Instructor: Paul Nguyen (paul_nguyen [at] utexas [dot] edu)
TAs: Farzam Farahani, Eric Wei

Introduction to Python

THE APTAMER STREAM

Spring 2016

"Ars longa, vita brevis." (Art is long, life is short.)

Week 1: A 50-Minute Language Overview

Course overview

- Lectures will be fast-paced (at first) so you can have exposure to enough material to be able to write interesting programs
- Lectures are for *exposure*, exercises are for *learning*
  - Feel free to ask questions during lectures
- I am not assigning grades; this course is for your own personal development and enrichment
  - This means that you should focus on work for your actual classes first
- Topics: Python syntax, programming concepts, (very) basic data structures and algorithms, applications to biology
- Recommended: try solving bioinformatics problems on Rosalind (http://rosalind.info/)
  - Some of the exercises I assign may be inspired by Rosalind problems
- Set up a convenient folder (in Dropbox, on the Desktop, etc…) for your Python scripts
- Independent effort on your part is required
- You can collaborate with others, but struggle on your own first

Installing Python

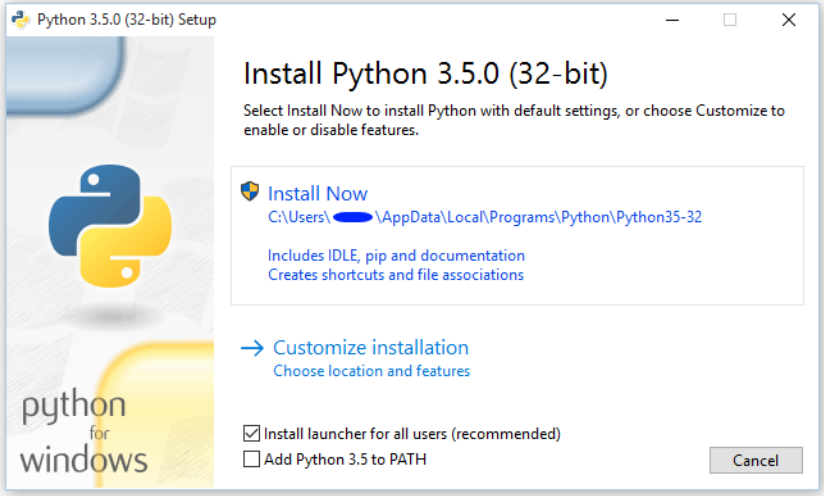
Visit https://www.python.org/downloads/

Directions linked on the mini-course website under "Readings"

Mac OS and Linux users may already have Python

- However, do not use Python 2.x
- Type `python --version` in Terminal to check which version you have

Make sure to use **Python 3.x** (latest: **3.5.1** as of March 2016)



http://python.readthedocs.org/en/latest/using/windows.html

Interactive mode

In Command Prompt/PowerShell/Terminal: type `python` to enter Python's interactive mode

In IDLE: IDLE automatically launches in interactive mode

Expressions are combinations of values, variables, and operators

Expressions to try in interactive mode (use Python as a calculator):

`2 + 2`	`2 ** 3`
`3.6 / 3`	`7 - 9 * 8`
`3.6 // 3`	`(7 - 9) * 8`
`17 % 4`	`9 + 2 // 3`

What do you think `//`, `%`, and `**` do?

Each expression in the table above **returns** a **value**

Three distinct numeric **types**: **int** (integers), **float** (floating-point numbers), and **complex** (complex numbers, which we will likely not use in this course)

- Integers: 4, 99, -2, 0
- Floating-point numbers: 3.3, 3.33, 7.0, -2.3, 7e9

First program (script mode)

In IDLE, go to File > New File to begin writing a new script. Type the following line.

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

Save this in a folder with a sensible name, like "hello.py".

Then, run your program (in IDLE, go to the Run > Run Module or press F5).

`print()` is a **function** (more on functions later).

`"Hello world!"` is a **string** that is passed to the `print()` function as a **parameter**.

You can print out multiple items on the same line using `print()` as long as the items are separated by a comma.

Strings

Strings are basically sequences of characters.

They are surrounded by straight quotation marks (single or double, as long as they match on both sides of the string).

Do not use curly quotes.

Examples of strings:

```
"Hello"
"test string"
"12345"
"testing"
"2 + 2"
"ACTGGTCGATCGAT"
```

" " (straight)

“ ” (curly)

Variables

If you want to hold on to data, **assign** it to a **variable.**

Examples:

```
pi = 3.14159
```

```
greeting = "Bonjour."
```

Variable **identifiers** (names) are often lowercase and derived from English words, but you can use almost any sequence of letters, numbers, and underscores, as long as the sequence **begins with a letter** and is **not a reserved keyword** (a word reserved for the Python language).

Not valid names:

```
2chainz
#winning
space bar
```

Reserved keywords

```
False      class      finally    is         return
None       continue   for        lambda     try
True       def        from       nonlocal   while
and        del        global     not        with
as         elif       if         or         yield
assert     else       import     pass
break      except     in         raise
```

Don't try to use any of these for variable names!

Variables cont.

You can reassign values to variables you have already assigned values to.

Example:

```
number = 2
print(number)
number = 7 * 7
print(number)
number = "This is a string."
print(number)
```

Variable updating

You can update variables by adding a value to the value already stored by the variable.

Example:

```
x = 4
print(x)
x = x + 2
print(x)
x += 2
print(x)
```

Besides addition, you can also subtract, multiply, divide, and even use the modulo (remainder) operator (%).

String concatenation

The + operator is useful for more than adding numbers; you can also use it to **concatenate** (combine) strings, creating a new string.

```
string_one = "Hello "
string_two = "world!"
print(string_one + string_two)
```

You cannot "subtract" a string from another string the way you can concatenate strings together.

You cannot concatenate a string and an integer without converting the integer to a string first using `str()`.

Comments

Use the pound sign (#) to mark off the rest of the line as a comment that will not affect the running of the program. This is useful for leaving notes to yourself or for disabling code while debugging.

Example:

```
number = 3
# number = 4
print(number) # TODO: add new features
```

Commenting is very important! Why?

You might leave a project for a few months, then come back to it and not remember why you wrote the original code that way.

You may work on a team where other people may have to build upon your code but have not seen it before and need to get up to speed quickly.

Getting user input

You can prompt the user to type in something, like his/her name, and save the value that is given.

This is useful for numerous things, like text-based games.

Use the `input()` function to get input ; like `print()`, `input()` accepts a string **parameter** (or **argument**).

```
name = input("Enter your name: ")
print("Hello " + name + "!")
```

The string passed as a parameter to `input()` is the prompt that will be displayed to the user.

Try to come up with some creative ideas for using user input!

Logic and control flow

- Logic
  - `if` statement
  - `else` statement
  - `elif` statement
- Preview of lists
- Loops
  - `for` loops (`range()` function in Week 2)
  - `while` loops
- More control flow
  - `break`, `continue`, `pass` statements (Week 2)

Boolean values (`True` and `False`)

In Python, the objects `True` and `False` are **Boolean values**.

When you perform a test using **relational (comparison) operators** (like <, >=, or !=), the test returns one of these values.

Examples:

```
3 > 2 # returns True
3 < 2 # returns False
```

At home: what do you think would happen if you tried to add a number to a Boolean value in Python?

Examples:

```
1 + True # can you guess what happens?
3 + False # what happens here?
```

`if` statements

`if` statements allow certain actions to be performed when a specific **condition** is met.

Example:

```
x = 5
if x > 2:
    print("x is greater than 2")
```

Here, `x > 2` is the condition. The `if` statement checks if the condition is **true** or **false**; if the condition is true, the code under the `if` statement is executed.

Specifically, `x > 2` returns a value of `True` in the case above because x was assigned the value of 5.

At home: Truth values

What will this code output?

```
if 238:
    print("success!")
```

How about this?

```
if 0:
    print("success!")
```

How about this?

```
if -49:
    print("success!")
```

`else` statements

`else` statements are often seen with `if` statements. The code under an `else` statement runs when a condition is not met.

```
name = input("Please enter your name: ")
if name == "Paul":
    print("Greetings, " + name + "!")
else:
    print("Hello, " + name + ".")
```

Nesting `if` statements

When you have multiple conditions you want to check, one thing you can do is **nest** `if` statements.

```
if grade > 90:
    if grade > 93:
        if grade > 97:
            print("Grade = A+")
        else:
            print("Grade = A")
    else:
        print("Grade = A-")
else:
    print("Grade is less than A-")
```

(There are better ways to write this code, by the way.)

`elif` statements

`elif` is short for "else if" and helps reduce the indentation you need.

```
if name == "Paul":
    print("Greetings, " + name + "!")
elif name == "Farzam":
    print("Salutations, " + name + "!")
else:
    print("Hello, " + name + ".")
```

Logical operators: and

You can evaluate whether multiple conditions are all true by using the **logical operator** and.

Example:

```
x = 20
if x > 5 and x < 21:
    print("Success 1")
    # "Success 1" is printed
if x > 5 and x > 21:
    print("Success 2")
    # "Success 2" is not printed
```

Logical operators: or

You can evaluate whether *at least* one condition in a set of conditions is true by using the **logical operator** or.

Example:

```
x = 20
if x > 5 or x < 21:
      print("Success 1")
      # "Success 1" is printed
if x < 5 or x < 21:
      print("Success 2")
      # "Success 2" is printed even though the first
      # condition is false
if x < 5 or x > 21:
      print("Success 3")
      # "Success 3" is not printed
```

Logical operators: not

`(not a)` returns `True` if a returns `False` and returns `False` if a returns `True`. This is called **negation**.

Example:

```
x = 20
print(x > 3) # prints "True"
print(not (x > 3)) # prints "False"
if x > 5 or x < 21:
     print("Success 1")
     # "Success 1" is printed
if not (x > 5 or x < 21):
     print("Success 2")
     # "Success 2" is not printed
```

At home: summary (truth table)

a	b	a and b	a or b	not a
True	True	True	True	False
True	False	False	True	False
False	True	False	True	True
False	False	False	False	True

A sneak peek at lists

A **list** is a **mutable** sequence of items. By "mutable," we mean that you can change the list (such as by adding items to the list or changing the items in the list).

You can construct lists with brackets. Lists are zero-indexed, meaning that the first item in a list has index 0, the next item has index 1, and so on…

You can retrieve the first item from list a by typing `a[0]`.

Examples:

```
a = [] # empty list
b = [2, 3, 5, 7]
c = ["Hello", "Goodbye"]
d = ["Hello", 32]   # lists can contain items of
                    # different types
```

Something to think about…

Notice that strings are sequences of characters (as mentioned earlier), while lists are sequences of items!

Just like with indices for list items, you can use [] with indices for characters in strings.

```
string_one = "ABCDEFG"
string_one[1] # returns "B"
```

In the future, we will talk about **slicing** as well as about **tuples**, another built-in sequence type in Python. We will also talk more about useful sequence operations.

for loops

for can be used to **iterate** over sequences such as lists! This is extremely powerful. We will talk more about iteration in the future.

```
fruits = ["apple", "banana", "cherry"]
for fruit in fruits:
     print(fruit)

numbers = [2, 3, 4, 5]
for number in numbers:
     number = number * 2
print(numbers)
```

for loops cont.

Since for loops work on sequences (because sequences are **iterable**), you can use for loops on strings too!

```
vowels = ["A", "E", "I", "O", "U"]
string = "facetiously"
vowel_count = 0
for letter in string.upper(): # make string uppercase
     if letter in vowels:
          vowel_count += 1
print("The number of vowels is: " + str(vowel_count))
```

`while` loops

`while` loops continually execute the statements within while a condition is met. Be careful about accidentally writing infinite loops! You should either update the condition that the `while` loop is checking or otherwise `break` out of the loop. This example program causes x to eventually be greater than 5 by adding 1 to x at the end of each pass through the loop. When x is greater than 5, the condition is not met.

```
x = 0
while x <= 5:
    print(x, x*x)
    x += 1
```

Week 2:

Functions

Very important concept!

Functions take inputs and (usually) do something with those inputs and/or return an output.

- `print()` is a function.
- `type()` is a function that tells you the type of a value.

Examples (type conversion, int to string):

```
a = 74
b = str(a)
type(a)
type(b)
```

Similar functions: `int()`, `float()`

Note: `int()` does not round; it simply removes the fractional part of a number!

```
c = int(5.55) # returns 5
```

The `math` module

Python has a useful module called `math` that provides you with extra functions you might want for calculations.

First, in your script or in interactive mode, type the following: `import math`

- This loads the `math` module so that you can use certain functions in your program.
- Functions from modules are (traditionally) accessed by typing the name of the module, a dot, and the function name.
  - Example: `math.sqrt(16)`
  - This is useful because it prevents you from accidentally calling the wrong function in case you have imported another module with a function with the same name as the one you meant to use.
  - `math.sqrt()` is a function that returns a value. We can use this value inside other functions!
  - Example: `math.pow(math.sqrt(16), 3)`

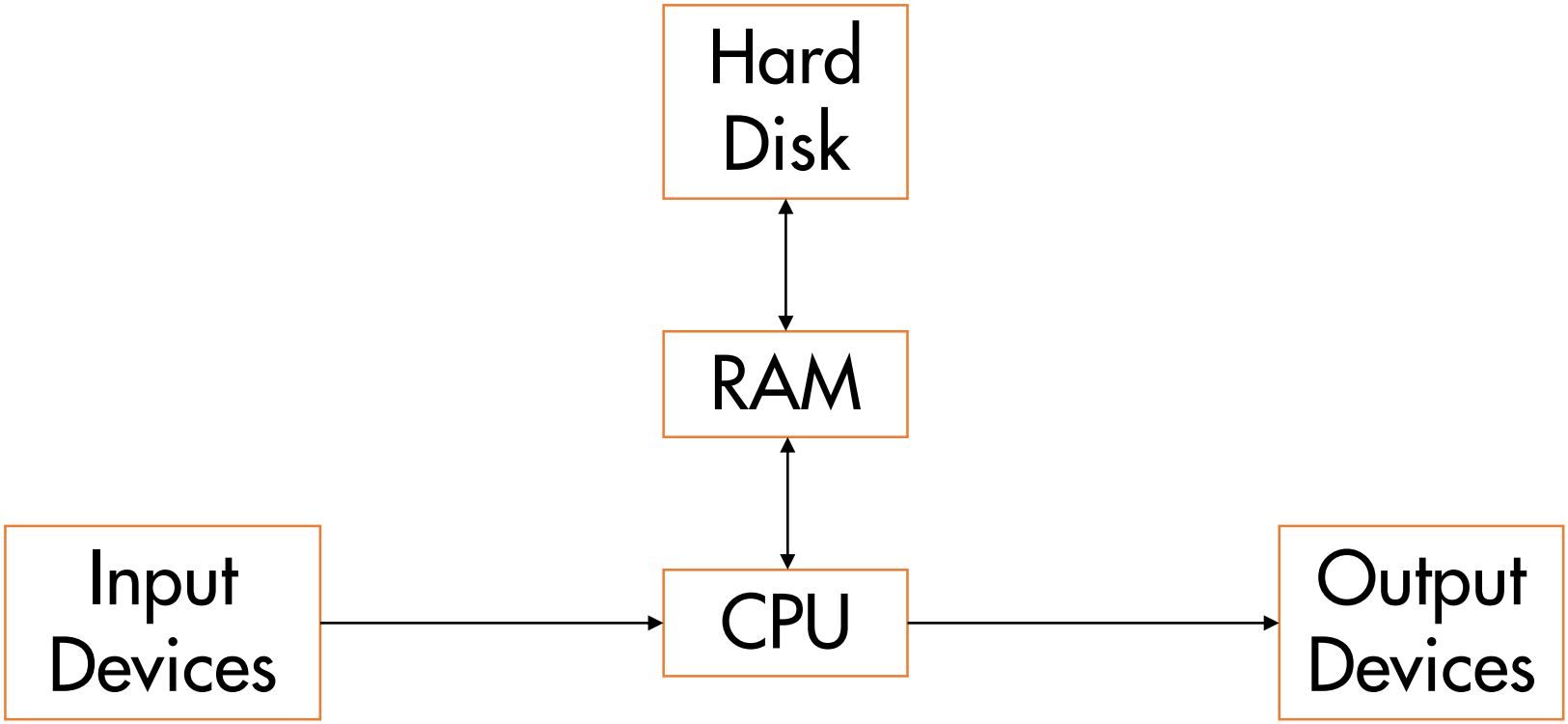
Defining custom functions

```
def myfunction():
    print("This is my function.")
    print("There are many like it.")
    print("But this one is mine.")
myfunction() # call our function
```

Notice the indentation on the `print()` lines. Python uses indentation to indicate lines of code in the same **block**, while other languages would surround blocks with braces instead. You can indent with either a tab character or a fixed number of spaces (typically 2 or 4).

This function does not take any arguments, but you could easily make it so (stay tuned).

Introduction to computers



Variables and memory

Basically, values are stored in RAM (which serves as **main memory** or **primary memory**). There are many locations in RAM at which to save values, and each location has a **memory address**. A variable name is essentially a label for an address. When you reassign a variable, you are changing the address that the "label" is referring to.

Python takes care of memory management for our needs, so we are likely not going to talk about manual memory allocation in this course (specifically, Python uses a garbage collector to reclaim memory).

Relevant links:

https://en.wikipedia.org/wiki/Computer_data_storage#Primary_storage
https://en.wikipedia.org/wiki/Memory_address

*Notes on floating-point

*"On a typical machine running Python, there are **53 bits of precision available** for a Python float…" – Python Documentation (emphasis added)*

When you print floating-point numbers in Python, the output can be misleading because a) Python uses binary fractions (base 2 instead of base 10) to store floats and b) there are only a limited number of bits available.

From the documentation:

If Python were to print the true decimal value of the binary approximation stored for 0.1, it would display

```
>>> 0.1
```

```
0.1000000000000000055511151231257827021181583404541015625
```

This is a lot of digits, so Python displays a rounded number instead.

```
>>> 0.1
```

```
0.1
```

Try adding 0.1 to 0.2 and see what happens. Compare what happens when you add together ten copies of 0.1 and when you just multiply 0.1 by 10.

Other languages have this problem, so just keep this in mind when dealing with precise numbers. Use integers when you only need to deal with integers!

To be continued…

At home: short-circuiting (might be slightly confusing at first)

When `and` is used, if the first argument is false, then the second argument will not even be checked.

Similarly, with `or`, if the first argument is true, then the second argument will not be checked.

Why is this useful?

Sometimes the arguments are functions that take in data and will return True or False, and perhaps the second function should only be run if the first function returns `True` to indicate that the data is valid (for example, when checking to see if a file has a certain word, your first function could check whether the file exists; if it doesn't, then the second function (the one checking for the word) won't even run due to short-circuiting).

Relevant link: https://en.wikipedia.org/wiki/Short-circuit_evaluation