Python, Part 1

CS 8: Introduction to Computer Science
Lecture #3

Ziad Matni
Dept. of Computer Science, UCSB

A Word About Registration for CS8

- This class is currently **FULL**
- The waitlist is **CLOSED**

Lecture Outline

- Numbers and Arithmetic in Python
- Variables in Python
- The Python Interpreter
  - Using Python IDLE tool for demos/labs
- Modules
- Functions

Yellow Band = Class Demonstration! ☺

~1990…2017…

- Derived from ABC – a language designed for learning how to program
  - By Guido van Rossum (an ABC designer) – to be a more general purpose language than ABC

BDFL Guido (1956 -)

- Open sourced since version 1.0 (1991)
  - So it is free!
  - Huge community of volunteer developers
  - Guido still the BDFL (Benevolent Dictator for Life)

- Lots of handy modules ready to use at **http://docs.python.org/**

Numbers are Objects to Python

- Each object *type* has: data and related operations

- 2 basic number types and one derived type
  - **Integers** (like `5`, `-72`) – add, subtract, multiply, …
  - **Floating point** numbers (like `0.005`, `-7.2`) – operations similar but *not exactly the same as integer* operations
  - **Complex** numbers (like `3.4 + j5`) – have *two* floating point parts, but operations are specific to complex numbers

- Expect many ***non-number object*** types later
  - But they still will have data and related operations

Problem-Solving Strategizing

- Helps to think about a problem at different scales
  - Big picture first – devise a general, overall strategy
  - Then progressively refine the overall solution by applying tactics and tools
  - Overall approach in computer science is known as *"top-down programming by stepwise refinement"*
- Best strategies, tactics and tools vary by problem
  - Idea: learn techniques applicable to many situations
- But first learn about our basic tools – computers

The Python Interpreter

- A program that performs three steps over and over and …until `exit()`
  1) It reads Python instruction statements
     - From a standard input (a.k.a. `stdin`; usually keyboard)
     - Or from a text file (usually has file ending `.py`)
  2) It executes Python commands
  3) It prints results of commands if there are any

Let's try some arithmetic with it!

Arithmetic Summary

- Operators:

`+ - * /` add, subtract, multiply, (ordinary) divide

`%` modulus operator – remainder

`( )` means whatever is inside is evaluated first

`**` raise to the power

- Special Python division operator for integers:

`//` result is truncated: `7 //2` → `3` (not 3.5)

- Precedence rules so far (will expand):

1. ()
2. **
3. *, /, %, //
4. +, -
5. =

Some Notes on Floating Point & Complex Number Operations

- Floating Point
  - Scientific Notation: "`AeN`" equivalent to "`A x 10`^{N}"
  - **A** is a real number, but **N** must be an integer (i.e. positive/negative <u>whole</u> number)

- Complex Numbers
  - Form is: `x + yj`
    - Note NO SPACE between y and j
  - All arithmetic operations return complex numbers
    - So, `5j ** 2` returns `-25 + 0j`

Comments in Python

- Anything placed after the # symbol is considered a "comment"
  - Is completely ignored by the compiler
  - Typically place commentary next to code for the benefit of others (humans) reading our code

Variables

- A **variable** is a *symbolic* reference to data
- The variable's **name** represents *what* information it contains

- They are called "**variables**" because

  --- *data can change* ---

  while **operations** on the variable remain the same
  - e.g. Variables "a" and "b" can take on different *values*, but I may always want to add them together

Variables

- Variables are like "buckets" that can keep data
  - You can label these buckets with a **name**
  - When you reference a bucket, you use its name, not the data stored in the bucket
  - You can "re-use" the buckets

- If two variables are of the same ***type***, you can perform ***operations*** on them

Variables in Python

- We assign a **value** to variables with the *assignment operator* =
  - Example: >>> a = 3

- We can change that value stored
  - Example: >>> a = 5 # it's not 3 any more!!!

Assigning Names to Variables

- Variable names are actually references
  - Like "pointers" to objects
  - Can have multiple references to the same object

```
x = 5   # x refers to an integer
y = x   # Now x and y refer to the same object
```

- Dynamic typing is a key Python feature
  - Any legal name can point to any type – even different types at different times

```
x = 1.2  # Now x refers to floating point
         # (y still refers to 14)
```

Variable Names in Python

- 3 simple rules for choosing names:
  - Can ONLY use **letters**, **digits**, and _ (underscores) only
  - Must NOT begin with a digit
  - Cannot use Python **keywords** (see Table 1.1 on p. 22)

- Also some advisories/conventions to follow:
  - Choose brief, but *meaningful* names
  - Avoid names of common Python modules, types, etc.
  - Most programmers prefer lower case – use "camel case" or underscore to separate words (`aCat`, or `a_cat`)

- All above apply to functions, modules, & types too

Objects

- An ***object*** in Python is anything that has:
  - an identity
  - a type
  - a value

- Example: `pi = 3.14159`
  - Identity: pi
  - Type: floating point
  - Value: 3.14159

- Additionally, objects can have:
  - Attributes
  - Methods

← *More on these later…*

Procedural Abstraction

- A "black box" – a piece of code that can take inputs and gives me some expected output

- A **function**, for example, is a kind of procedural abstraction

  25 → Square Root Function → 5

  - What goes on inside the function?
  - Doesn't matter, as long as it works!!

Modules and Objects

- A module is a description of an abstraction that can help with the programming
  - i.e. it's a bunch of code that does 1 specific thing and is presented to us as an abstract "black box"
  - A module can contain **multiple functions**.

- *Example*: There's a module (a "black box") called a "Piano".
  It has 12 inputs. Every input I engage results in an output:
  a certain note is played. I can also create multiple "instances" of "Piano".

The Turtle Module Example

- A "Turtle", for example is a kind of data abstraction – and it has some functions too
  - It's a simple graphics tool that's already been created for you to use

- To use it in Python, first "import" it in

```
>>> import turtle
```

- To create an "instance" of "Turtle", we do the following:

```
>>> t = Turtle.turtle()
```

← *Don't worry about why that is for now…*

Let's try it out!

Functions

- "Self contained" modules of code that accomplish a specific task.
- Functions have inputs that get processed and the function often (although not always) "returns" an output (result).
- Can be "called" from the main block of the program
  - Or from inside other functions.
- A function can be used over and over again.
  - Example: A function called "distance" that always returns the value of the distance between a point with coordinates (a, b) and the origin (0,0)

$$distance(a,b) = \sqrt{a^2 + b^2}$$

Using Functions/Methods

- To use (a.k.a. *invoke* or *call*) a function:

  `functionName(`*list of arguments*`)`

  - The list of arguments is typically all the inputs to the func.
  - These arguments are "passed" to the function
  - When function completes/is executed – we are returned to the point in the program where the function was called
    - May also return a result – it depends on the function definition

- Need "`.`" (dot operator) if the function is defined in a module or if it is a *class method* ← *Class methods are like functions… We'll worry about the differences later!*

  - Then full syntax is: `moduleName.functionName(…)`
  - Sometimes written as: `objectReference.methodName(…)`

Example of a Function Call

- Let's adopt the function we mentioned earlier:
  `distance(a, b)`

```
…
a = 3.0
b = 4.0
d = distance(a, b)
x = d – b
…
```

- What will the value of variable **d** be? What about **x**?
- Will type of variable will **d** be? And **x**?

Defining Your Own Function

- To define a function in Python, the syntax is:

```
def name ( list of parameters ):
    # a block of statements here – all indented
```

  - `def` – mandatory keyword defines a function
  - *name* – any legal Python identifier (e.g. myLittleFunction)
  - `( ):` – mandatory set of parentheses **and** colon
  - parameters – object names
    - Local references to objects (i.e. raw data or variables) that are passed into the function
    - May be an empty list!
  - e.g. `def myLittleFunction(pony1, pony2, 3.1415):`

Example Definition

```
def distance(a, b):
    x = a**2
    y = b**2
    z = (x + y) ** 0.5
    return z
```

!!OR!!

```
def distance(a, b):
    return ( (a**2) + (b**2) ) ** 0.5
```

A Function To Draw A Square

- Part of listing 1.2 from the text (p. 30)

```
def drawSquare(myTurtle, sideLength):
    myTurtle.forward(sideLength)
    myTurtle.right(90)   # side 1
    …
```

- Then to invoke it for drawing a square that has 20 pixels on each side using a turtle named `t`:

```
>>> drawSquare(t, 20)
```

- What might happen if we invoked `drawSquare(20, t)`?

Let's try it out!

Importing From A Module

- Imagine the **`drawSquare`** function is in a file called **`ds.py`**
- We have two basic choices to use this function:

1. Import whole module, and specify module to use

```
>>> import ds
>>> ds.drawSquare(t, 20)
```

2. Import part(s) of module, then just use the part(s)

```
>>> from ds import drawSquare
>>> drawSquare(t, 20)
```

"sys" is a standard module and "path" is one of its objects that stores the directory paths where your Python files will be

- Of course, Python must know where **`ds.py`** is on the computer!
  - Easy solution: store it in current directory or along **`sys.path`**

- Or in Python IDLE: *File* → *Open* – no need to import

YOUR TO-DOs

- ❑ Read **Chapter 2**
- ❑ Finish **Homework1** (due **Thursday 4/13**)
- ❑ Prepare for **Lab1**

- ❑ Hug a tree

```
>>> import turtle
>>> g = turtle.Turtle()
>>> g.forward(100)
>>> g.left(90)
>>> g.forward(100)
>>> g.left(90)
>>> g.forward(100)
>>> g.left(90)
>>> g.forward(100)

>>> h = turtle.Turtle()
>>> h.circle(100)

>>> def drawSquare(myTurtle, sideLength):
        myTurtle.forward(sideLength)
        myTurtle.right(90)   # side 1
        myTurtle.forward(sideLength)
        myTurtle.right(90)   # side 2
        myTurtle.forward(sideLength)
        myTurtle.right(90)       # side 3
        myTurtle.forward(sideLength)
        myTurtle.right(90)       # side 4

>>> t = turtle.Turtle()
>>> drawSquare(t, 20)
>>> drawSquare(t, 200)
>>> drawSquare(20, t)
```