Administration

- Office hours
  - Held in BA 2270 at M4-6, F2-4
- The second ramp-up session hasn't happened yet.
  - Saturday 10am - 4pm
  - In BA 3185
  - Register on the CSC 148 website.
- Help centre is now open.
  - BA 2270 M-R 2-4
Administration

• Exercise 1 is up, premarking will go live tomorrow.

• If you don't have a cdf account/can't login yet, talk to the cdf support staff.
  • Try to login to Markus tonight or tomorrow, and let me know if you can't.

• Anonymous Feedback.

• Some people have asked for more detailed python installation instructions.
  • I will do them tomorrow post pre-marking setup.
Last Week

• Variables.
  • a name that refers to some value.
  • assigned with:
    
    name = expression
  • The expression is any legal python statement that can evaluate to one value.
  • variable names can consist of digits, letters and underscores.
  • convention in python is to use pothole_case.

May 24 2012
Last Week

• Functions.
  • A way to reuse code.
  • created by:
    
    def name(parameters):
      block
  
  • called by:
    
    name(expressions)
  
  • Will evaluate to None or the return value if one exists.
Why functions?

• Allow us to reuse bits of code, which makes updating and testing much easier.
  • Only need to test and update the function, rather than every place that we use it.

• Chunking! Allows us to parse information much better.
  • Human mind is pretty limited in what it can do.
  • Function names allow us to have a shorthand for what a function does.
Functions in detail

- We missed or didn't cover a lot of stuff in the first lecture.
  - print vs. return.
  - variable scope.
  - nesting function calls.
  - designing functions
  - function documentation.
Aside: Command Line Python

- Python can be run from the command line.
  - Usually referred to as a terminal in OS X/Linux
  - Start -> run -> cmd.exe in Windows.
- Can run python files with
  - `python file_name.py`
  - `python` will just run the shell.
- Command line python allows one to use python in scripts, and is faster.
Print vs. Return

- Recall that functions end if they see a return statement, and return the value of the expression after the keyword return.
  - If there is no return statement, the function returns None.
- We've also seen snippets of the print statement.
  - Print takes one or more expressions separated by a comma, and prints them to the screen.
  - This is different than a return statement, but looks identical in the shell.
Variable scope

- Scope refers to the area in which a variable is defined.
  - If there is an undefined variable the code will crash.
  - Knowing scope is key to being able to trace code.
- There are two types of variables:
  - Local variables defined in functions
  - Global variables defined in the body of the program.
Local Variables.

```python
def name(parameters):
    block
```

- Defined within a function.
  - They exist only during a function call.
  - They stop existing once the function call is resolved, and are recreated if the function is called again.
  - The parameters are viewed as local variables.
Local Variables.

def name(parameters):
    block

• Defined within a function.
  • They exist only during a function call.
  • They stop existing one the function call is resolved, and are recreated if the function is called again.
  • The parameters are viewed as local variables.
Global variables

• Defined outside of a function.

• Exist between function calls.

• Cannot be changed by a function call!
Global variables

- Defined outside of a function.
- Exist between function calls.
- Cannot be changed by a function call!

```python
def name(parameters):
    block1

    block2
```

Local Scope

Global Scope
Global variables

- Defined outside of a function.
- Exist between function calls.
- Cannot be changed by a function call!

Local Scope

Global Scope

```python
def name(parameters):
    block1
    block2
```
Variable name overlap

- It is possible for local and global variables to have the same name.
- If this occurs, python will use the local variable.
- In general, if python sees a variable name, it will try and use as local a variable name as possible.
Nesting Function calls

• Sometimes we want to have functions calling other functions.
  • f(g(4))

• In this case, we use the 'inside out' rule, that is we apply g first, and then we apply f to the result.

• If the functions can have local variables, this can get complicated.
Variable Lookup

• First, check local variables defined in a function.

• Then check local variables in an enclosing function.
  • That is for \( f(g(4)) \) it will check \( g \)'s local variables first, and then \( f \)'s local variables.

• Then check global variables.
How to think about scope.

- We use namespaces.
- A name space is an area in which a variable is defined.
- Each time we call a function, we create a local namespace.
- We refer to that first, and go down to the enclosing functions name space or global namespace as necessary.
**Namespaces**

```python
def f(x):
    return x + 4

def g(y):
    return f(y) + 10

z = 14
z = z + g(z)
```

*Global namespace*
def f(x):
    return x + 4

def g(y):
    return f(y) + 10

z = 14

z = z + g(z)
Namespaces

def f(x):
    return x + 4

def g(y):
    return f(y) + 10

z = 14

z = z + g(z)
Call Stack

• The mechanism through which python does lookups.

• Python starts with a lookup table for global variables.
Lookup Table

• Variables on one side, memory addresses on the other.

• Useful to write something that indicates what namespace the look up table refers to.

<table>
<thead>
<tr>
<th>Global</th>
</tr>
</thead>
<tbody>
<tr>
<td>y: 0x2</td>
</tr>
<tr>
<td>x: 0x3</td>
</tr>
</tbody>
</table>
Call Stack

- The mechanism through which python does lookups.
- Python starts with a lookup table for global variables.
- Each time a function call is evaluated a new lookup table for local variables is created.
- This table is put 'on top' of the currently extant tables.
Call Stack

• To look up a variable one tries to find it in a lookup table.
• Start at the top, and go down until one finds a lookup table that contains the variable one is looking for.
• If one can't find it, the program crashes.
• Note: A variable can only exist at most once in a given lookup table.
Call Stack example.

```python
def f(x):
    return x + 4

def g(y):
    return f(y) + 10

z = 14

z = z + g(z)
```
def f(x):
    return x + 4

def g(y):
    return f(y) + 10

z = 14

z = z + g(z)

Global

z: 0x1
Call Stack example.

def f(x):
    return x + 4

def g(y):
    return f(y) + 10

z = 14
z = z + g(z)

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>g</td>
<td></td>
</tr>
<tr>
<td>y:</td>
<td>0x1</td>
</tr>
<tr>
<td>Global</td>
<td></td>
</tr>
<tr>
<td>z:</td>
<td>0x1</td>
</tr>
</tbody>
</table>
Call Stack example.

```python
def f(x):
    return x + 4

def g(y):
    return f(y) + 10

z = 14
z = z + g(z)
```

<table>
<thead>
<tr>
<th></th>
<th>f</th>
</tr>
</thead>
<tbody>
<tr>
<td>x:</td>
<td>0x1</td>
</tr>
<tr>
<td>y:</td>
<td>0x1</td>
</tr>
<tr>
<td>Global</td>
<td></td>
</tr>
<tr>
<td>z:</td>
<td>0x1</td>
</tr>
</tbody>
</table>
Why do we care about Namespaces and Call Stacks?

- Understanding this will make tracing easier.
  - The better this can be internalised, the more one can trace code without needing to explicitly write things down.
- Useful for debugging.
- Common stumbling block for beginners.
Break, the first
Global or Local Variables?

- Functions can reference global variables.
- Global variables can also be passed to functions.
Global or Local Variables?

- Functions can reference global variables.
- Global variables can also be passed to functions.
- The latter is strongly preferred.
  - The former tends to make code hard to read and prone to errors.
- Global variables tend to be used only for constants that will never change.
Designing Functions

• Need to choose parameters.
  • Ask “what does the function need to know”.
  • Everything it needs to know should be passed as a parameter.
  • Do not rely on global parameters.

• Need to choose whether to return or not to return.
  • Functions that return information to code should return, those that show something to the user shouldn't (print, media.show(), etc).

May 24 2012
Function Documentation

- Recall that we can use the built-in function `help()` to get information on functions or modules.
- We can do this on functions that we've defined as well, but it doesn't give much information.
- We can add useful documentation with docstrings.
  - A docstring is surrounded by "" and must be the first line of a module or function.
Docstrings

- If the first line of a function or module is a string, we call it a docstring.
  - Short for documentation string.
- Python saves the string to return if the help function is called.
- Convention: Leave a blank line after but not before a docstring.
- The first line of a docstring should contain information about the parameter and output types.
Docstrings

- The first line of a docstring should contain information about the parameter and output types.

  (int, float) -> int
  picture -> NoneType
  NoneType -> float
Why Docstrings?

• If you write the docstring first, you have an instant sanity check.

• Makes portability and updating easier.
  • Allows other people to know what your functions do and how to use them, without having get into the code.
  • Allows for good chunking.

• Every Function should have a docstring!
Writing Good Docstrings.

- ""A sunset module.""
- ""Changes into a sunset.""
- These are terrible docstrings.
  - They are vague and ambiguous. They don't tell us what the function expects or what it does.
- How can we make it better?
Writing Good Docstrings.

- Describes what a function does.
- ""Changes into a sunset."
- ""Makes a picture look like it was taken at sunset."
- ""Makes a picture look like it was taken at sunset by decreasing the green and blue by 70%.""
Writing Good Docstrings.

- Describes what a function does.
- "Changes into a sunset."
- "Makes a picture look like it was taken at sunset."
- "Makes a picture look like it was taken at sunset by decreasing the green and blue by 70%."
Writing Good Docstrings.

- Does not describe how a function works.
  - More useful for chunking, and it's unnecessary information if we're using the function.

- "Makes a picture look like it was taken at sunset."

- "Makes a picture look like it was taken at sunset by decreasing the green and blue by 70%."
Writing Good Docstrings.

- Does not describe how a function works.
  - More useful for chunking, and it's unnecessary information if we're using the function.

- ""Makes a picture look like it was taken at sunset.""

- ""Makes a picture look like it was taken at sunset by decreasing the green and blue by 70%.""
Writing Good Docstrings.

- Makes the purpose of every parameter clear and refers to the parameter by name.
- "'Makes a picture look like it was taken at sunset.'"
- "'Takes a given picture and makes it look like it was taken at sunset.'"
- "'Takes a picture pic and makes it look like it was taken at sunset.'"
Writing Good Docstrings.

- Makes the purpose of every parameter clear and refers to the parameter by name.
- "Makes a picture look like it was taken at sunset."
- "Takes a given picture and makes it look like it was taken at sunset."
- "Takes a picture pic and makes it look like it was taken at sunset."
Writing Good Docstrings.

• Be clear if a function returns a value, and if so, what.

Consider `average_red(pic)`

• ""Computer the average amount of red in a picture.""

• ""Returns the average amount of red (a float) in a picture pic.""
Writing Good Docstrings.

- Make sure to explicitly state any assumptions the function has.
  def decrease_red(pic, percent)
- """Decreases the amount of red per pixel in picture pic by int percent. percent must be between 0 and 100."""
Writing Good Docstrings.

- Be concise and grammatically correct.
- Use commands rather than descriptions.
- """Takes a picture pic and makes it appear as it if was taken at sunset.""
- """Take picture pic and make it appear to have been taken at sunset."""
Writing Good Docstrings.

- Docstrings do not include definitions or hints.
- The docstring for sqrt is not:
  "Return the sqrt of (x). The sqrt of x is a number, that when multiplied by itself evaluates to x'.
- Is it simply:
  - Return the square root of x.
Writing Good Docstrings.

- Describes what a function does.
- Does not describe how a function works.
- Makes the purpose of every parameter clear and refers to the parameter by name.
- Be clear if a function returns a value, and if so, what.
- Make sure to explicitly state any assumptions the function has.
- Be concise and grammatically correct.
- Use commands rather than descriptions.
Break, the second.
Adaptive Programs

- We've seen programs that are executed line by line.
  - Even if they had function calls, we could expand these to something that was line by line.
- This is very limited.
  - Can't make choices, adapt to information.
Booleans: A new type.

• Can have two values True, False.
• Have three operations: not, and, or.
  • not changes a True to a False and vice versa.
  • and returns False unless all the arguments are True.
  • or returns True unless all the arguments are False.
Truth Tables

- A way of representing boolean expressions.

<table>
<thead>
<tr>
<th></th>
<th>y</th>
<th>not x</th>
<th>not y</th>
<th>x and y</th>
<th>x or y</th>
</tr>
</thead>
<tbody>
<tr>
<td>T</td>
<td>T</td>
<td>F</td>
<td>F</td>
<td>T</td>
<td>T</td>
</tr>
<tr>
<td>T</td>
<td>F</td>
<td>F</td>
<td>T</td>
<td>F</td>
<td>T</td>
</tr>
<tr>
<td>F</td>
<td>T</td>
<td>T</td>
<td>F</td>
<td>F</td>
<td>T</td>
</tr>
<tr>
<td>F</td>
<td>F</td>
<td>T</td>
<td>T</td>
<td>F</td>
<td>F</td>
</tr>
</tbody>
</table>
What if we want to adaptively assign Boolean values.

- We can use relational operators.
  - <, >, <=, >=, !=, ==
- These are all comparison operators that return True or False.
- == is the equality operator.
- != is not equals.
Boolean Expressions and Representation

- Can combine boolean operators (and, or, not) and relational operators (<, >, etc) and arithmetic operators (+, -, *, etc).
  
  - $5 + 7 < 4 \times 3$ or $1 - 2 > 2 - 4$ and $15 == 4$ is a legal expression.
  
  - Arithmetic goes before relational goes before boolean.

- False is represented as 0, and True is represented as 1.
  
  - Can lead to weirdness. Best to avoid exploiting this.
Short Circuit Evaluation

• Python only evaluates a boolean expression as long as the answer is not clear.
  • It will stop as soon as the answer is clear.
• This, combined with the nature of boolean representation can lead to strange behaviour.
• Exploiting these behaviours is bad style.
How to use boolean variables

• Recall that we want to make our code adaptive.

• To use boolean variables to selectively execute blocks of code, we use if statements.
If statement

• The general form of an if statement is:

  if condition:
      block

• Example:

  if grade >= 50:
      print “pass”
If statement

• The general form of an if statement is:
  
  ```python
  if condition:
      block
  ```

• The *condition* is a boolean expression.

• Recall that a block is a series of python statements.

• If the *condition* evaluates to true the block is executed.
Other Forms of if statement

• If we want to execute different lines of code based on the outcome of the boolean expression we can use:

    if condition:
        block
    else:
        block

• The block under the else is executed if the condition evaluates to false.
More general if statement.

```python
if condition1:
    block
elif condition2:
    block
elif condition3:
    block
else:
    block
```

- Python evaluates the conditions in order.
- It executes the block of the first (and only the first) condition that is true.
- The final else is optional.
Style advice for booleans.

- If you are unsure of precedence, use parentheses.
  - Will make it easier for a reader.
  - Also use parentheses for complicated expressions.
- Simplify your Boolean expressions.
  - Get rid of double negatives, etc.
Boolean Docstrings.

- def: is_odd(x):
  
  return (x%2)==1

- The docstring for this might look like
  """Return True if int x is odd, and False otherwise.""

- Commonly shortened to:
  - """Return True iff int x is odd.""
IFF

- iff stands for if and only if.
- So in fact we wrote:
  - ""Return True if int x is odd and only iff int x is odd.""
- We didn't specify what to do if x is not odd.
- But for boolean functions, it is understood that we are to return False if we're not returning True.