# Introduction to Python, Part 1

**CS 8: Introduction to Computer Science, Winter 2018**
**Lecture #3**

Ziad Matni
Dept. of Computer Science, UCSB

# A Word About Registration for CS8

- I will let a few people into the class today from the waitlist.

- After that, this class will be **FULL**,

  & the waitlist will be **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! ☺**

# Note: Difference Between **Python IDLE and Python Programs**

- *Python IDLE* is the program we use to demonstrate Python in class
  - You can also use it at home to do one line at a time Python code

- If you want to create a *Python program*, then you will place *all* the program code inside a text file
  - Text file always ends in **.py**
  - You can *run (execute)* the program from Python IDLE

# 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 in the quarter…
  - But they also 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

# 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

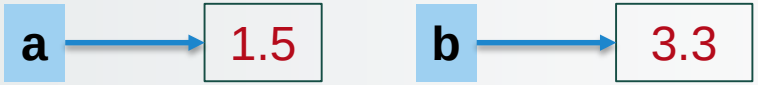
- **Floating Point**
  - Can use Scientific Notation: "`AeN`" equivalent to "`A x 10^N`"
  - **A** is a real number, but **N** must be an integer (i.e. positive/negative whole number)
- **Complex Numbers**
  - Form is: `x + yj`
    - Note NO SPACE between y and j
  - All arithmetic operations return 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 VARY or 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
```

# Assigning Names to Variables

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

```
x = 5       # x refers to an integer
y = x       # Now x and y refer to the same object
x = 1.2     # Now x refers to floating point 1.2
            # (y still refers to the integer 5)
```

# Variable Names in Python

**3 simple rules for choosing names:**

- Can ONLY use **letters**, **digits**, and _ (underscores) only
  - So, **UserName**, **Age1**, **Age2**, **_Deviation** are ok

- Must NOT begin with a digit or non-alphabet character (except underscore)
  - So, **2Good2BTrue**, **$$MaMoney!!**, **<0_0>**, **#YOLO** won't work…

- Cannot use Python **keywords** (see Table 1.1 on p. 22)
  - Example: **def**, **False**, **True**, **print**, etc…

# Variable Names in Python: Other Conventions

- Choose brief, but *meaningful* names
- Most programmers prefer lower case use
  - Example: total vs. TOTAL
- Use either “camel case” or underscore to separate words
  - Camel Case is using capital letters to separate words, like `NumOfCats`
  - Underscoring is using underscores to separate words, like `num_of_cats`
  - Be consistent: use one or the other

- All the above applies to function names, module names, etc…

# 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…*

# Demo

Let’s try this out – what do you think it’ll do?

```
pi = 3.14159
radian_angle = 0.7853975
degree_angle = radian_angle*180/pi
print(degree_angle)   # What is print()?
```

**Let’s try it out!**

# Procedural Abstraction: The Function

- 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’s happening inside the function?
  - Doesn’t matter, as long as it works!!

# Functions

- A function does “something” to one/several input(s) and sends back one/several output(s)
  - Always has braces to “carry” the inputs

- Example: the sqrt() function (square root)
  - With an input of 25, I expect an output of 5
  - That is, **sqrt(25)** will give me **5**

# More About 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!

# More About Functions

- A function can be used over and over again.
  - Example:
    Consider a function called "***distance***" that returns the value of the distance between a point w/ coordinates (a, b) and the Cartesian origin (0, 0)

$$\textbf{\textit{distance (a, b) = square root of } } (a^2 + b^2)$$

- We will learn how to craft functions later on…

# Modules and Objects

- A **module** is a description of an abstraction that can help with the programming
  - Sooooo…. It's a function?
  - Nooooo…. It's a mega-function, of sorts…
  - And it can be "objectified", unlike functions
    - Libraries, Classes, etc… More on those later

- A module can contain **multiple functions** and we can "call it up" as different versions of the same thing

# Example: Modules & Objects

Let's say, there's a module (a "black box") called a "Piano".
It has 12 inputs (keys that play notes). Every input I engage the inputs, an output is the result – a certain note is played.

I can also create multiple "instances" or "objects" of the module "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

```
>>> t.forward(50)
>>> t.right(90)
>>> t.forward(50)
>>> t.right(90)
>>> t.forward(50)
>>> t.right(90)
>>> t.forward(50)
```

- To use it in Python, first “import” it in

```
>>> import turtle
```

- To create an “instance” of “Turtle”, do the following:

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

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

**Let’s try it out!**

# How Do We “Call” A Function?

- 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 function
- These arguments are “passed into” the function
- When function completes/is executed – we are returned to the point in the program where the function was called
  - It may also return a result – it depends on the function definition

- Need to use the “**.**” (dot operator) if the function is defined inside a module
  - 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)`**

```
… # inside the Python code…
a = 3.0
b = 4.0
d = distance(a, b)
x = d – b
… # more down here
```

- 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 functionName (list of parameters):
    # a block of statements appear here
    # all of them must be indented (with tabs)
```

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

# Example Definition

```
# This function calculates the distance between (a,b) and (0,0)
def distance(a, b):
      x = a**2     # Note the tab indent!!!
      y = b**2     # Recall ** means “to the power of”
      z = (x + y) ** 0.5
      return z     # I need to “return” the result
```

**!!! Alternatively !!!**

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

**Let’s try it out!**

# 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)
```

**Let's try it out!**

# Importing From A Module

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

1. Import the whole module, and specify the part of the 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)
```

# Importing From A Module

- Of course, Python **must be told *where*** `ds.py` is on the computer!
- How do we do that?
  - Store the file in the same directory where you're running Python (also known as the "current directory")
  - Place the *pathname* in `sys.path`
    - This is a little involved and you might need help with it
    - "sys" is a standard Python module and "path" is one of its objects that stores the directory paths where your Python files will reside
  - In Python IDLE, Go to *File* → *Open* and open `ds.py`

# YOUR TO-DOs

- ❑ Read **Chapter 2**
- ❑ Finish **Homework1** (due **Monday!**)
- ❑ Prepare for **Lab1** next week

- ❑ Hug a tree

# </LECTURE>