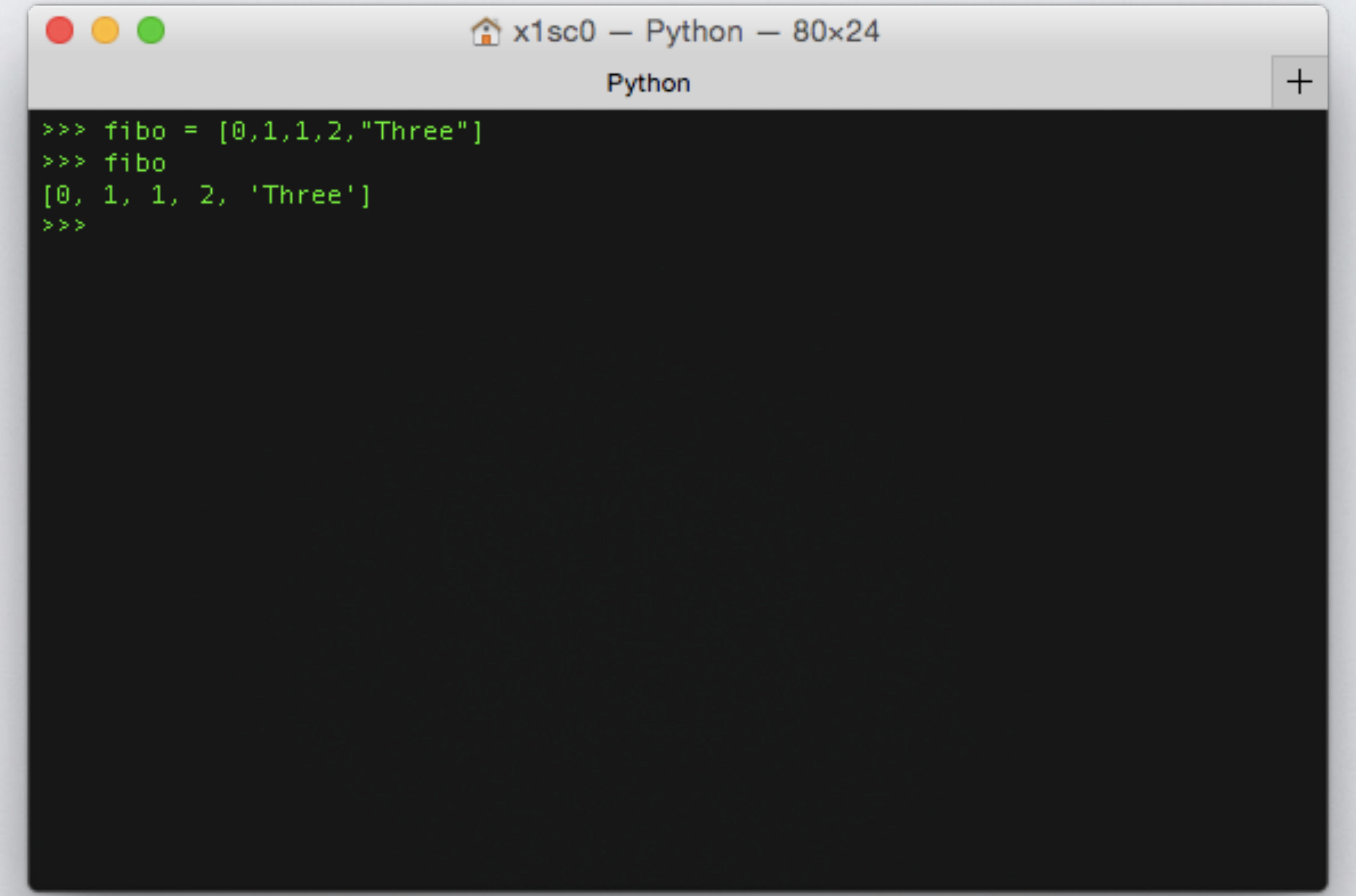
- The **+** operator concatenates strings
- The ***** operator repeats strings
- Numbers and strings cannot be concatenated (different data types)



```
>>> "Twinkle" + " " + 'Twinkle'
'Twinkle Twinkle'
>>> a = "Little "
>>> b = 'Star'
>>> a+b
'Little Star'
>>> a+1
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: cannot concatenate 'str' and 'int' objects
>>> a + "1"
'Little 1'
>>>
```

Working with Lists

- Lists are a **sequences** (compound data type) that group together other values.
- Lists can be written as **comma-separated** values in square brackets.
- Lists may contain items of different types (usually they're of the same type).



```
>>> fibo = [0,1,1,2,"Three"]
>>> fibo
[0, 1, 1, 2, 'Three']
>>>
```

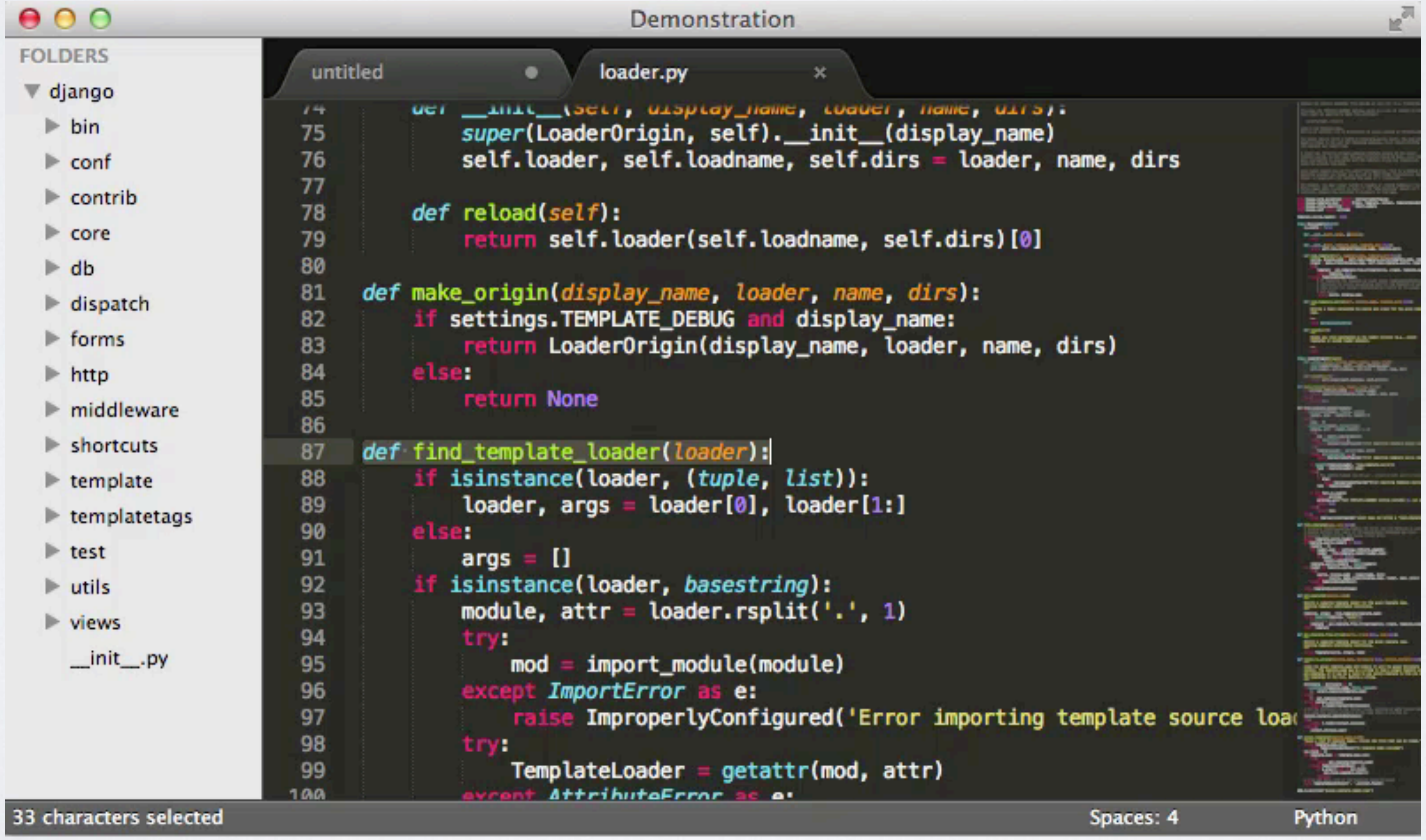
Beyond ‘calculator’ mode:

Scripts & Modules

Scripts

- A script is a file consisting of Python code.
- Create them with **plain text** editors like Notepad, TextEdit, Sublime Text, etc
- File extension should be **.py**.

Recommended text editor:
https://www.sublimetext.com/

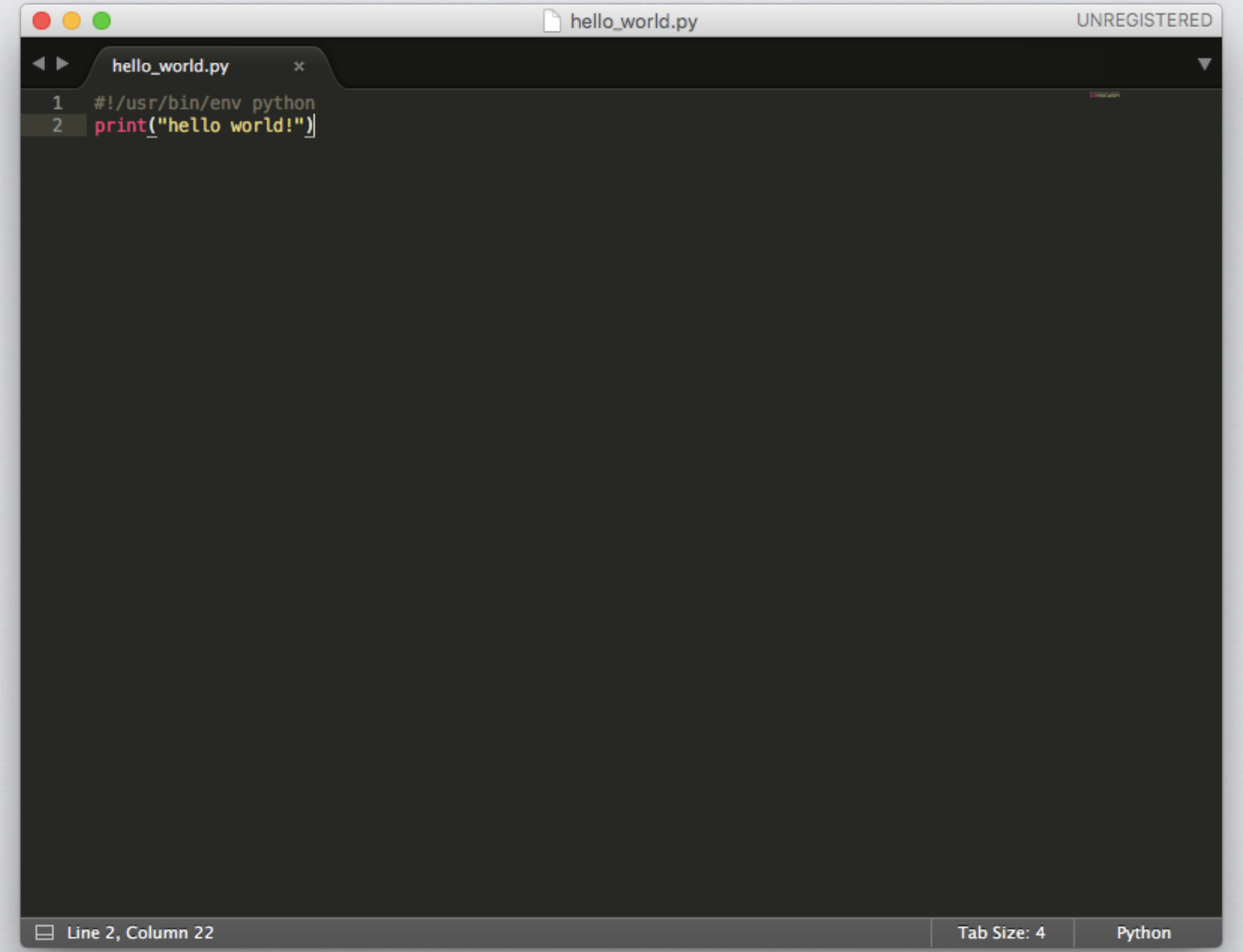


```
    def __init__(self, display_name, loader, name, dirs):
        super(LoaderOrigin, self).__init__(display_name)
        self.loader, self.loadname, self.dirs = loader, name, dirs

    def reload(self):
        return self.loader(self.loadname, self.dirs)[0]

def make_origin(display_name, loader, name, dirs):
    if settings.TEMPLATE_DEBUG and display_name:
        return LoaderOrigin(display_name, loader, name, dirs)
    else:
        return None

def find_template_loader(loader):
    if isinstance(loader, (tuple, list)):
        loader, args = loader[0], loader[1:]
    else:
        args = []
    if isinstance(loader, basestring):
        module, attr = loader.rsplit('.', 1)
        try:
            mod = import_module(module)
        except ImportError as e:
            raise ImproperlyConfigured('Error importing template source loa
        try:
            TemplateLoader = getattr(mod, attr)
        except AttributeError as e:
```

Scripts

- Writing your first script: **hello_world.py**

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

print("hello world!")
```

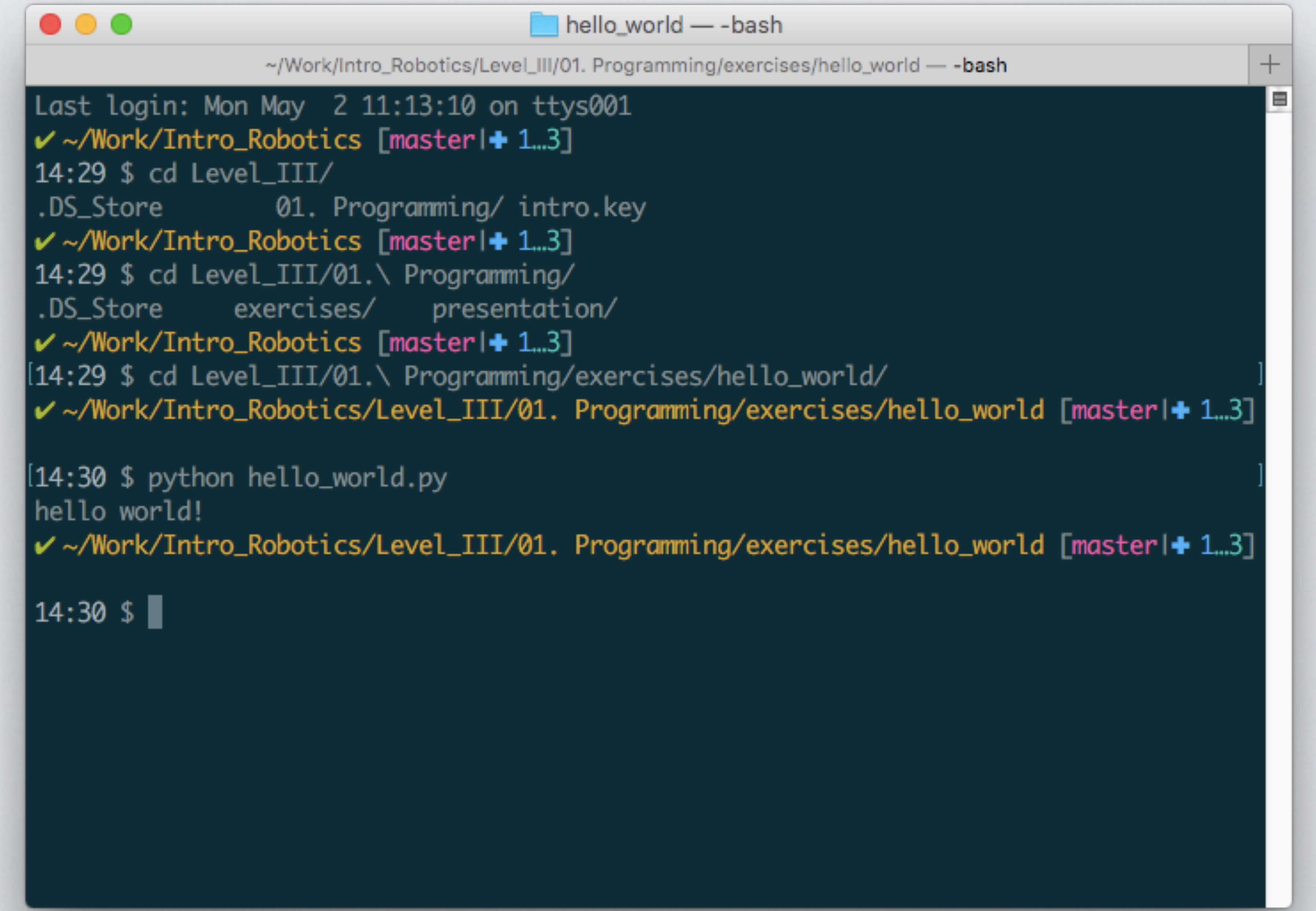


```
#!/usr/bin/env python
print("hello world!")
```

Scripts

- Running your first script (**hello_world.py**) from the Terminal app:

```
cd /path/to/the/directory

python hello_world.py
```



hello_world — -bash
~/Work/Intro_Robotics/Level_III/01. Programming/exercises/hello_world — -bash

```
Last login: Mon May  2 11:13:10 on ttys001
✔ ~/Work/Intro_Robotics [master|✚ 1…3]
14:29 $ cd Level_III/
.DS_Store        01. Programming/ intro.key
✔ ~/Work/Intro_Robotics [master|✚ 1…3]
14:29 $ cd Level_III/01.\ Programming/
.DS_Store     exercises/     presentation/
✔ ~/Work/Intro_Robotics [master|✚ 1…3]
[14:29 $ cd Level_III/01.\ Programming/exercises/hello_world/                    ]
✔ ~/Work/Intro_Robotics/Level_III/01. Programming/exercises/hello_world [master|✚ 1…3]

[14:30 $ python hello_world.py                                                   ]
hello world!
✔ ~/Work/Intro_Robotics/Level_III/01. Programming/exercises/hello_world [master|✚ 1…3]

14:30 $ █
```

Modules

- A module is a file consisting of Python code (just like scripts).
- A module contains definitions and statements (just like scripts).
- A module can define **variables**, functions, and classes (just like scripts).
- Within a module, the module's name is available as the value of the global variable **`__name__`** (just like scripts).
- Definitions from a module can be imported into other modules or into the **`main`** module (script or REPL instance).

Scripts vs. Modules

Similarities:

- Scripts and modules are files containing Python code.
- Both ‘scripts’ and ‘modules’ are executable and importable.

Differences:

- Modules typically won't do anything or will just run tests when executed.
- Modules are meant to be imported and scripts are meant to be executed.

Notes:

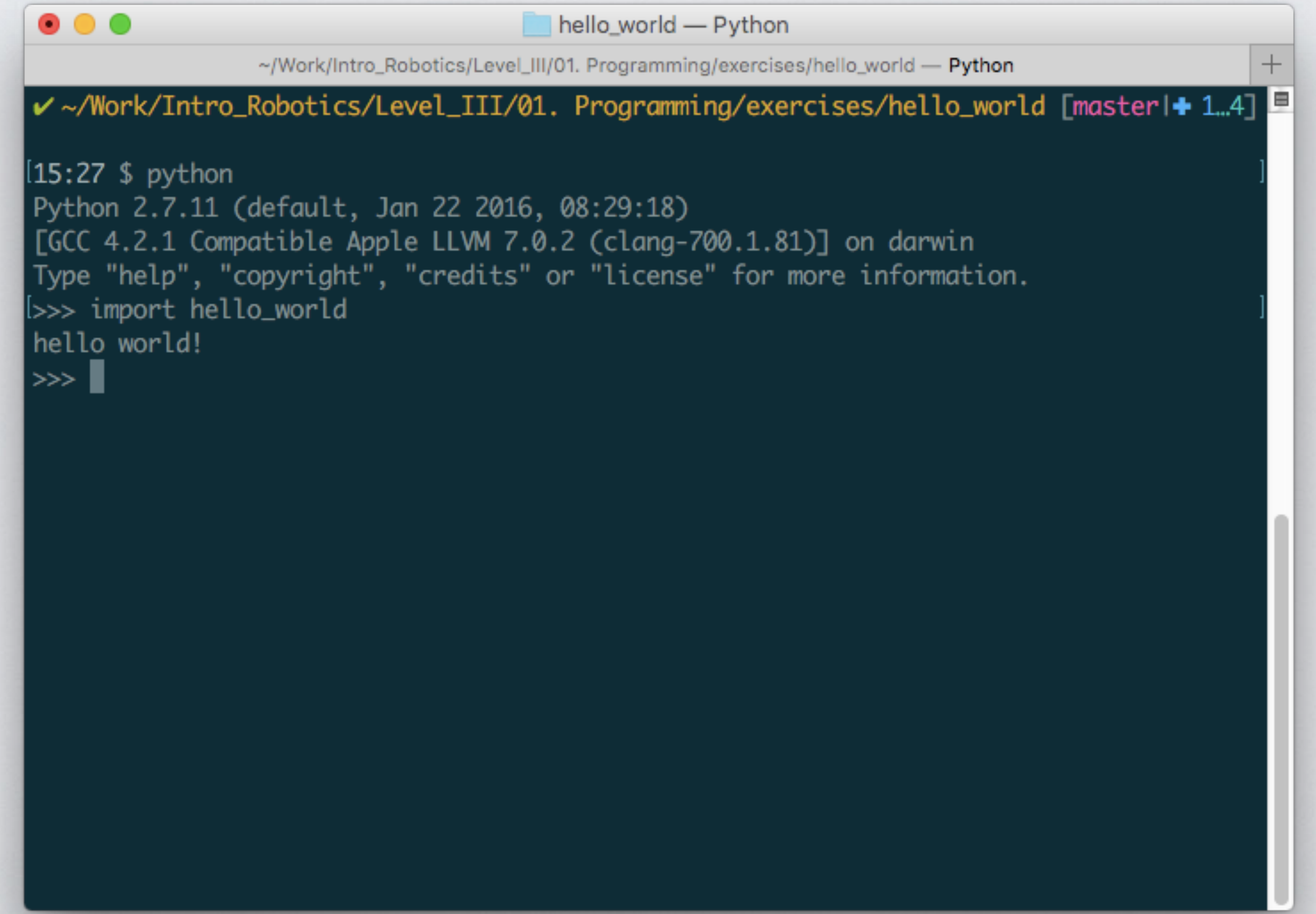
- Importing code designed to be a script will cause it to execute.

Scripts vs. Modules

- Importing code designed to be a script will cause it to execute.

```
cd /path/to/the/directory
python
```

```
>>> import hello_world
```



hello_world — Python
~/Work/Intro_Robotics/Level_III/01. Programming/exercises/hello_world — Python

```
✔ ~/Work/Intro_Robotics/Level_III/01. Programming/exercises/hello_world [master|✚ 1…4]
15:27 $ python
Python 2.7.11 (default, Jan 22 2016, 08:29:18)
[GCC 4.2.1 Compatible Apple LLVM 7.0.2 (clang-700.1.81)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> import hello_world
hello world!
>>>
```

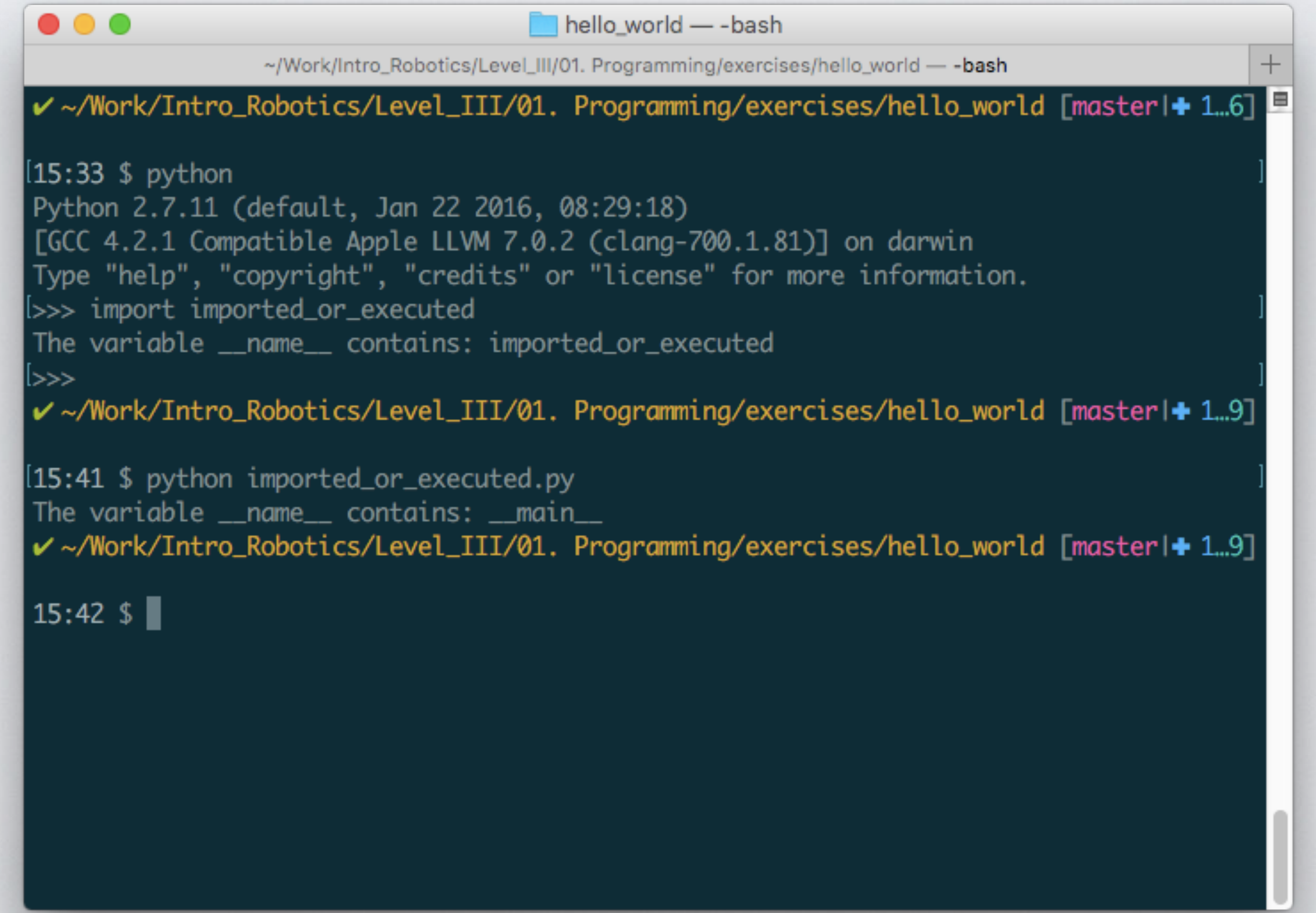
Scripts vs. Modules

- The global variable **__name__** allows us to tell if a file is executed:

```
>>> python imported_or_executed.py
```

or imported:

```
>>> import imported_or_executed
```


hello_world — -bash
~/Work/Intro_Robotics/Level_III/01. Programming/exercises/hello_world — -bash

```
✔ ~/Work/Intro_Robotics/Level_III/01. Programming/exercises/hello_world [master|✚ 1…6]

[15:33 $ python
Python 2.7.11 (default, Jan 22 2016, 08:29:18)
[GCC 4.2.1 Compatible Apple LLVM 7.0.2 (clang-700.1.81)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
[>>> import imported_or_executed
The variable __name__ contains: imported_or_executed
[>>>
✔ ~/Work/Intro_Robotics/Level_III/01. Programming/exercises/hello_world [master|✚ 1…9]

[15:41 $ python imported_or_executed.py
The variable __name__ contains: __main__
✔ ~/Work/Intro_Robotics/Level_III/01. Programming/exercises/hello_world [master|✚ 1…9]

15:42 $
```

Built-in Modules

- Python comes with a *library* of standard modules (cf. Python Library Reference).
- Some modules are built-in (written in C), others in Python.
- Example: the module **`os`** allows using operating system dependent functionality.

```
>>> import os
```

```
>>> os.mkdir("My Directory")
```

You could use the `os` module to rename a bunch of files in your computer with ease!

Built-in Modules

Example

- Create the script **delay.py**

```
#!/usr/bin/env python
import time
print("hello")
time.sleep(5)
print("world!")
```

- Run it:

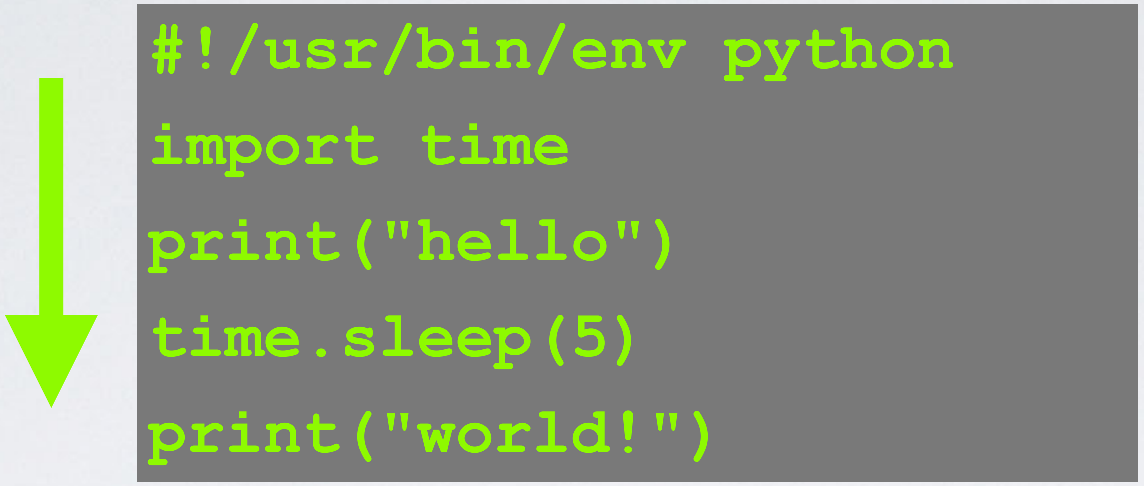
```
python delay.py
```

- When imported, modules add functionality not readily available in the core Python implementation.

Controlling code execution

Code Execution

- In a simple program code is executed from the first line going **downward**.



```
#!/usr/bin/env python
import time
print("hello")
time.sleep(5)
print("world!")
```

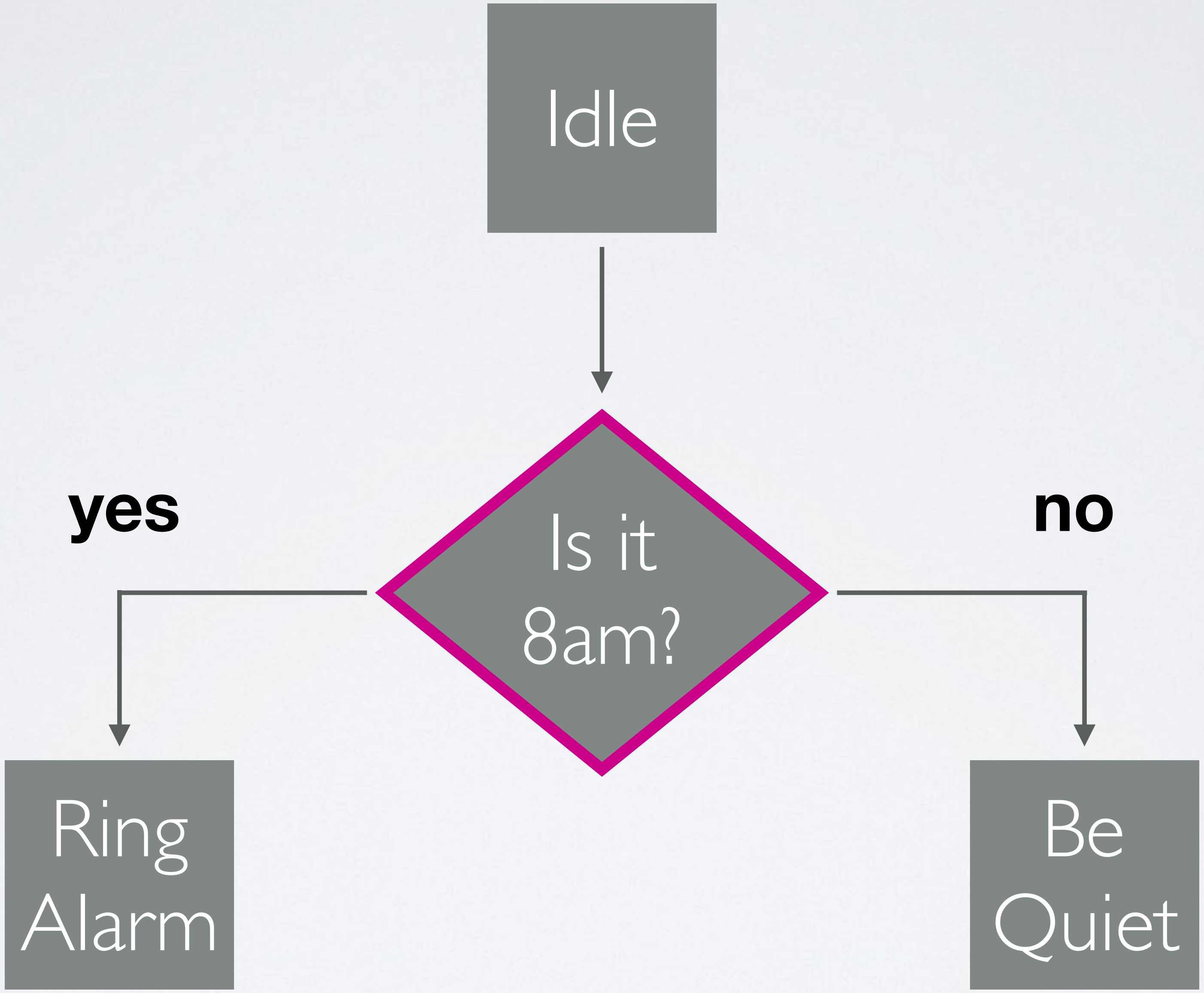
- We can use different **code structures** to change how execution occurs.

- The main code structures used for controlling **program flow** are loops and conditionals.

- Loops and conditionals heavily depend on evaluating **logical expressions**.

Truth Value Testing

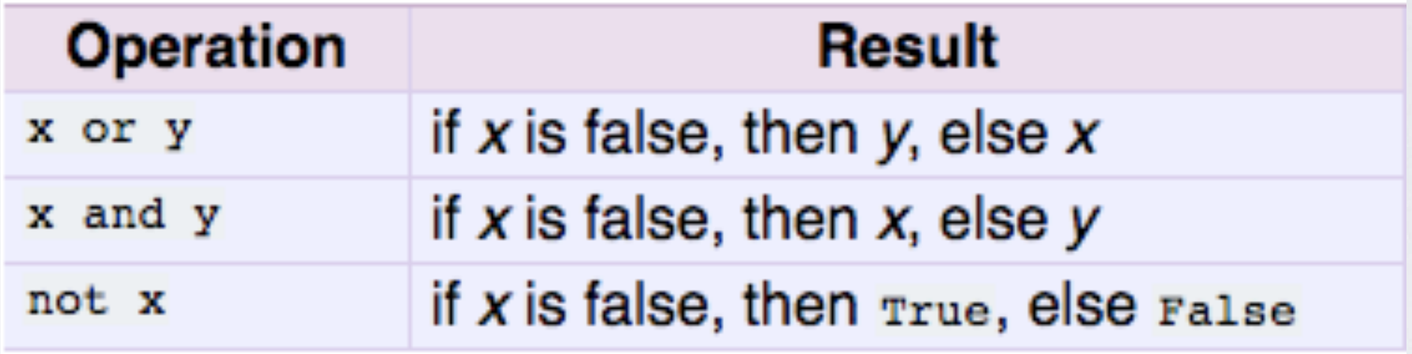
- Truth value testing is typically used as a condition to control the flow of the program.



Example: basic alarm clock

Boolean Operations

- Boolean operations are a form of algebra in which all values are reduced to either TRUE or FALSE.



Operation	Result
`x or y`	if *x* is false, then *y*, else *x*
`x and y`	if *x* is false, then *x*, else *y*
`not x`	if *x* is false, then `True`, else `False`

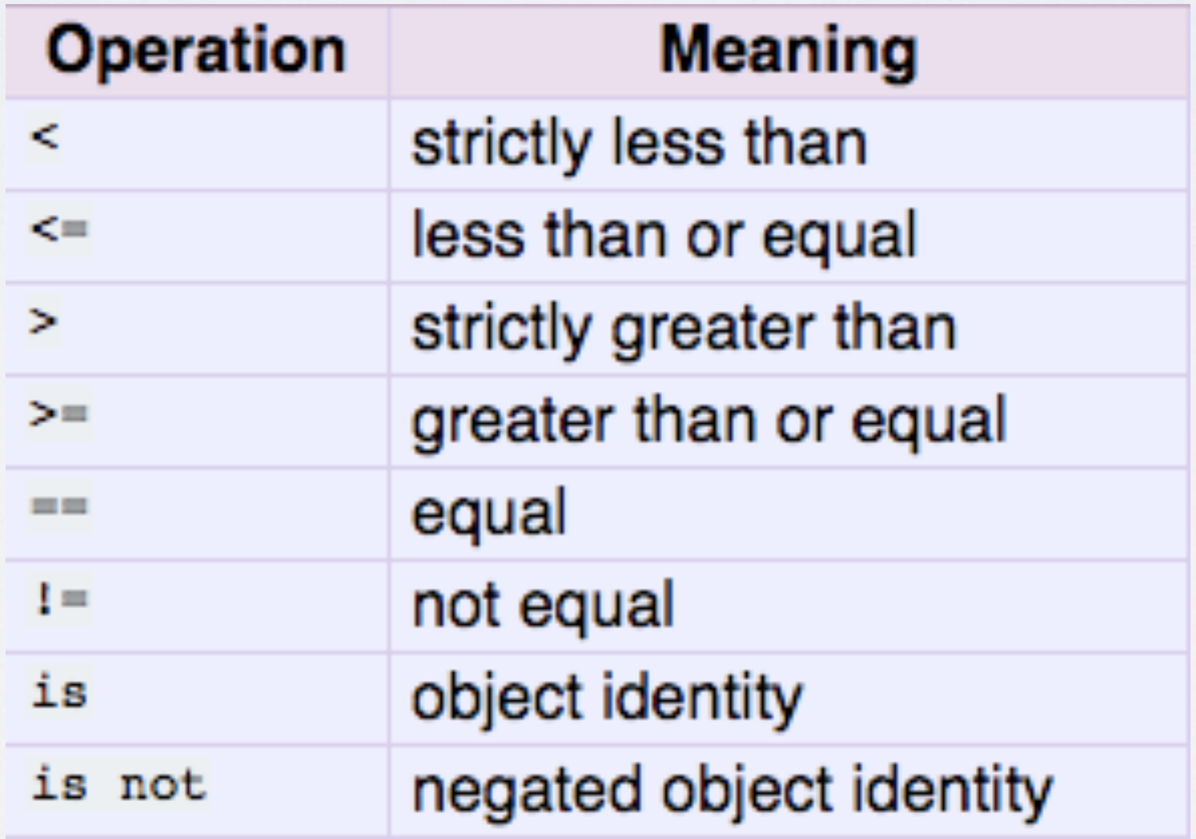
- Python considers any value to be **`True`** (Boolean) **<u>except</u>**:

  `False`, **`None`**, any empty sequence (e.g., **`''`**, **`()`**, **`[]`**), zero of any numeric type, for example, **`0`**, **`0L`**, **`0.0`**, **`0j`**.

Comparisons

- Comparison (or relational) operators compare the values on either sides of them to decide determine their relation.

- When comparisons are evaluated they return truth values (i.e., **`True`** or **`False`**)



Operation	Meaning
`<`	strictly less than
`<=`	less than or equal
`>`	strictly greater than
`>=`	greater than or equal
`==`	equal
`!=`	not equal
`is`	object identity
`is not`	negated object identity

Control of Flow: Choice

The `if` statement

- The **`if`** statement is used for performing different computations or actions depending on whether a condition evaluates to true or false

- The general form of the **`if`** statement in Python looks like this:

```
if condition_1:
    statement_block_1
elif condition_2:
    statement_block_2
else:
    statement_block_3
```

Optional

Any number of `elif`'s allowed!

Control of Flow: Choice

Example

- Create the script **choice.py**

4 spaces
for indentation

```
#!/usr/bin/env python
import random
x = random.randint(0,9)
x_str = str(x) + " "
if ((x%2) == 0):
    print(x_str + "is even")
else:
    print(x_str + "is odd")
```

- Run it:

```
python choice.py
```

Control of Flow: Choice

Quiz 1: Conditional statements

- Create the script **throw_die.py**

- When executed, the script should:

Generate a random integer between 0 and 9.

Print “value’s too small” if the result is less than 1.

Print “value’s too big” if the result is greater than 6.

Print the actual value otherwise.

Control of Flow: Choice

Quiz 2: Modules vs. Scripts

- We can use the conditional `if __name__ == "__main__":` to determine wether the current module is imported or executed.
- Create the script **test_import.py**
- When executed, the script should:

Check if the module has been executed or imported (value of the variable __name__).

Print the name of the module if it has been executed.

Print the string “__name__ isn’t __main__” if it has been imported.

Print the string “Something’s wrong!” otherwise.

Control of Flow: Loops

- The **`while`** statement is used for repeated execution as long as an expression is logically true.

- The **`for`** statement iterates over the items of any sequence (a list or a string), in the order that they appear in the sequence.

Control of Flow: Loops

Example (`while`)

- Create the script **loop_while.py**

```
#!/usr/bin/env python
import random
x = 1
while (x != 0):
    x_str = str(x) + " "
    print(x_str + "is not 0")
    x = random.randint(0,3)
print("is " + x_str + "== 0?")
```

- Run it:

```
python loop_while.py
```

Control of Flow: Loops

Example (`for`)

- Create the script **loop_for.py**

```
#!/usr/bin/env python
import random
x = 1
seq = [0,1,2,3,4,5,6,7,8,9]
for i in seq:
  x = random.randint(0,1)
  x_str = str(x)
  print("x_str" + " is " + x_str)
print("Done!")
```

- Run it:

```
python loop_for.py
```

Code structures

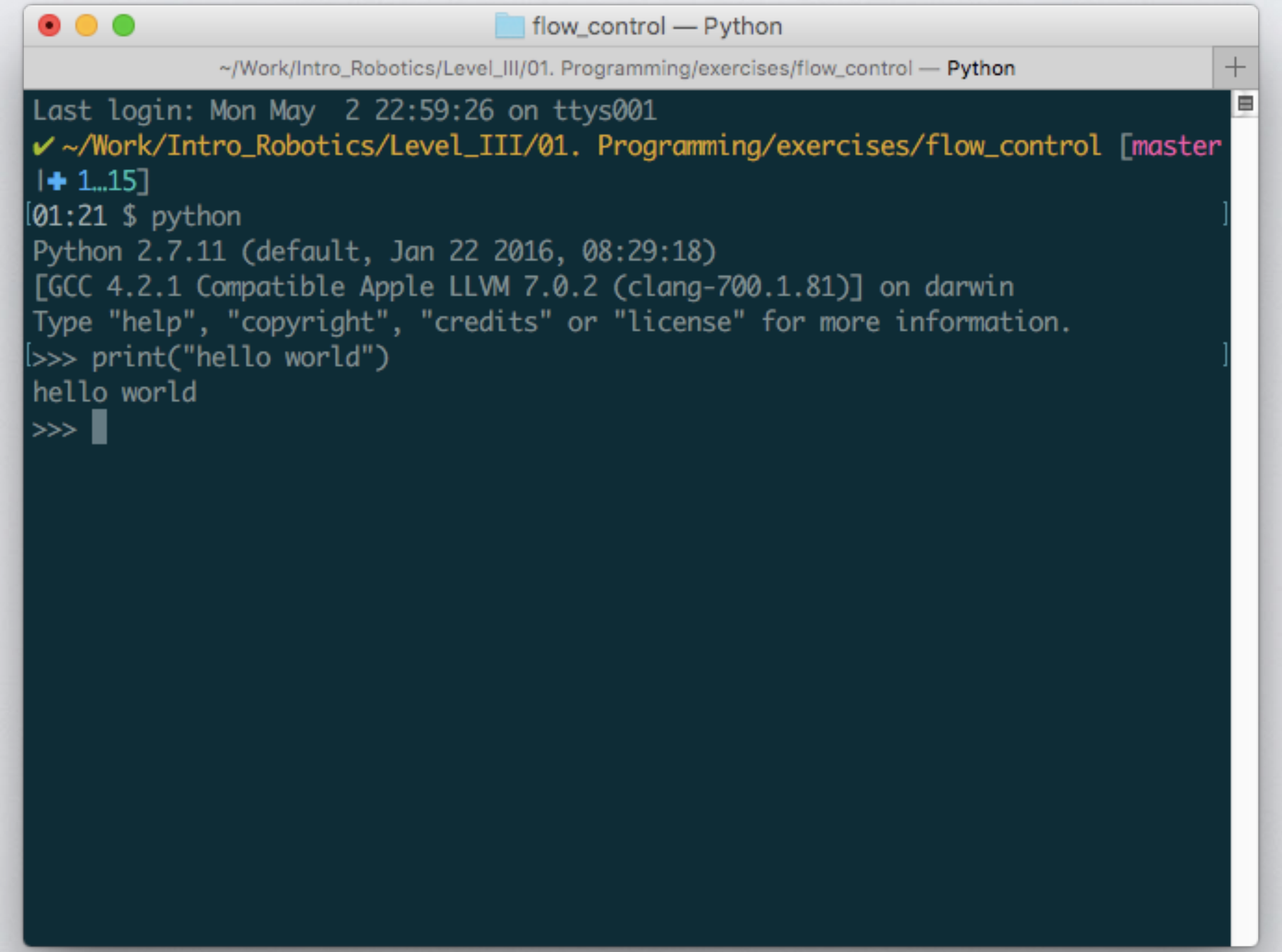
Functions

- Functions are useful blocks of code encapsulated and given a name.
- Typically functions operate on something (arguments).

```
python
```

```
>>> print("hello world!")
```

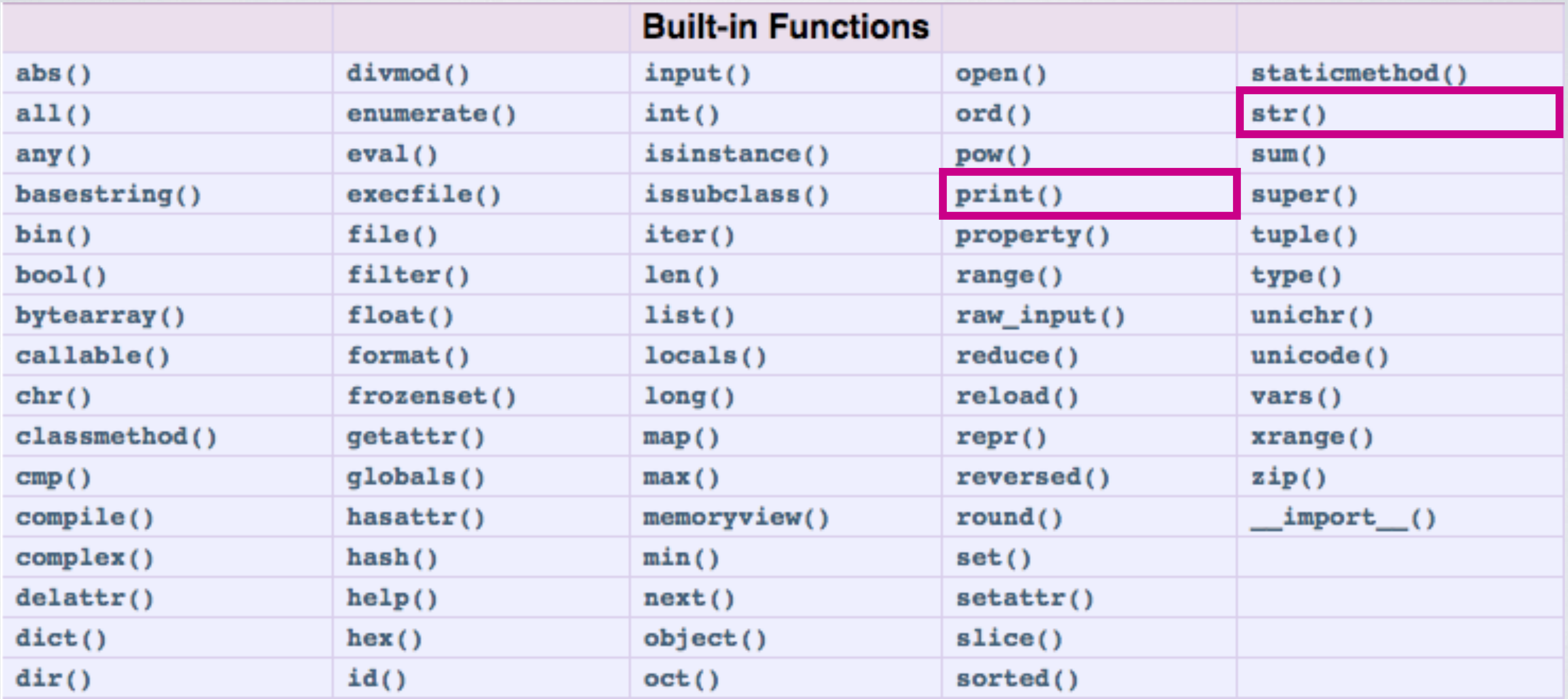
function's argument


flow_control — Python
~/Work/Intro_Robotics/Level_III/01. Programming/exercises/flow_control — Python

```
Last login: Mon May  2 22:59:26 on ttys001
✔ ~/Work/Intro_Robotics/Level_III/01. Programming/exercises/flow_control [master
|✚ 1…15]
[01:21 $ python                                                            ]
Python 2.7.11 (default, Jan 22 2016, 08:29:18)
[GCC 4.2.1 Compatible Apple LLVM 7.0.2 (clang-700.1.81)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
[>>> print("hello world")                                                  ]
hello world
>>>
```

Built-in Functions

- The Python interpreter has a number of **built-in** functions that are always available.



		Built-in Functions		
abs()	divmod()	input()	open()	staticmethod()
all()	enumerate()	int()	ord()	str()
any()	eval()	isinstance()	pow()	sum()
basestring()	execfile()	issubclass()	print()	super()
bin()	file()	iter()	property()	tuple()
bool()	filter()	len()	range()	type()
bytearray()	float()	list()	raw_input()	unichr()
callable()	format()	locals()	reduce()	unicode()
chr()	frozenset()	long()	reload()	vars()
classmethod()	getattr()	map()	repr()	xrange()
cmp()	globals()	max()	reversed()	zip()
compile()	hasattr()	memoryview()	round()	__import__()
complex()	hash()	min()	set()	
delattr()	help()	next()	setattr()	
dict()	hex()	object()	slice()	
dir()	id()	oct()	sorted()	

Built-in Functions

- The Python interpreter has a number of **constants** (**`False, True, None, ...`**) and **functions** built into it that are always available.

- The built-in function **`dir()`** if called without an argument, return the names in the current scope.

```
python
```

```
>>> dir()
```

- The argument **`__builtin__`** allows it to return the names and functions built into the interpreter.

```
>>> dir(__builtin__)
```

- The built-in function **`help()`** returns a short summary of its argument if available.

```
>>> help(dir)
```

Functions

Example

- Create the script **flip_coin.py**

```
#!/usr/bin/env python
import random
num_to_string = ["Heads","Tails"]
def flip(times):
    for i in range(times):
        x = random.randint(0,1)
        print("Coin flip is "+num_to_string[x])

x = input("How many flips you want? ")
flip(x)
```

How many functions does this program use?

- Run it:

```
python flip_coin.py
```

Custom Modules

Example

- A module can define **variables**, **functions**, and classes (just like scripts).
- Create the module **flipper.py**

```
#!/usr/bin/env python
import random
num_to_string = ["Heads","Tails"]
def flip(times):
    for i in range(times):
        x = random.randint(0,1)
        print("Coin flip is "+num_to_string[x])
```

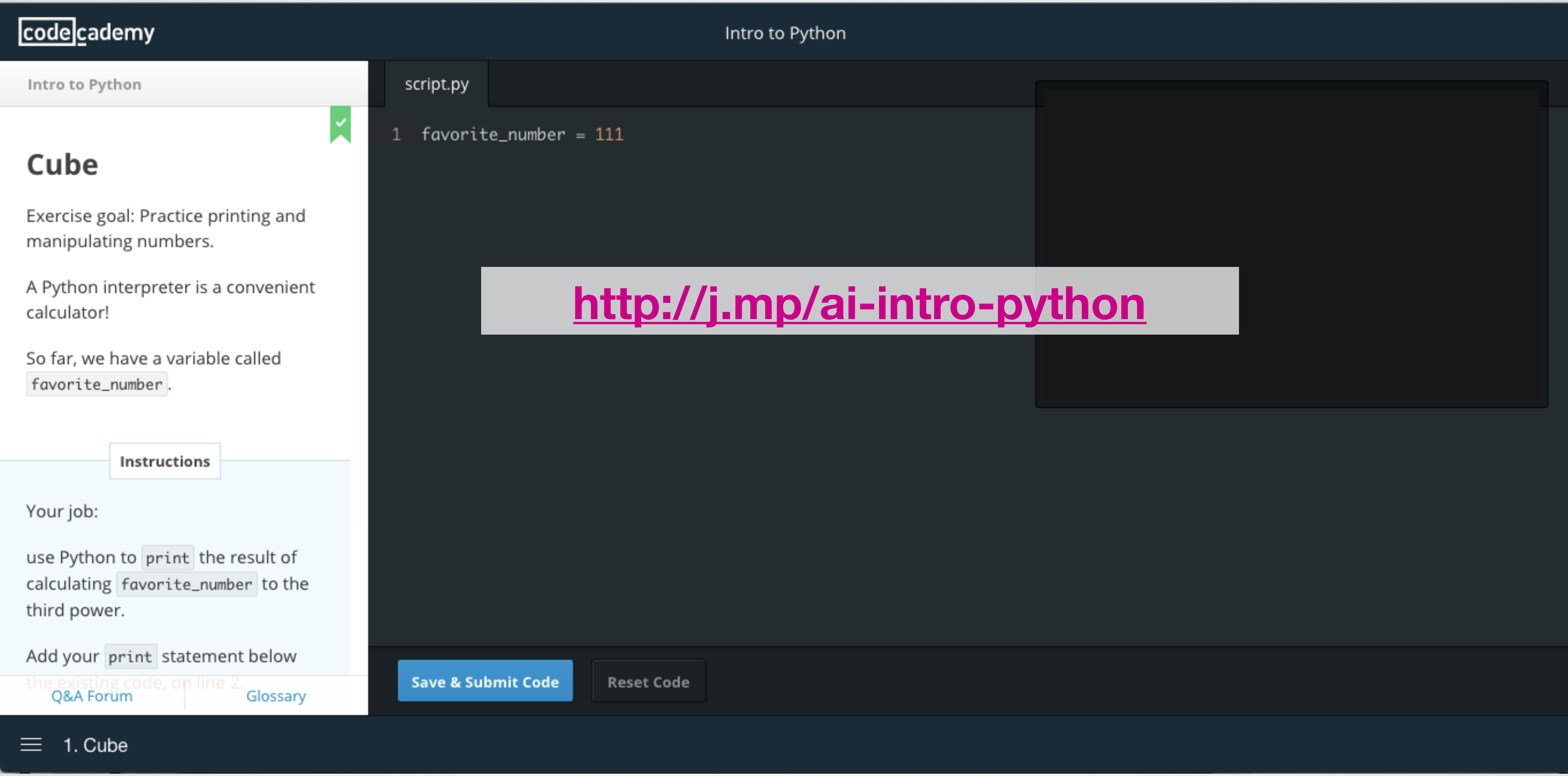
- Create the script **main.py**

```
#!/usr/bin/env python
from flipper import flip

if __name__ == '__main__':
    flip(10)
```

Mastering Your Python ABCs

Practice

- Getting some hands-on practice.



Cube

Exercise goal: Practice printing and manipulating numbers.

A Python interpreter is a convenient calculator!

So far, we have a variable called `favorite_number`.

Your job:

use Python to `print` the result of calculating `favorite_number` to the third power.

Add your `print` statement below the existing code, on line 2.

```
1  favorite_number = 111
```

http://j.mp/ai-intro-python