LECTURE 2

PYTHON REPL & SCRIPTS; ARITHMETIC

MCS 260 Fall 2020

Emily Dumas

REMINDERS

- Complete worksheet 1 this week
  - e.g. in Tue/Thu discussion
- Quiz 1 released, due Mon Aug 31 at 6pm Central

QUICK BLACKBOARD SITE TOUR

TERMINOLOGY

In this course we can treat **terminal** and **shell** as equivalent terms for a text-based interface to your operating system. PowerShell on Windows or Terminal on Mac OS X are examples.

(There is a subtle difference between the two terms, but we won't discuss it.)

The actual difference:

- **shell**: A program that listens for commands and runs them (e.g. PowerShell, bash)
- **terminal**: A system that can run the shell, give it keyboard input, show its output on screen (e.g. PowerShell, gnome-terminal, Terminal.app)

Terminals used to be physical devices. Today, the shell and terminal may be combined in a single program.

If you are typing and running commands on your computer, you are using both.

TERMINOLOGY

- **Python**: the language
- **Python interpreter**: the program you run to execute Python code

There are actually several interpreters for Python, including CPython (a name for the one we use), PyPy, Jython, and others.

INTERPRETER MODES

There are two ways to use the Python interpreter

- **Interactive mode:** Each line of code you type is executed immediately. Used for experimentation.
- **Script mode:** Execute Python code in a file. The most common way to use Python.

THE PYTHON REPL

Interactive mode is also called the **REPL** or Read-Evaluate-Print Loop: The interpreter Reads a line of code, Evaluates it, and Prints the result, all in an endless Loop.

This mode opens if you type python in the shell and press Enter.

```
$ python
Python 3.8.2 (default, Jul 16 2020, 14:00:26)
[GCC 9.3.0] on linux
Type "help", "copyright", "credits" or "license" for more informa
>>> print("MCS 260!")
MCS 260!
>>>
```

PLATFORM-DEPENDENCE NOTE

The name of the python interpreter may be "python" or "python3", or possibly something else under unusual circumstances.

The recommended ways of installing python in the startup instructions give the following names:

- Windows: python
- Mac OS X: python3
- Linux: python3 will work, python may open python 2 or 3

REPL pros:

- Quick iteration, great while learning
- Help system (covered later)

REPL cons:

- Start from scratch each time
- Results depend on history
- Inconvenient to edit larger blocks of code
- No syntax highlighting

Alternative interactive Python interfaces fix many deficiencies (e.g. iPython/Jupyter, IDLE, ...).

PYTHON SCRIPTS

Create a text file containing Python code, traditionally with extension ".py" (e.g. with VS code).

Add the name of this **script file** just after the interpreter name when running Python in the shell.

```
$ python hello.py
Hello world!
$
```

Content of hello.py:

```
print("Hello world!")
```

ARITHMETIC IN PYTHON

Python has arithmetic operators, including:

- `+` addition and `-` subtraction
- `*` multiplication
- `/` division and `//` integer division
- `**` exponentiation (`a**b` means a^b.)
- Parentheses for grouping

```
>>> 1+1
2
>>> 2*130
260
>>> 1 / (1 + 1 + 1)
0.3333333333333333
>>> 2**5
32
>>> 7/2
3.5
>>> 7//2
3
>>>
```

ORDER OF OPERATIONS

Python mostly follows the mathematical convention on order of operations.

PEMDAS is a convenient mnemonic. It means the following are listed from highest precedence (first evaluated) to lowest (last):

- **P** : parentheses
- **E** : exponentiation (e.g. 2**3)
- **MD** : multiplication, division (equal precedence)
- **AS** : addition, subtraction (equal precedence)

PEMDAS example:

```
>>> 1 + 1/2**3
1.125
```

This was evaluated as

$$1 + (1/(2^3)) = 1 + (1/8) = 1.125$$

INTEGER LITERALS

Python prints numbers in decimal, but in a script or the REPL it can read them in binary, hex, or octal.

```
>>> 0b1001
9
>>> 0xfa
250
>>> 0o775
509
```

These ways of expressing an integer that are recognized by Python are called **integer literals**.

Arithmetic can be done directly on literals regardless of base:

```
>>> 0xfa + 2
252
>>> 0o777 + 0x12
529
>>> 5**0b10
25
```

FLOATING-POINT LITERALS

Python also supports an approximation of the real number system. The approximation uses **floating-point numbers** or **floats**.

```
>>> 1.15
1.15
>>> 2.158 - 0.325
1.833
```

Keep in mind that floats are an imperfect approximation of the reals:

```
>>> 0.1+0.2
0.30000000000000004
```

SCIENTIFIC NOTATION

Floating-point literals support scientific notation, with the letter `E` or `e` taking the place of "$\times 10^{\cdots}$"

```
>>> 1e-3
0.001
>>> 500e-2
5.0
>>> 0.115e1
1.15
>>> 1e-9
1e-09
>>> 1e-3
```

COMPLEX LITERALS

Complex numbers are also supported. The Python notation for the imaginary unit is `j`, but it cannot stand on its own; it must be preceded by a floating-point literal:

```
>>> j
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'j' is not defined
>>> 1j
1j
>>> 2j+1
(1+2j)
>>> 1j * 1j
(-1+0j)
>>> 0.1 - 0.2j + 0.5 - 0.9j
(0.6-1.1j)
```

VALUES AND TYPES

Every value we work with in Python has a **type**. You can determine the type using the `type()` built-in:

str means string, a sequence of characters

```
>>> type("Hello world!")
<class 'str'>
```

int means integer

```
>>> type(77)
<class 'int'>
```

float means floating-point number

```
>>> type(0.1)
<class 'float'>
```

complex means floating-point complex number

```
>>> type(1j)
<class 'complex'>
```

Note how 77 is different from 77.0

```
>>> type(77.0)
<class 'float'>
```

Note how `"0.1"` (in quotes) is different from `0.1`:

```
>>> type("0.1")
<class 'str'>
```

Notice that the result of some arithmetic operations can be of a different type than the operands.

```
>>> 5/2
2.5
>>> type(5)
<class 'int'>
>>> type(2)
<class 'int'>
>>> type(5/2)
<class 'float'>
```

PRINTING

The `print()` function is used to print values to the terminal.

The basic syntax is `print(val1, val2, val3, ...)`.

```
>>> print("The decimal value of binary 1001 is",0b1001)
The decimal value of binary 1001 is 9
>>> print("The sum of",99,"and",0b10,"is",99+0b10)
The sum of 99 and 2 is 101
>>> print(1,1.0,1+0j)
1 1.0 (1+0j)
>>>
```

When multiple values are given, `print()` separates them with a space by default.

After it is finished printing, the cursor is moved to the next line by printing a special "newline" character.

Both behaviors can be changed, e.g. use no separator at all:

```
>>> print(1,2,3,sep="")
123
>>>
```

Use a longer string as a separator:

```
>>> print(1,2,3,4,sep="potato")
1potato2potato3potato4
```

It is also possible to disable the newline:

```
>>> print(1,2,3,end="")
1 2 3>>>
```

You can actually specify an arbitrary string to be printed at the end of the line, but usually the only relevant options are newline or nothing at all.

There's a lot more to say about printing; we'll come back to this in a later lecture (currently scheduled for Lec 13 / Wed 23 Sep).

REFERENCES

- Most of this material is discussed in Sections 1.4-1.5 of *Downey*.

ACKNOWLEDGEMENTS

- Some of today's lecture was based on teaching materials developed for MCS 260 by Jan Verschelde.

REVISION HISTORY

- 2020-08-25 Initial publication