An Introduction to Programming with Python

Variables, types, statements, functions
Last Time

- Brief History of Computing
- Short Intro to Python

Due:
- PA1
- PS1
- Academic Integrity Pledge
Reminders

- Lab Tonight
- OLI Iteration Module Tonight
- PS2 (i.e. oli quiz 1) due for tomorrow
Today’s Lecture

- Introduction to Python
- Mechanics

Some Specifics:
- Basic datatypes
- Variables
- Operators
- Expressions
- Functions
Questions

- How was the OLI module?
Execution of Python Programs

Python Program

Applications

Python Interpreter

Operating System

Hard Drive

Monitor
A programming “language” is a formal notation
Recipe

Toast and cereal
Toast the bread. Butter it. Put some cereal in a bowl. Add mikl.

- Interpreted by a person
- Unclear? Can be figured out (What kind of bread? Butter? What kind of milk?)
- Typos? Can be figured out (“mikl” means “milk”)

Computer program

```python
for i in range(5):
    pritn(whatever I want)
```

- Interpreted by a machine
- ...for a human (“somebody wants to print something”)
- Unclear? Not a program (“whatever I want”???)
- Typos? Program errors (“pritn”???)
A programming “language” is a formal notation for generalised problem solving.
Programs should be *general*

**Program**

- `def force(mass, accel):

  return mass*accel`

Specific: “output” is two cups of sauce.

General: output is force for *any* combination of mass and acceleration.

**Recipe**

**Program**

- `def force(mass, accel):

  return mass*accel`

**Specific:**

- “output” is two cups of sauce.

**General:** output is force for *any* combination of mass and acceleration.

**SWISS CHEESE & WHITE WINE SAUCE**

- 1/4 c. butter
- 4 tbsp. flour
- 2 c. milk
- 1 c. Swiss cheese
- 1/2 c. white wine
- Salt & pepper

Make a roux, heat the milk, and when the roux is cooked, add some of the warm milk. Break or grate the cheese and stir it into the sauce until it is melted. Now add the rest of the milk and wine. Season with salt and pepper. Makes 2 cups.
Python

- Python is one of many programming languages.
- 2 widely used versions. We will use Python 3.
- Running Python on the command line:

  > python3

  or

  > python3 -i filename.py
Be aware of the difference between “talking to the shell” and “talking to Python”

```
$ ssh annpenny@linux.andrew.cmu.edu
annpenny@linux.andrew.cmu.edu's password: ...
[annpenny@unix2 ~]$ pwd
/afs/andrew.cmu.edu/usr14/annpenny
[annpenny@unix2 ~]$ python3
Python 3.3.2 (default, Aug 12 2013, 13:12:23)
[GCC 4.6.3] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> quit()
```
Expressions and Statements

- *Know the difference!*

  Python evaluates an expression to get a *result* (number or other value)

  Python executes a statement to perform an action that has an *effect* (printing something, for example)
Variables
Data Types

- **Integers**
  - 4
  - 15110
  - -53
  - 0

- **Floating Point Numbers**
  - 4.0
  - 0.80333333333
  - 7.34e+014

- **Strings**
  - "hello"
  - "A"
  - ""
  - ""
  - 'there'
  - ''
  - '15110'

- **Booleans**
  - True
  - False
Arithmetic Expressions

- Mathematical Operators
  +  Addition
  -  Subtraction  //  Integer division
  *  Multiplication  **  Exponentiation
  /  Division  %  Modulo (remainder)

- Python is like a calculator: type an expression and it tells you the value.

```
>> 2 + 3 * 5
⇒ 17
```
Order of Evaluation

Order of operator precedence:

**  * / %  + -

Use parentheses to force alternate precedence

\[ 5 \times 6 + 7 \neq 5 \times (6 + 7) \]

Left associativity except for **

\[ 2 + 3 + 4 = (2 + 3) + 4 \]
\[ 2 \times 3 \times 4 = 2 \times (3 \times 4) \]
Integer Division

In Python3:

- 7 / 2 equals 3.5
- 7 // 2 equals 3
- 7 // 2.0 equals 3.0
- 7.0 // 2 equals 3.0
- -7 // 2 equals -4
  - beware! // rounds down to smaller number, not towards zero
A variable is *not* an “unknown” as in algebra.

In computer programming, a variable is a *place* where you can store a value.

In Python we store a value using an *assignment statement*:

```
>>> a = 5
>>> a
5
```
Variables

Expression

\[ \gg a \]
\[ \Rightarrow 5 \]

Assignment statement

\[ \gg b = 2 \times a \]
\[ \gg b \]
\[ \Rightarrow 10 \]

Computer memory

\[ a: 5 \]
\[ b: 10 \]
Variables

>> a
⇒ 5
>> b
⇒ 10
>> a = "Woof"
>> a
⇒ "Woof"
>> b
⇒ 10

Variable b does not “remember” that its value came from variable a.
All variable names must start with a letter (lowercase recommended).

The remainder of the variable name (if any) can consist of any combination of uppercase letters, lowercase letters, digits and underscores (_).

Identifiers in Python are case sensitive. Example: *Value is not the same as value.*
# Syntax vs. Semantics

<table>
<thead>
<tr>
<th>Syntax</th>
<th>Semantic</th>
</tr>
</thead>
<tbody>
<tr>
<td>Rules, structure</td>
<td>Meaning</td>
</tr>
<tr>
<td>Errors result when code is not well formed.</td>
<td>Error results when expression/statement can’t be evaluated or executed due to meaning.</td>
</tr>
</tbody>
</table>

Colorless green ideas sleep furiously
Function Syntax

```python
def functionname (parameterlist) :
  instructions
```

- `def` is a reserved word and cannot be used as a variable name.

- *Indentation is critical.* Use spaces only, **not tabs**!
Functions are general

- The parameter list can contain 1 or more variables that represent data to be used in the function’s computation.

- A function can also have no parameters – but now it can only do one thing!

```python
def hello_world():
    print("Hello World!\n")
```

(\n is a newline character)
Example: area of a countertop

\[
\frac{4}{2} = 2
\]
```python
def compute_area():
    square = 4 * 4
    triangle = 0.5 * (4 / 2) * (4 / 2)
    area = square - triangle
    return area
```

To call (use) the function in `python3`:

```
python3 -i countertop.py
>>> compute_area()
14.0
```
Generalizing the problem

\[ \frac{X}{2} \]
def compute_area(side):
    square = side * side
    triangle = 0.5 * (side / 2 * side / 2)
    area = square - triangle
    return area

To call (use) the function in python3:

```bash
python3 -i countertop.py
>>> compute_area(109)
29
```
Function Outputs

A function outputs a value by `return`

def three_x(x):
    return x * 3

... or it might do some action and (by default) return `None`:

def hello_world():
    print("Hello World!\n")
Method Outputs

- >>> three_x(12)
  36  → value returned
  >>> print(three_x(12))
  36  → value returned and printed

- >>> hello_world()
  Hello World!  → value printed by method
  >>> print(hello_world())
  Hello World!  → value printed by method
  None  → value returned and printed
To use a method, we “call” the method.

A method can return either one answer or no answer (`None`) to its “caller”.

The `hello_world` function does not return anything to its caller. It simply prints something on the screen.

The `three_x` function does return its result to its caller so it can use the value in another computation:

```
three_x(12) + three_x(16)
```
Suppose we write `compute_area` this way:
```python
def compute_area(side):
    square = side * side
    triangle = 0.5 * side/2 * side/2
    area = square - triangle
    print(area)
```

Now the following computation does not work. Why?
```python
compute_area(109) + compute_area(78)
```
Lots of math stuff, e.g., sqrt, log, sin, cos

```python
import math
r = 5 + math.sqrt(2)
alpha = math.sin(math.pi/3)
```
Using predefined modules

- **math** is a predefined module of **functions** (also called **methods**) that we can use without writing their **implementations**.

```
math.sqrt(16)
math.pi
math.sin(math.pi / 2)
```
What Could Possibly Go Wrong?

```
alpha = 5
2 + alpha

3 / 0
import math

math.sqrt(-1)
math.sqrt(2, 3)
```
Try

- Create a function that calculates 18% tip
- `input("Enter your check’s total: ")` would return a user-entered variable. Write a short python script that would advise users of an appropriate tip based on their input.

- Create a function that takes two parameters (mass and radius) and calculates escape velocity. Note:
  - \( G = 6.67 \times 10^{-11} \)
  - Our fine planet has mass of 5.9742e+24, and a radius of 6378.1

\[
\nu_{esc} = \sqrt{\frac{2GM}{R}}
\]
Remember

- Next Lecture: Programming with Python
  - Note resources link and tutorials have extra info on getting running with python

- Tonight:
  - Lab 2
  - OLI Iteration Module

- For tomorrow (9:00):
  - PS2
All commands must be typed in lower case.

`pwd` --> print working directory, prints where you currently are

`ls` --> list, lists all the files and folders in the directory

`cd` stands for 'change directory':
- `cd lab1` --> change to the lab1 directory/folder
- `cd ..` --> going up one directory/folder
- `cd ../..` --> going up two directories
mkdir lab1 --> make directory lab1 aka makes a folder called lab1

rm -r lab1 --> removes the directory lab1
   (-r stands for recursive, which deletes any possible
    folders in lab1 that might contain other files)

cp lab1/file1.txt lab2 --> copies a file called file1.txt, which is inside of the folder lab1, to the folder lab2

mv lab1/file1.txt lab2 --> moves a file called file1.txt, which is inside of the folder lab1, to the folder lab2

zip zipfile.zip file1.txt file2.txt file3.txt -->
   zips files 1 to 3 into zipfile.zip

zip -r zipfile.zip lab1/ --> zips up all files in the lab1 folder into zipfile.zip
Useful Unix Commands (Part 3)

^c --> ctrl + c, interrupts running program

^d --> ctrl + d, gets you out of python3

"tab" - autocompletes what you're typing based on the files in the current folder

"up" - cycles through the commands you've typed. Similarly for the opposite effect, press "down"
**Useful Unix Commands (Part 4)**

**python3 -i test.py -->** load test.py in python3, and you can call the functions in test.py.

**gedit lb1.txt & --&gt;** opens up lb1.txt on gedit and & allows you to run your terminal at the same time (else your terminal pauses until you close gedit)

And lastly, you can always do man &lt;command&gt; to find out more about a particular command you're interested about (eg. man cp, man ls)