Hands-On Introduction to Python Programming

Jacek.Generowicz@cern.ch

This is a combination of slides and notes. The slides provide visual cues for the course. The notes provide further information and explanation. Both should be viewable in standards compliant web browsers. Printing should ignore the slides, and print the notes.

- Enable JavaScript in your browser.
- Use `t` to toggle between slides and notes.
- Arrow keys, `return`, `space`, `n`, `p`, `control n`, `control p` and mouse-click to move forward and back through the slides.
- page number followed by `return` to jump to that page.
- Mouse over bottom-right corner for mouse-based navigation.

Running Python

- We will be using Python 2.7

Other Development Methods

- Python provides its own interactive shell. (Try it now.)
  Type `python` in the UNIX shell.
- Whole programs can be run by:
  - Shebang notation (plus `chmod u+x program.py`)
    Unix-like operating systems understand that files starting with `#! /path/to /interpreter` should be intrepreted by the program specified after the `#!`. For this to work, the file must be labelled as executable, using `chmod`.
  - `python2.7 program.py`
    Invoke the Python interpreter in the UNIX shell, passing the file containing the program source as an argument.
- Any (plain ASCII!) text editor will do, BUT ...
- I URGE you to find a more interactive solution than running whole scripts in one go.
- Both commercial and free Python IDEs exist.
- Emacs, Python mode is VERY useful and highly interactive.
  Emacs' Python mode allows to to run Python from within Emacs. This enables you to execute your whole program, or portions of your program from within Emacs without having to retype them. This style of development offers many benefits and is a central point of this course.

This course aims to give you

- the ability to write useful programs in Python
- an appreciation of the advantages of a dynamic, introspective, incremental environment for rapid program development
- an appreciation of the 'batteries included' principle
- a look at programming techniques you might not have seen before
- to give you a deep understanding of fundamental concepts on which Python rests

Learn by doing

That which we must learn to do, we learn by doing.

Aristotle - *Nicomachean Ethics*

- You'll DO lots
- Try out almost anything that appares on the slides and looks like source code (it will be in `this font`), in your Python interpreter.
  - This is vital!
- Please, please, PLEASE **DO** ask me lots of questions.

Incremental Development

- No lengthy edit-compile-debug cycle.
  If you are used to languages such as Smalltalk or Lisp, this will be familiar to you. If your experience is limited to languages such as C++, Java and Fortan, you are will be used to a development style something like the following.
  1. Write the minimal boilerplate code necessary for the language to do anything at all. For example `#include <iostream> int main(int argc, char*[] argv) { return 0; }`
  2. Write a bit of code that actually **does** something
  3. Compile
  4. Wait
  5. Look at the compilation errors
  6. Swear at the semicolon you forgot
  7. Edit
  8. Save
  9. Compile
  10. Wait
  11. Look at the compilation errors
  12. Try to figure out what you did wrong
  13. Edit
  14. Save
  15. Go back to step 3

  The time spent waiting for the compilation to finish, breaks your concertation. This disruption is absent in Python development.
- Make tiny changes and test them immediately.
  Python's interactive nature allows you to test tiny fragments of code, instantly, giving you immediate feedback on your ideas. Less interactive languages strongly discourage seeking such feedback: boilerplate code such as that shown above, in combination with the edit compile cycle, make the cost of trying tiny fragments of code, much more expensive than it is in Python.
- Program state is not lost.
  If you run your code interactively, a runtime error will drop you into Python's interactive shell, where much of your data will still be available to you for inspection. It may even be possible to fix some buggy function, and resume the computation successfully.
- This can reduce development time by an order of magnitude, sometimes even more.

A different kind of program

The following, while written about a different language, applies almost equally to Python.

Imagine the kind of conversation you would have with someone so far away that there was a transmission delay of one minute. Now imagine speaking to someone in the next room. You wouldn't just have the same conversation faster, you would have a different kind of conversation. In Lisp, developing software is like speaking face-to-face. You can test code as you're writing it. And instant turnaround has just as dramatic an effect on development as it does on conversation. You don't just write the same program faster; you write a different kind of program.

Paul Graham - *On Lisp*, p. 38 http://www.paulgraham.com/onlisp.html

The whole book is available for free on the web at the above URL. If your experience is limited to languages such as Java, C++ or Fortran, then this book could be a serious eye-opener.

You **MUST** work interactively and try things out for yourself, otherwise you will benefit little from this course, and fail to grasp one of the big advantages Python has to offer.

Introspection

- A Python program can ask interesting questions about its own state, and the objects it is manipulating:
- `dir(), dir(1)`

  `dir` is a built-in function which returns a list of all the names belonging to some namespace. If no arguments are passed to `dir`, it inspects the namespace in which it was called. If `dir` is given an argument, then it inspects the namespace of the object which it was passed.
  You will find many names which both start and end with a double underscore (e.g. `__name__`). These are called *special* (colloquially also called *magic*) names. Please ignore these for the time being. They will be covered in due course, but for now just imagine that they are not there.

- `help(dir) # [pydoc]`

  Try `help()`!

  The UNIX command `pydoc` has access to essentially the same information as Python's built-in function `help`. `help`, however, does have access to information about objects in the currently running program, while `pydoc` can only access those objects which are available by default.

  `pydoc -p 8000` will start a web server on localhost:8000, through which you can browse the documentation through a web interface.

- `type(1)`
- `isinstance(...)`

  `isinstance` returns `True` if the given object is an instance of the given type, or any of its superclasses. Alternatively the typespec can be a tuple of types. To get the full details about how `isinstance` can be called, try using `help`.

Duck Typing

- If it looks like a duck ...
- ... and it quacks like a duck ...
- ... then it's a duck !
- What the object can do, is far more important that it's actual type.
- Checking types in Python is often the wrong thing to do.
- Try to see whether on object can do what you want it to, rather than checking its type.
- You can't possibly know ALL the types that are able to do what you want ...
- ... some of them might not have been written yet!
- Often, the simplest way to see whether an object can do what you want, is to try to get it to do it.
- I don't care who your parents are, as long as you can do the job.

Docstrings

- `help(...)` accesses the *documentation strings* of objects
- Any literal string apparing as the first item in the definition of a class, function, method or module, is taken to be its docstring.
  `help` includes the docstring in the information it displays about the object. In addition to the docstring it may display some other information, for example, in the case of functions, it displays the function's signature.

Multi-paradigm

- Imperative
- Object oriented
- Functional
- Mix-in
- (Aspect oriented)
- ...

While Python provides excellent support for object-oriented programming, this does not mean that you should write all Python programs in OO style. Pick the programming style that is most appropriate to the problem you are solving.

Numerical types

- `int` (C's long), `float` (C's double) - nothing exciting to say
  - The Python interpreter we are using is implemented in C. Python's `int` is based on the underlying C implementation's `int` type, while Python's `float` is based on the underlying C implementation's `double`. There is only one precision of floating point number in Python.
  - While the C-based implementation (sometimes known as *CPython*) is the "official" Python, other implementations do exist. An important one is Jython which is implemented Java, and allows seamless import and use of any Java class.
  - IronPython is an implementation on top of .NET, allowing use of all .NET libraries.
  - Another noteworthy Python implementation is PyPy, which implements Python in Python itself.
- `long`
  - Try a thought experiment: Imagine you have to calculate the factorial of 100. What problem will you face?
  - Reminder: the factorial of an integer `N` (written `N!`) is the product of all integers from 1 through `N`. So `4! = 1x2x3x4 = 24`
  - Factorials grow very quickly, and 100! is a **very** large number, so large that it cannot be represented by the integer type. You need some other data structure capable of representing bigger integers.
  - This capability is provided in Python by the `long` type. It as an integer type of **unlimited** precision. The precision is limited only by the memory available to your program.

  - ```
    import sys  # (importing modules will be explained shortly)
    a = sys.maxint
    b = a + 1
    print a, b, type(a), type(b)
    ```

- complex: `1 + 1j`
    ```

Namespaces, Callables ...

The `complex` type provides an excellent opportunity to learn some important lessons about Python.

- Use `dir` and `help` to find some *interesting* attributes of `complex` numbers
  Remember that I asked you to ignore magic names: those with leading and trailing double underscores. So "interesting" should be taken to mean, "not magic".
- Check that Python's `complex` numbers behave the way you would expect
  - Just in case you are unfamiliar with complex numbers: A complex number consists of two component numbers known as the *real* and *imaginary* parts. Complex numbers are typically written as `a + ib`, where `a` is real component, and `b` is the imaginary component. The symbol `i` denotes the *imaginary* number: the square root of -1. A common operation performed on complex numbers is that of *conjugation* in which the sign of the imaginary part is reversed.
  - Note that while the imaginary number is typically denoted by an `i` in mathematics and physics literature, the symbol `j` is preferred by electrical engineers. This is also the symbol used by Python.
  - Note that while `j` and `1 + j` would be perfectly acceptable ways of writing a complex number in mathematical notation, Python would interpret these two expressions as "the value of the variable `j`" and "one plus the value of the variable `j`" respectively. In order to make Python understand that you are referring to the imaginary number, rather than the variable `j`, you must directly precede `j` with a coefficient, even if it is the mathematically superfluous coefficient 1.

This little exercise is designed to teach some important lessons.

- **Namespaces** It is important to understand the namespaces in which various names are found. `dir(complex)` or `dir(1j)` yields `conjugate`, `imag` and `real` as the interesting attributes of complex numbers. A common mistake at this point is to try to find the conjugate of a complex number thus: `conjugate(1+1j)`. This produces a `NameError` because the name `conjugate` is not present in the global namespace, in which the command was issued. The `conjugate` name resides in the complex number class, and therefore must be accessed through the class itself, or through instances of the class. The most obvious correct way to calculate the conjugate is probably `(1+1j).conjugate()`. Another obvious way of achieving the same result would be `complex.conjugate(1+1j)`. Other more indirect methods exist.
- **Functions must be called, if they are to do something.** Often people try `1j.conjugate` and think that the output that Python gives them is an error. It is not. What was asked for was "the `conjugate` attribute of the object `1j`". It so happens that that attribute is a function (method). If you asked for it, you got it, and Python told you that that is what you have. (Python's interactive shell shows you the printed representation of the expressions you evaluate in the shell. If your expression evaluates to a function, the shell shows you the printed representation of a function, which in this case looks something like `<built-in method conjugate of complex object at 0x30d200>`.) In order to get the function (or any other *callable*) to do any work, you must call it by placing parentheses behind it. Those parentheses should contain the arguments to the function. Even if the function takes zero arguments, the parentheses are still required.
- **Functions are data.** In languages such as C++ there is a clear distinction between functions and data. At this point, students are often tempted to ask "how can I tell which are the functions, and which are the data?"
  - They are all data. In Python, functions, just like any other data, are first class objects.
  - Inspect the type of the attribute, by passing the attribute to `type`.
- **It is perfectly normal *not* to call functions in Python**, but to ask for the function itself, in order to use it (or make it be used) at a different point. For example you might store a function in the variable `f`, like this:`f = complex.conjugate` and later use that stored function like this: `f(1+1j)`. Such treatement of functions as data is central to functional programming. It is quite common for functions to take other functions as arugments. Functions which take other functions as arguments are called higher-order functions.
- (The astute student may have noticed that the implementation of complex numbers is not quite as simple as might be inferred from the above. `imag` and `real` are not implemented as pure data attributes, rather as descriptors, but these are far beyond the scope of this course, certainly at such an early stage. For the purposes of illustrating the points in the exercise, it is best to think of `imag` and `real` as pure data attributes.)

Modules

- Group together functionality
- Provide namespaces
- Means of extending Python
- Python's standard library contains a vast collection of modules - "Batteries Included"
  - Try `help('modules')`
  - `help` is a function which gives information about the object which is passed as its argument. Most things in Python (classes, functions, modules, etc.) are objects, and therefore can be passed to `help`. There are, however, some things on which you might like to ask for help, which are not existing Python objects. In such cases it is often possible to pass a string contaning the name of the thing or concept to `help`.
    - `help('modules')` will generate a list of all modules which can be imported into the current interpreter. Note that `help(modules)` (note absence of quotes) will result in a `NameError` (unless you are unlucky enough to have a variable called `modules` floating around, in which case you will get help on whatever that variable happens to refer to.)
    - `help('some_module')`, where `some_module` is a module which has not been imported yet (and therefore isn't an object yet), will give you that module's help information.
    - Keywords. For example `and`, `if` or `print`. These are special words recognized by Python: they are not objects and thus cannot be passed as arguments to `help`. Passing the name of the keyword as a string to `help` works, but only if you have Python's HTML documentation installed, and the interpreter has been made aware of its location by setting the environment variable `PYTHONDOCS`.

Importing modules

- `import math`
  This will introduce the name `math` into the namespace in which the `import` command was issued. The names within the `math` module will **not** appear in the enclosing namespace: they must be accessed through the name `math`. For example: `math.sin`.
- `import math, cmath`
  More than one module can be imported in the same statement.
- `import math as calc`
  The name by which the module is known locally can be different from its "official" name. Typical uses of this are
  - To avoid name clashes with existing names
  - To change the name to something more manageable. For example `import SimpleHTTPServer as shs`. I would discourage this for production code (as longer meaningful names make programs far more understandable than short cryptic ones), but for interactively testing out ideas, being able to use a short synonym can make your life much easier. Given that (imported) modules are first class objects, you can, of course, simply do `shs = SimpleHTTPServer` in order to obtain the more easily typable handle on the module.
- `from math import sin`
  This will import the module `math`, but it will not introduce the name `math` into the current namespace. It will only introduce the name `sin` into the current namespace. It is possible to pull in more than one name from the module in one go: `from math import sin, cos`.
- `from math import *`
  Once again, this does not introduce the name `math` into the current namespace. It does however introduce all public names of the math module into the current namespace. Broadly speaking, it is a bad idea to do this:
  - Lots of new names will be dumped into the current namespace.
  - Are you sure they will not clobber any names already present?
  - It will be very difficult to trace where these names came from
  - Having said that, some modules (including ones in the standard library, recommend that they be imported in this way).

  Use with caution!

Don't forget to try `dir(...)` and `help(...)`

- The above list is not exhaustive. For example `from math import sin as cos`.
- I haven't touched on packages.
- The `import` statement underwent enhancements in Python 2.4 and 2.5.
- Python tries to avoid importing the same module twice: It maintains a list of all imported modules which `import` checks before importing a module. Therefore, if you are developing some code interactively, and you make changes to some module you are writing and `import` the module again from some code you are running interactively, the changes in the module will not become available to you, as Python would have recognized that this module is already loaded and will not have bothered to do it again. In such situations, you should use the `reload` function (or remove the module's name from `sys.modules`).

Sequences

- List - mutable, heterogeneous
- Tuple - immutable, heterogeneous
- String - immutable, homogeneous

Lists

- `[]` - the empty list
- `[6]` - a 1-element list
- `[5, 1+2j, 'hello']` - a 3-element list
- `[[1,2],[3,4],[5,6]]` - a list of lists
- WRT other languages, resembles vector or array more than list
  In computer science, the name "list" typically refers to *linked lists*: chains of objects where each link in the chain contains a datum and a reference to the next link. Python's lists, however, are objects in which the elements occupy contiguous locations in memory: the next object in the list is not found by following a reference, but by looking at the next location in memory. This type of data structure is usually referred to as an *array* or *vector*. It is important to be aware of this difference, as the algorithmic complexity of various operations on the object changes significantly depending on the implementation used.

	Linked lists	**Vectors (Python lists)**
Random access	O(n)	O(1)
Extension	O(1)	O(n)

  Sometimes vector implementations buy a reduction in the *average* algorithmic time complexity of extension by sacrificing some memory. Python's lists (which are really vectors) implement such an optimization and therefore extending Python lists, *on average*, is independent of the length of the list, or O(1). Such accounting over an increased amount of time is called *amortization*, therefore Python's list extensions are sometimes described as taking *amortized constant time*.

Tuples

- `()` - the empty tuple
- `(1,)` - a 1-element tuple
  - Note the comma: without it, this would evaluate to the integer 1 !
    Compare `(1)` and `(1,)`. Both Python lists and tuples may be written with a supefluous comma following the last element. It is only in the case of the single-element tuple that such a trailing comma is necessary, in order to ensure that Python understands the expression to be a tuple, rather than a parenthesized value.
- `(1, 'hello', [1,2])` - a 3-element tuple
- The parentheses are not always required: `a = 1,2,3`
- Tuples are **IMMUTABLE**
  This means that once a tuple is created, it cannot be modified. Its value cannot change: it will continue to have exactly the same value (contents, structure) until the moment it is destroyed. Try `t = (1,2,3); t[0] = 9`. (The alert student may have found a way of mutating a tuple. Shhhh ... don't tell the others for now.) Compare the *interesting* attributes of lists to those of tuples. What do you notice? Why?

What does += do?

```
l1 = [1, 2, 3]
l2 = l1
l2 == l1, l2 is l1 # Both names refer to the same object.
l2 += [4, 5]
l1, l2
l1 == l2, l2 is l1 # Mutations of the object are visible through both names.

t1 = (1, 2, 3)
t2 = t1
t2 == t1, t2 is t1 # Again, both names refer to the same object.
t2 += (4, 5) # But tuples are immutable. The original object cannot be modified, so a new object is created and bound to t2. The original remains unaltered.
t1, t2
t1 == t2, t2 is t1 # t1 continues to refer to the original object.

i1 = 0
i2 = i1
i2 == i1, i2 is i1 # Again, only one object ...
i2 += 1 # ... which is immutable, so a new one is created and bound to i2, .
i1, i2
i1 == i2, i2 is i1 # i1 continues to refer to the original object.
```

Strings

- `'A string'`
- `"Another string"`
- `"A string with a ' in the middle"`
- `'A string with a \' in the middle'`
- `"""A multi-line string"""`
- Strings are **IMMUTABLE**

Unpacking tuples

- `a,b,c = 1,2,3`
- `a,b = b,a`
  Note that the assignments occur *logically* in parallel. Without parellel assignment, you would need a third temporary variable in order to swap the values of two variables. Parallel assignment allows you to do it in a single line with no extra variables, as above.
- `w = 5,6,7`
- `x,y,z = w`
- Return multiple values from functions using tuples: output parameters are un-pythonic
  You may have experience in lanugages in which it is common practice to make use of *output parameters*, in cases where a function should be returning more than one value. This practice involves passing pointers or references into a function, and looking for the result at the end of such pointers, once the function has finished running.
  - This requires the caller to prepare the locations where the results will appear beforehand (even for those outputs which he wishes to ignore)
  - It is often not obvious which arguments are inputs, and which are outputs of the function.

  It is possible to implement something like output parameters in Python, but you should avoid doing so. Pass inputs in as arguments; return all outputs in an (implicit) tuple. For example: `return width, height, length`

Indexing and slicing

- `a = range(10)`
  I trust that you tried `help(range)` without prompting. Get used to using `help` and `dir`.
- `a[3]`
  Indexing is zero-based.
- `a[-3]`
  Negative indices are an offset from the **end** of the sequence.
- `a[3:6]`
  This is known as a slice. Note that the lower limit is inclusive, while the upper limit it exclusive.
- `a[3:-3]`
- `a[6:3]`
  Yes, it's empty: you reached the end before you started!
- `a[:6]`
  Leaving out a limit means "go as far as you can in that direction".
- `a[0:6:2]`
  A third number gives the step size. It can be negative.
  The step-size argument was implemented for the built-in sequences only in Python 2.3, even though the third argument was available for use in user-defined types much earlier.
- `a[2:4] = ['x']`
- `a[-1:] = 'abc'`

Slicing works for all bult-in sequences and can be implemented in user-defined types too.

Sequence operations

- `a = range(4)`
- `a + [9,8,7]`
- `a * 4`
- `b = [[0]*4]*4` Common pitfall
  The inner list is a list of four elements. Each element is (a reference to) the number `0`. The outer list is also a list of four elements. Each of those elements is (a reference to) the same single inner list ...
- `b`
- `b[0][0] = 1`
  ... This assignement rebinds the first element of the first inner list. This element therefore no longer refers to the number `0`, but to the number `1`. However, because the outer list contains **four** references to the inner one, this change is reflected in all four elements of the outer list.
- `b`
- `3 in a; 3 not in a`
- `max([1,'x',2,'a'])` ?
- `a += [9]`
- `a.append(7)`

Variable, binding, call-by-value

- Variables do not have types; objects have type - dynamic typing
  In *statically typed* languages, types are associated with variables. The association of some data with a variable determines the type of the data. In dynamically typed languages, the objects themselves carry arround their type information. Variables are associated with types only incidentally.
  Consider the following example in C++:

```
int i;
i = 3;
i = std::vector<float>();
```

  - First you inform the compiler that the variable `i` is to hold an integer.
  - You assign an integer to `i`. That's fine.
  - Then you attempt to stick something of a completely different type into `i`: the compiler complains.

  Contrast this to something similar in Python:

```
i = 3
i = "hello"
```

  - There is no need to declare any variable.
  - First you bind the name i to the value 3. The name `i` neither knows nor cares about the type of the object to which it refers; the object itself knows that it is of type `int`.
  - Then you rebind `i` to an object with a different type. The name still doesn't care about the type of object it refers to; the new object knows that its type is `str`.
- *binding* is the association of a variable (more generally, a location) to an object
- Python uses call-by-value semantics ...
- ... but all values are references
  Consequently, it is easy to be fooled into believing that Python uses call-by-reference semantics. For example:

```
def foo(a):
    a.append(6)

x = range(3)
foo(x)
print x   # =>  [0, 1, 2, 6]
```

  However, if it *really* were call-by-reference semantics then the following code

```
def bar(a):
    a = [5]

x = range(3)
bar(x)
print x   # =>  [0, 1, 2]
```

  would yield `[5]`.
- Parameter binding uses exactly the same mechanism as variable assignment.
  If you understand one, then you immediately understand the other.

Indentation

- Python uses indentation to delimit blocks
- ```
  if True:
      print 'this'
      print 'that'
  else:
      print 'the other'
  ```

  Now that we are starting to write snippets of code which are more than one line long, it is time to find a more efficient way of working interactively with Python. The tool I will be using to demonstrate this is Emacs, as it enables good interaction with Python, is widely available and probably familiar to many of you. I apologise to anyone who has religious objections or allergy to Emacs.
  ```

- Fire up Emacs
- Open a file whose name ends in `.py` (either by clicking on "Open File" in the "File" menu, or by issuing the keystrokes `C-x C-f`)
  - In Emacs convention `C-x` means "control-x" or "hold down the `control` key while pressing the `x` key". Similarly `M-x` means "meta-x". The `meta` key is usually labelled `Alt` on your keyboard. An alternative way of issuing `M-x` is to press `Escape` and release it before pressing `x`.
- At this point you will be prompted for the filename in the minibuffer. The minibuffer is the line at the bottom of the Emacs frame. Type something like `foo.py` (anything ending in `.py` will do) and press `RETURN`.
- At this point you should see the word "Python" appear in the modeline, confirming that Emacs has guessed (from the `.py` filename extension) that you are intending to work on a Python program. (The modeline is the line at the bottom of every Emacs window, which gives some information about what is going on in that window.) If "Python" does not appear, then your Emacs does not have Python mode installed, or it is incorrectly configured.
- When the active buffer is in Python mode, you should notice that two menus containing the word "Python" appear in the menu bar.
- Click on the menu named "Python" and select the item "Start interpreter".
  - There is on alternative python mode, which has somewhat different behaviour, and completely different keybindings. If you have this alternative mode (e.g. if you are running GNU Emacs 22 or newer, or something built on top of it, such as AquaMacs), then the keybindings mentioned below will not match those available to you. Just look at the keybindings shown in the (single) Python menu.
  - One problem with this other python mode is that it creates an `emacs` namespace into which it places names that it creates, thus making interactive programming more difficult. I have written a simple workaround for this, which you can find here. Simply add this code to your Emacs startup file, or evaluate it interactively.
- At this point, your Emacs frame should split into two windows: one containing the buffer you were previously viewing; the other containing a running interactive Python shell.
- Note that the "Start interpreter" item in the "Python" menu also shows the characters `C-c !`. This is the keyboard command for starting the interpreter. This means that you could have started the interpreter by pressing `Control-C` followed by `!`.
- You should also find "Evaluate buffer" in the menu, bound to `C-c C-c`. This will execute the entire contents of the active buffer. Note that if you have not saved the buffer, the contents may differ from the file's contents as stored on disk.
- **A very useful feature is "Evaluate region", bound to `C-c |`. This allows you to highlight some portion of the buffer, and execute just that portion. You should make plenty of use of this feature for testing out small framents of code, or updating functions you have debugged and fixed, etc.**
- You can navigate the history of commands you issued in the Python interpreter running within Emacs with `M-p`, `M-n` (`p` for previous, `n` for next).
- Alternatively use the cursor keys to move to some previous input line (showing the prompt `>>>`) and press `RETURN`. That input will then be copied to the current input line. You now have an opportunity to edit it. Pressing `RETURN` again, will execute the input.

- Note that blocks start with colons
- `pass` - the empty block
  The situation may arise, in which you need to write an empty block: a block containing no code. In such situations you should use the keyword `pass` which is a statement that does nothing.

Indentation

- Indentation is the **only** block delimitation mechanism

  There are no `begin`s/`end`s, no `{ ... }` or anything else: just indentation. Some people, when initially confronted with this fact have quite a negative reaction. With time, however, most of those who try working in Python, come to appreciate this feature. Some of the advantages it offers are:

  - Gets rid of the visual clutter caused by the presence of block delimiters, making the code easier to read.
  - Avoids wasting lines on block delimitation tokens, thus making the code shorter, which makes more code fit on the screen, which makes it easier to program.
  - Ensures that the block structure perceived by the compiler is the same as that perceived by the human. Here are some examples (in C) where, more often than not, the human reader will parse the code differently form the compiler:

    ```
    if (...)
    {
        do_this();
    do_that();
    }

    if (...)
        do_this();
        do_that();

    if (...)
        if (...)
            do_something();
    else
        surprise();
    ```

    In Python, such mistakes don't occur.

- Indentation is ignored inside brackets

  ```
  a = [1,
  2,
    3       ,
       4
    , 5]
  ```

  and on continued lines

  ```
  b = 1 + \
  2 +\
              3 +\
      4
  ```

- You can use tabs or spaces: just don't mix the two as that is likely to make things go wrong.

  The Python command line interpreter has two options for invoking the detection of mixed tabs and spaces: `-t` and `-tt`. The former issues warnings about mixed tabs and spaces, the latter promotes such warnings to errors.

- You can use whatever number of tabs or spaces you like, as long as the number is the same within a single block.
- Use your editor's support for indenting Python.

  In Emacs you can use the `TAB` and `Backspace` keys to jump between various levels of indentation that Emacs thinks are possible at this point. Emacs inserts spaces for indentation. By default, it inserts four spaces, though you can configure it to whatever you like.

Boolean contexts

- False
  - Numbers equal to zero
    `0, 0.0, 0j, 0l`
  - Empty sequences and dictionaries
    `[], (), '', {}`
  - `None`
    an object denoting the absence of an object
  - `False`
    `True` and `False` appeared in Python 2.1.1, where they were names bound to the integers 1 and 0, respectively. In Python 2.2 a separate `bool` type was introduced.
- True
  - Just about everything else
  - `True`
- User defined types can determine their own interpretation in boolean contexts.

Loops

- ```
  epsilon = 1.0
  while 1 + epsilon > 1:
      epsilon /= 2
      print epsilon
  ```

- ```
  for c in 'hello':
      print c
  ```

  For loops in many other languages update some number on each iteration, which is then, typically, used only as an index into a sequence: the number itself is often not interesting, it is merely used to implement sequence iteration. In Python, this iteration over sequences is emphasized in for loops. There are, however, situations in which you are genuinely interested in the for loop generating such numbers. The built-in functions `range` and `xrange` can be used for this purpose. A related utility, `enumerate`, was introduced in Python 2.3.

- for-loops work with any iterable
- `help(enumerate)`

- ```
  for index, item in enumerate('my data'):
      print item, 'was in position', index
  ```
  ```

Loops

- `break`

  immediately terminates the enclosing loop

- `continue`

  skips the rest of the current iteration and proceeds immediately with the next iteration

- `else`

  Both `while` and `for` loops have an optional `else` clause, the body of which is executed only if the loop runs to completion, without being cut short by:

  - `break`,
  - `return` - returning from the containing function within the body of the loop,
  - `raise` - an exception being raised in the body of the loop but not being handled.

  Exception-handling `try` blocks have a similar optional `else` block.

Laziness (is a virtue)

- ```
  for x in range(1000000):
      if x > 2:
          break
      do_stuff_with(x)
  ```

- Use `xrange` instead of `range`
  Imagine you want to generate the numbers for 0 to 1000000. Using `range` would result in the creation of a list of length 1000000, which would eat up significant amounts of memory. `xrange` has similar behaviour to `range`: rather than creating a list, it returns an object capable of producing the desired numbers as and when they are required. Consequently it is a little faster for looping, and considerably more memory efficent.
- Laziness allows you to avoid doing work that never needs to be done
- Lazines can help you to deal more efficiently with large amounts of data.
- Laziness is becoming increasingly more important in Python as the language evolves.
  ```

Functions

```
def my_function(arg1, arg2):

    """Optional docstring."""

    # Your implementation goes here

    return a, b, c  # optional


this_is_not_part_of_the_function()
```

... because the indentation has returned to the level at which the function definition was started. Therefore the block (containing the function body) has been closed.

If a function does not `return` anything explicitly, then it will, implicitly, return `None`.

- `help(zip)`
- Write your own (non-lazy) implementation of `enumerate`
- Later we will write one just as lazy as the real `enumerate`

Multiple return values

```
def powers(n):
    return n, n*n, n*n*n
```

The function returns three values all at once. The values are packed into a tuple.

```
ps = powers(2)
```

You can collect all the returned values at once, in a tuple ...

```
a,b,c = powers(2)
```

... or unpack the tuple at the call site, and receive each value in its own variable.

First class objects

- Almost everything (including functions, classes, modules...) in Python is a first-class object
  This means that they can be stored in variables and data structures, passed to functions, returned from function, just like any other value.

- ```
  import math
  trig = math.sin, math.cos, math.tan
  for fn in trig:
      print fn, fn(math.pi/3)
  ```

- Note the important difference between a function and the result of calling that function.
  - In order to call a function, you must place parentheses just after it. Any arguments to the function are placed between the parentheses.
  - In Python it is quite normal to want to refer to the function itself without wanting to call it (to store it somewhere, or to pass it to another function etc.).
  - Make sure to add the parentheses where necessary, and to leave them out when you don't want the function to be called.

Lexical Closures

```
def make_adder(n):

    def adder(x):
        return n+x

    return adder
```

- Two functions are being defined, one inside the other.
- The outer function returns the inner function.
- The inner function refers to a name (`n`) which is bound in the scope of the outer function.
  - When the outer function returns, `n`, being a local variable, goes out of scope, and it would normally disappear along with its binding.
  - The inner function prolongs the lifetime of this binding, so that it can use `n` when it itself is called.
  - Each time the outer function is called, a different version of the inner function is returned, each with its own, distinct binding of `n`.
  - The inner function the binding of `n`. It is a *closure* over `n`.

```
add3 = make_adder(3)
add9 = make_adder(9)
print add3(4), add9(4)
```

Each different instance of `adder` carries around its own version of `n`.

Aside: notice that a closure is a combination of functionality and state ... which is the basis of object oriented programming.
```

Unqualified name search

- L^{n}GB: Local (nested), Global, Builtins

  An unqualified name is looked up in the following order. The first matching binding found is the one used.

  - **L**: Inside a function body, look for a binding in the function's **L**ocal scope. (Outside functions, go straight to step `B`)
  - **n**: Progress outwards through any surrounding **n**ested functions, looking in the local scope of those functions.
  - **G**: Look for a binding in the **G**local scope.
  - **B**: Look for a binding in the **B**uiltins.
  - If no binding was found, raise a `NameError`.

  If a name is bound inside a function (either by virtue of being a parameter of the function, or by having an assignment made to it), the binding is local to that function. Sometimes you may wish to assign to a global name from within a function. The the implication that bindings inside functions are local, prevents you from doing so. In such situations, you may declare this intent using the `global` keyword:

```
a, b = 0, 0

def foo():
    global a
    a,b = 3, 3

foo()
print a, b  # => 3 0
```

  In the above, `a` is bound to `3` in the global namespace, rather than the local namespace. **b** is bound to `3` in the function's local namespace, *shadowing* the global binding, so the global binding of `b` is unaffected by the assignment.

- Prior to Python 2.1, enclosing function scopes were not searched, so the lookup rule was simply LGB.

Argument passing

Please make sure you understand a given function before trying out the next.

```
def one(*args):
    return args
```

All (positional) arguments are collected into a tuple. That tuple will be bound, within the body of the function, to the name specified after the `*`.

```
def two(a=1, b=2):
    return a,b
```

These are called *keyword parameters* or *default arguments*. They provide default values, in case no corresponding argument is passed in. They also allow you to pass in arguments out of order, by naming them. Try `two(c=9)`.

```
def three(*args, **kwds):
    return args, kwds
```

Just like a single `*` provides a means of collecting all remaining positional arguments into a tuple, all remaining keyword arguments may be collected into a dictionary: the dictionary will be bound to the name specified after `**`. (Dictionaries will be introduced soon.)

```
def four(a,b=1, *args, **kwds):
    return a, b, args, kwds
```

A combination of all sorts of things to test your understanding. Note that it is impossible to pass positional arguments after a keyword argument.

```
one(*'hello')
one(*range(3))
two(**{'b':'banana', 'a':'apple'})
three(*(range(3)+range(3)),**dict(a=1, b=2))
```

In function parameter lists, the stars collect positional and keyword arguments into tuples and dictionaries respectively. In function argument lists the starts work in reverse: single star unpacks any iterable into separate positional arguments; double start unpacks any mapping type into separate keyword arguments.

Garbage collection

- Allocating and freeing memory is NOT YOUR PROBLEM.
  - Yippee !!!
- Python uses reference counting.

  Reference counting relies on maintaining a count of the number of ways in which a given object may be accessed: the number of references to the object. When the number of references is zero, we know that the object can never be accessed again, and it may be forgotten, or deallocated.

- There is also a garbage collector for collecting circular garbage: module `gc`.

  Consider two objects referring to eachother. If all external references to these objects disappear, they will become inaccessible and should therefore be deallocated. However, their reference count will remain greater than zero, because they are referring to eachother. Such objects are known as *circular garbage* and constitute a memory leak in reference counting schemes.

Type systems

- Python is NOT weakly typed
- There are (at least) 3 orthogonal axes spanning type-system space
  - Static vs Dynamic
  - Weak vs Strong
    - nobody really knows what this means!
  - Explicit vs Implicit
    - Declaration vs Inference
- Many people confuse the first two
- Python is *strongly dynamically* typed

Typesystem space

Dictionaries

Dictionaries are hash-tables: fast maps

- `type({})`

  The type is called `dict`.

- `d = {}`

  An empty dictionary. You can also create one by calling the `dict` type thus: `dict()`.

- `d[1] = 'eins'; d['zwei'] = 2`

  This is how you add key-value pairs into the dictionary. `1` is the key corresponding to the value `'eins'`; `'zwei'` is the key corresponding to the value `2`. Any *immutable* type can act as a key.

- `d[1]`

  This is how you look up values corresponding to some key. Here we ask for the value associated with the key `1`.

- `d['eins']`

  Lookup can only be done on keys. `'eins'` is present in the dictionary only as a value, not as a key, therefore we get a `KeyError`.

- `d['zwei']`

  This works, because `'zwei'` is a key.

- `len(d)`

  This gives the number of key-value pairs in the dictionary.

- `1 in d, 'eins' in d, 1 not in d`

  The `in` operator tests whether the object is equal to one of the keys of the dictionary. It does not check the dictionary values.

- `squares = {2:4, 3:9, 4:16, 5:25}`

  You can create populated dictionaries inline, by typing in their printed representation. Alternatively, if the keys are strings, you can instantiate the `dict` type with keyword arguments: `dict(a=1, b=2, c=3)`.

  Keys must be immutable

  Dictionaries are un-ordered. Yes, they are presented visually in some order, and you can iterate over their contents in some order, but that order is meaningless, and will change as the dictionary changes size and is re-hashed.

  Don't forget `dir(...)`

Dictionaries

- Python uses dictionaries heavily in its own implementation, so dictionaries are very highly optimized.
  Just about every name lookup in Python is a dictionary lookup.
- Python provides no switch construct. A common idiom is to use dictionaries:

```
dispatch = { 'a': do_this,
             'b': do_that,
             'c': the_other }

reply = raw_input()
dispatch[reply]()
```

`do_this`, `do_that` and `the_other` must already be defined at the time you define the dictionary, must be callable, and must accept zero arguments.

I trust that I don't have to tell you to try `help(raw_input)`.

In the above we have created a dictionary which maps choices to actions. When the choice has been obtained (in this case by using `raw_input`) it is used as a key in a dictionary lookup. The dictionary returns the function corresponding to the choice, and that function is immediately called.

File I/O

-
```
file = open('myfile', 'w')
print >> file, 1, 2, 3, 4
file.write('5 6 7 8')
file = open('myfile', 'r')
for line in file:
    print line
```

- Oops - we have just shadowed the built-in type `file`
- The original is still available in `__builtins__`

You can use `del` to remove names from a namespace. In this case, we can undo our mistake by issuing `del file`. This will remove `file` from the global namespace, causing `file` to be found in the builtin namespace once more.

```
file is __builtins__.file    # False: the builtin is shadowed
del file                     # Remove the shadowing binding
file is __builtins__.file    # True
```

Sorting

- Make a list of all usernames in `/etc/passwd`, sorted by userid
  - That's the number, not the name
- Make sure you sort the numbers as integers, not as strings!
- Constructors of the built-in types act as type converters

  For example `tuple([1,2,3])` or `list('really?')` or `str(123)`.

  - Avoid calling this casting

    Casting has subtly different meanings in different languages. In some cases it is used to describe the process of reinterpreting the same bit-representation of the data as representing some other type. This in NOT what is happening in Python. If you avoid the word 'cast' then you avoid the temptation to believe that such a meaning is implied. What is really happening is that you are instantiating a new type using an instance of some other type as a source of data. Call it type conversion.

Frequently people choose to solve this using a dictionary. Given that sorting is involved, the collection must be ordered: dictionaries are unordered (remember, any order that they happen to have is arbitrary and may change). Furthermore, it is possible (though rare) for the password file to contain multiple usernames with the same userid. Such a dictionary based solution will throw away all but the last of a set of usernames that share an id. The temptation to use a dictionary arises from the need to maintain an association between the userid and the username, which is indeed the forte of dictionaries. In this case, however, we can use a far more lightweight means of maintaining an association between two objects.

I recommend using tuples to store the two interesting data from each record as a pair, and collecting these pairs in a list:

```
pairs = []
for record in passwordfile:
    name, id = ...
    pairs.append((name, id))
```

Read `help([].sort)`. Note that the sort is performed **in place**. While it is not explicitly stated, you could infer that `list.sort` returns `None`. Therefore you must be very careful ... many people are tempted to write

```
stuff = stuff.sort()
```

which is just a very inefficient equivalent of `stuff = None`. The correct way to proceed is this:

```
stuff = ...   # collect your data
stuff.sort()  # sort the data
print stuff   # inspect the sorted data
```

From Python 2.4 onwards, it is also possible to do

```
stuff = sorted(stuff)
```

The question remains, how to ensure that the sort is performed on the id? There are a number of approaches possible.

The first approach, which will work even in ancient versions of Python, relies on knowing how Python compares sequences. Sequences are compared by initially comparing the first elements only. If they

differ, then a decision is reached on the basis of those elements only. If the elements are equal, only then are the next elements in the sequence compared ... and so on, until a difference is found, or we run out of elements. Therefore, we can arrange to have our pairs sorted by id, by placing the ids first in the tuple:

```
pair = id, name
```

The second approach (which also works in ancient Pythons), relies on being able to decypher the cryptic `help(list.sort)`. You should notice that `list.sort` has three keyword parameters. The first of these is called `cmp`. If this argument is `None` (which it will be, by default), `list.sort` will use Python's built-in `cmp` function to compare the elements of the list it is sorting. `cmp` is a function of two arguments which returns a positive number when it considers the first argument to be 'greater' then the second, returns `0` when it considers the arguments to be equal, and returns a negative number otherwise. You can instruct `list.sort` to use a different comparison criterion, by supplying some other function with the same interface. Let's illustrate this in the context of our exercise, by assuming that we have stored a list of pairs like this

```
pair = name, id
```

We want to sort according to `id` and ignore `name`. We can achieve this by writing a function that compares the second elements of the pairs it receives, and returns a number matching the convention used by `cmp` as described above.

```
def my_cmp(a, b):
    if a[1] > b[1]:
        return +1
    if a[1] == b[1]:
        return 0
    return -1
```

Note that you must remember the convenion, and that there is a lot of fiddling around with indices. A much more concise and less error-prone (and more efficient) way of writing this function is

```
def my_cmp(a, b):
    return cmp(a[1], b[1])
```

Now you can use this function to tell `list.sort` how to compare the pairs:

```
pairs.sort(cmp=my_cmp)
```

or

```
pairs.sort(my_cmp)
```

or even

```
pairs.sort(lambda a,b: cmp(a[1], b[1]))  # lambda explained on next slide
```

Another solution, which became available in Python 2.4, relies on `sort`'s keyword parameter `key`. Here you can pass a function which `list.sort` will use to generate values according to which the list's elements should be sorted. In our case

```
pairs.sort(key=lambda p:p[1])  # lambda explained on next slide
```

will do the trick.

A note about efficiency. Sorting typically involves making many comparisons between the elements.

Python's built-in `cmp` is an efficient low-level function. By making `list.sort` use a pure-Python function instead, you are getting hit by the pure-Python function overhead (which is relatively large compared to the C function overhead) on each and every comparison. This might slow you down noticeably. The `key` function will be called exactly once, for every element in the list. This approach is likely to be measurably faster for large lists. If efficiency is really important (and you have proven that a significant proportion of time is spent in these functions) then you have the option of re-coding them in C (or another low-level language).

lambda, Higher-order functions

- anonymous function (function literal)
- `lambda a,b,c: a*b+c`
- `(lambda a,b,c: a*b+c)(2,3,4)`
- `lambda a,b,c: a*b+c(2,3,4)`
- `map(lambda a:a+1, [1,2,3,4])`
- `filter(lambda x:x%2, range(20))`
- `reduce(lambda a,b: a+b, range(10))`

  I trust that you didn't need prompting to do `help(map)` and so on.

  - `operator.add`
    The `opeartor` module contains functions which are typically accessed not by name, but via some symbols or special syntax. In this case, `operator.add` is the name used to access the function which is more commonly accessed via the infix operator `+`.

Functions like `map`, `reduce` and `filter` are found in just about any lanugage supporting functional programming. They all provide functional abstraction for commonly written loops. I recommend using these functions in preference to writing your own equivalent loops because

- Writing loops by hand is quite tedious and error-prone.
- The function version is often clearer to read.
- The functions result in faster code (unless it makes you introduce extra pure-Python function-call overhead to an otherwise quick operation ... as in the case of `lambda a,b: a+b`, which adds the relatively high pure-Python function call overhead to the very fast addition operation. In this case, we can used `operator.add` to avoid the problem.)

Summing exercise

- Given a file with an arbitrary number of numbers on each line ...
- ... make a list of the sums of the numbers on each line.
  For example, if your file contains:

  ```
  1 2 3
  4 5 6
  7 8 9 10
  ```

  Your program should give the results `6`, `15` and `24`.
- (Later ... allow non-numbers to appear in your file, and ignore them in the sum.)

Division

- `3/2, 3//2, 3.0/2.0, 3.0//2.0`
- `from __future__ import division`
- `3/2, 3//2, 3.0/2.0, 3.0//2.0`

  Note that the meaning of `3/2` will change in Python 3! In Python 2 integer divison is truncates and preserves the integer type; in Python 3 its result is always a float and no truncation occurs.

- `4/2`
- Available in Python 2.2
- Default in Python 3.0

Exceptions

```
try:
   # code body
except ArithmeticError:
   # what to do if arithmetic error
except IndexError as the_exception:
   # the_exception refers to the exeption in this block
except:
   # what to do for ANY other exception
else:  # optional
   # what to do if no exception raised

try:
   # code body
finally:
   # what to do ALWAYS
```

Starting with Python 2.5, you can use the `with` statement to simplify the writing of code which would previously use the `finally` block.

Before Python 2.5 `except` and `finally` were mutually exclusive within a single `try`.

Exceptions

- Derive your own exceptions from the built-in `Exception`.
- Raise exceptions with the `raise` keyword.
- The constructor of `Exception` accepts arguments, which are displayed in the printed representation of the exception
- The optional second argument of `raise` is passed to the constructor.

Raising Exceptions

- `raise OverflowError`
- `raise OverflowError, "Bath is full"`
  This form will be removed in Python 3.0
- `raise OverflowError("Bath is full")`
- `e = OverflowError("Bath is full")`
  `raise e`

- ```
  except Exception as e:
      ...
      raise e
  ```

- ```
  except Exception:
      ...
      raise
  ```

- Python 2:

  ```
  except Exception, e:
  ```

Exception hierarchy

- The standard exceptions are organized in an inheritance hierarchy
  - e.g. `OverflowError` is a subclass of `ArithmeticError`. (NOT `BathroomError` !)
- Allows you to catch a range of exceptions with a single statement.
- You can derive your own exceptions from any of the standard ones
- It is good style to have each module define its own base exception.
- Exceptions became new-style classes in Python 2.5

LBYL vs EAFP

- Look Before You Leap

- ```
  if denominator == 0:
      print "Oops"
  else:
      print numerator/denominator
  ```

- Easier to Ask for Forgiveness than Permission

- ```
  try:
      print numerator/denominator
  except ZeroDivisionError:
      print "Oops"
  ```

- EAFP is the Pythonic way
- Time to re-visit the sum-lines-in-file exercise.

A solution to the sum exercise

```
import operator

def float_or_zero(a_string):
    try:
        return float(a_string)
    except ValueError:
        return 0

for line in open('myfile', 'r'):
    print reduce(operator.add,
                 map(float_or_zero, line.split()))
```

Note: as of Python 2.3, there is a builtin `sum` which could be used here instead of `reduce(add, ...)`. You should be aware that `sum` refuses to work on strings. The reason for this is that adding strings together by summing them is far less efficient than using the idiom `''.join(sequence_of_strings)`, and `sum's` refusal to do the addition for you, is supposed to be taken as a hint to use that idiom instead.

IPython

- An enhanced Python shell
- http://ipython.scipy.org
- Persistent command history
- Name completion
- Shortcuts
- Easy system shell access
- Much more
- Try it: `ipython`
- NOT part of the standard distribution

Queue abstract type

An abstract type is one defined by the operations that can be performed on it.

Queue (FIFO)

FIFO: First In, First Out. Contrast this with a *stack* which is LIFO: Last In, First Out.

- Make a new queue
- Add an element to the back of the queue
- Remove an element from the front of the queue

A Python implementation of the queue abstract type

```
def make_queue():
    return []

def add_to_queue(queue, item):
    queue.append(item)

def remove_from_front_of_queue(queue):
    return queue.pop(0)
```

Classes

```
class Counter:

    def __init__(self, start):
        print(dir(self))
        self.count = start
        print(dir(self))

    def up(self, n=1):
        self.count += n

    def down(self, n=1):
        self.count -= n
```

We have written a *class*, a new type. The class is called `Counter`. It has two ordinary methods: `up` and `down`. It also has a *magic method* called `__init__`. `__init__` is also known as the *constructor*.

Now is the time to stop ignoring the magic names. Magic names are ones which, typically, are invoked by Python, in well-defined situations, on your behalf. You can refer to the names directly, if you so wish, but, typically, you don't. Some examples we have already met in passing:

- `__builtins__` - Python looks in this namespace when a name is looked up and is not found in a function's local scope, or in the global scope. We referred to `__builtins__` directly ourselves, when we investigated the consequences of having shadowed `file`.
- `__len__` - When we used `len` to find the size of lists, dictionaries, etc., Python called this magic method of the appropriate class.

```
a = Counter(10)
```

When calling a class, two magic functions are invoked on your behalf. First, the magic function `__new__` is called: this creates a new, unadulterated instance of the class. This new instance is then passed as the first argument to `__init__`: any arguments which were explicitly given in the call to the class, are passed in the following positions. In this case, the new instance is bound to `__init__`'s local variable `self`. The explicit argument `10` is bound to `__init__`'s local variable `start`. Within the body of the constructor (`__init__`), `self` (the new instance of the class) is given a new attribute called `count`. `self.start` is bound to `count`, so, in our case, `self.start` is now bound to `10`. The constructor terminates (it must always return `None`: in this case it does it implicitly), therefore the local names `self` and `start` go out of scope. The new instance is then returned as the result of the call to the class. We bind it to the name `a`, thereby preventing the reference count from dropping to zero, and keeping the instance alive.

```
a.up(2)
```

When calling a method (`up`) of a class (`Counter`) through its instance (`a`), in this fashion, the instance used is passed in as the first argument to the method. Any arguments explicitly given in the method call are passed in after the instance. In our case, the class instance we stored in the global variable `a` is bound to the `up` method's local variable `self`, and the number `2` is bound to `up`'s local variable `n`. In the constructor, the `count` attribute was bound to `10`. Now it is incremented by `n`, which happens to be `2`, so `self.count` is now 12. The `up` method terminates, so its local variables `self` and `n` go out of scope. The global variable `a` still refers to the instance of `Counter`, keeping it alive.

```
print a
```

Python merely tells us that it is an instance of a class called `Counter` which is in a module called `__main__`.

```
print Counter
```

Not all that helpful.

```
print count
```

There is no global name `count`, so you get a NameError (unless you happen to have introduced a global `count` somewhere along the way).

```
print a.count
```

But there is a `count` among the *instance attributes* of our instance of `Counter`.

```
print Counter.count
```

There is no *class attribute* `count` in the `Counter` class: AttributeError.

```
a.foo = 9
```

We are free to stick attributes onto instances whenever we like, though typically it is done in the constructor.

Queue class

Write a class which implements the queue idea discussed 2 slides ago.

- Move the three functions into a class
- Rename `queue` parameter to `self`.
- Turn `make_queue` into a constructor:
  - Rename `make_queue` to `__init__`
  - Give it a `self` parameter
  - Instead of returning the new list, make it an attribute of the newly created object by sticking it onto `self`.

Inheritance, operators

```
class Addcounter(Counter):

    def __repr__(self):
        return 'Addcounter({.count})'.format(self)

    def __add__(self, other):
        return Addcounter(self.count + other.count)
```

Here we define a new class `Addcounter`, which *inherits* from `counter.` `Addcounter` is a *subclass* of `counter`.

Multiple inheritance (inheritance from more that one class) is supported: list all the supreclasses, separated by commas, between the parentheses following the new class' name.

If a class has no superclasses, its name may be followed by empty parentheses, or the parentheses may be absent altogether. Prior to Python 2.5 empty parentheses were not permitted: they had to be absent in the no-inheritance case.

```
c = Counter(3)
a = Addcounter(4)
c + c
a + a
c + a
a + c
```

Here we see two more magic names in action

- `__add__` When the `+ operator is encountered, Python looks for the __add__ method of the left operand's class (or superclasses).` Failing that, it looks for an `__radd__` method of the right operand's class (or superclasses). Failing that it raises a `TypeError`.
- `__repr__` is called whenever the printed representation of the object is required. Note that `__repr__` must return a string.

Specialized Queue

Create an `EmergencyQueue` class, which does everything that the `Queue` class does, but also has the ability to add items to the *front* of the queue.

Privacy

- Python does not enforce privacy by restricting access.
- Convention: names starting with an underscore (`_`), refer to objects which you should not access directly, outside of their defining scope (class or module).
- Names starting (but not ending) with two underscores, will be mangled
  - Intended as protection against name clashes with subclasses.
- Names which both start and end with two underscores are special names (e.g. `__init__`).
- Exercise: Apply these conventions to the classes we have written so far.

Polymorphism Without Inheritance

- In statically typed languages, polymorphism is usually tightly coupled to inheritance (on interfaces in Java).
- Polymorphism in Python does not depend on inheritance.
- Inheritance is NOT required for heterogeneous containers

User-defined exceptions

- What happens if you try to remove an item from an empty `Queue`?
- `Queue` clients shouldn't be made aware of, and confused by the implementation details of the class.
- Can we send a clearer message about what is going on?
- Write your own `EmptyQueue` exception class and use it in the implementation of `Queue`.

Making modules

- Two types of modules exist:
  - Pure Pthyon module: any file called `<module-name>.py`, containing Python code.
  - Extension module: a shared library conforming to some format.
- Python looks for modules in the directories listed in `sys.path`.
  - `sys.path` includes any paths that are listed in the environment variable `PYTHONPATH` at the time the interpreter is started.

Testing

- Python's standard library includes a unit testing module: `unittest`.
- Test-Driven Development: write test before implementing functionality - the tests are a specifcation of the code you have to write.
- Write a module which passes `all these tests`.

`unittest` belongs to a family of unit testing frameworks in which the first implementation was written in Smalltalk. This was then copied into many other languages. In C++ it is known as CPPUnit, and in Java it is known as JUnit. If you are familiar with those, then you will recognize the features described below.

Look at the source code of the test-suite. You will notice that there are some classes in there which inherit from `unittest.TestCase`. This hooks those classes into the test framework.

You will notice that those classes contain some methods whose names start with `test`. The framework recognizes each of those methods as a single test.

You will also find methods called `setUp`. These are run just before each and every test.

If a `tearDown` method exists (none do, in our case), this is called after each and every test in that class.

Within the methods you will find calls to `AssertEqual`. This takes two arguments: if the two arguments are equal, the test continues; otherwise the test is aborted, and the framework records this test as having failed.

Another test assertion method is `AssertRaises`. This takes at least two arguments. The first is an exception type, the second is a callable. Any further arguments are passed as arguments to the callable. `AssertRaises` calles the callable with the given arguments, and only allows the test to continue if the given exception, or a subtype, is raised by the callable. If no exception is raised, the test is terminated, and registered as having failed.

There are many other assertion methods available (but not present in our example). Try `dir(unittest.TestCase)`.

If a test runs to completion, it is recorded as having passed, and a single dot is written out to the progress report.

If some unhandled exception is raised in a test, the test suite handles it by recording that the test contained an error. A single E is written in the progress report. At the end of the run, a few lines, including a traceback, are printed for that test.

If one of the framework methods (`assertXXX` or `failXXX`) is not satisfied, the framework considers the test to have failed. A single F is printed to the progress report. At the end of the run, a few lines, including a traceback, are printed for that test.

Before you start trying to implement the necessary code, it is important to be sure that you are able to run the test suite, and that the test suite is linking up with your code.

Download the test suite to your local filesystem.

Run the test suite in a separate terminal. Do not try to run the suite interactively. In a fresh UNIX terminal, type `python test-queue.py`. You should see a failure at line 2.

The cause of the faliure should be obivous. The test suite tries to import a module called `pqueue`, and no such module exists. This is the module you are about to write.

Create a flie called `pqueue.py`. It doesn't need to contain anything yet. It just needs to exist. Make sure that it is saved in your working directory. Run the test suite again: `python test_queue.py`. You should now get an error on line 3.

Once again, the reason should be obvious. The test suite tries to import something called `EmptyQueue` from the `pqueue` module. It is not there yet. We do not know what `EmptyQueue` is supposed to be yet, but we want to get past this line, so we do the simplest thing that will allow us to progress: add the line `EmptyQueue = 3` into the source of your module, and save it. We don't care, for now, that this is likely to be wrong: we can deal with that later. For now, we just want to see the tests running. Run the test suite once more.

Now, you should see some output which ends like this:

```
======================================================================
ERROR: Submitting a non-int priority does not raise TypeError
----------------------------------------------------------------------
Traceback (most recent call last):
  File "test-queue.py", line 17, in setUp
    self.q = pqueue.PriorityQueue()
AttributeError: 'module' object has no attribute 'PriorityQueue'

----------------------------------------------------------------------
Ran 6 tests in 0.002s

FAILED (errors=6)
```

What you are seeing is the report produced by the test framework. At the very bottom, you are told that the tests have failed, because there are six errors. Before that you see summaries of each failure. Before that, you should see six `E`s. It is your task to turn those six `E`s into six dots.

Various strategies and approaches to making the tests pass are discussed at length in class.

The priority queue specification

- `add(object, priority)` - adds object to queue.
- `pop()` - remove next object from queue.
- 5 priority levels: 0, 1, 2, 3, 4.
- 0 is the highest priority.
- 2 is the default priority.
- `len(q)` should return the number of items in the queue `q`.
- `pop`ping from an empty queue must raise `EmptyQueue` (a user-defined exception).
- `add`ing with an unacceptable priority must raise `ValueError`.
- `add`ing with a non-integer priority must raise `TypeError`.

List Comprehensions

- `[ x*x for x in range(10) ]`
- `[ (x,y) for x in range(4) for y in 'abc' ]`
- `[ (x,y) for x in range(6) if x%2 for y in range(6) if x>y ]`
-

Python 2.7 onwards

- Dictionary comprehensions
- `{ x:x*x for x in range(10) }`

- **Set comprehensions**

- `{ x*x for x in range(10) }`

Memoization

- Caching results of (expensive) function calls for future use.
- Write a **recursive** function to calculate the `N`th Fibonacci number (yes, it's supposed to be slow!)
- See how slow it gets
  - Careful! start with small arguments and progress very slowly past 35.
  - Write a utility which allows you to time exactly how long it takes to run.

    Hint: I suggest you use the module `time`.

    You need to write a utility which *conceptually* allows you to write

    ```
    timethis(fib(35))
    ```

    in order to find out how long it takes to evaluate `fib(35)`. However, Python has an *eager* or *strict* evaluation strategy. This means that all arguments are evaluated *before* being passed in to the function. Consequently, `fib(35)` would be evaluated before `timethis` starts running. `timethis` would receive the integer 14930352 (the result of evaluating `fib(35)`). The event `timethis` is supposed to measure will have already finished before `timethis` even starts. Therefore, your main problem is to find a way of delaying the evaluation of `fib(35)`. Put another way, you must find a way of communicating which function should be applied to which arguments, and allow `timethis` to decide when that application actually takes place.

- Write a memoizer ...

Memoizer

- Takes a callable as its argument.
- Returns a callable.
- The latter must
  - have identical semantics to the former,
  - never calculate the same value twice.
- Hints:
  - dictionary
  - `instance(...)` ==> `instance.__call__(...)`
  - The memoizer must have **no hard-wired references** to the Fibonacci function. It should be a general memoizer.

  The memoizer should work like this:

  Create a memoized version of your function, by passing your function to the memoizer

  ```
  memfib = memoi(fib)
  ```

  Running the UNmemoized `fib` function should give some result, and take an appreciable amount of time.

  `fib(35)` # => 14930352 in 30 seconds

  Running the UNmemoized `fib` function a second time with the same input should give the same result, and take about the same amount of time.

  `fib(35)` # => 14930352 in 30 seconds

  Running the memoized function with the same input should give the same result, and take about the same amount of time.

  `memfib(35)` # => 14930352 in 30 seconds

  Running the memoized function with the same input a second time, should give the same result, but take ALMOST NO TIME AT ALL.

  `memfib(35)` # => 14930352 in 0 seconds

- Make sure it passes these tests.
- Write an iterative Fibonacci function.

Name resolution

- `mfib = memo(fib)`
- `time(mfib, 30)` # first time: takes a few seconds
- `time(mfib, 30)` # second time: instantaneous
- Now give the memoized version the same name as the original, and repeat the above
- `fib = memo(fib)`
- `time(fib, 30)` # first time: instantaneous
- `time(fib, 30)` # second time: even faster!
- The name `fib` is resolved at runtime, so the memoized version is called by the recursive calls.
- Previously, `fib` referred to the un-memoized function, so the recursive calls called the un-memoized (slow) version.

Speed

Download this tarball, then

- `tar zxvf cython-etc.tgz`
- `cd cython-etc`
- `source source-me.sh`

Generators

```
from __future__ import generators  # only in Python 2.2

def gen_ints(start, stop):
    while start < stop:
        yield start       # next(...)
        start += 1
    return                # StopIteration


a = gen_ints(3,6)
for i in a: print i

a = gen_ints(10,12)
next(a)
next(a) # keep repeating this interactively
```

Python's iterator protocol offers an example of duck typing: the actual type of the object does not matter as long as the object is able to act like an iterator. In order for it to be an iterator, it must have a magic method `__next__` (accessed through the builtin `next(...)`) which either returns the next item, or signals the absence of further items by rasing the `StopIteration` exception. (Note that `StopIteration` is an example of an exception which is not an error.) To be fully compliant with the iterator protocol you also need the magic method `__iter__`. You can find the full details here.

itertools

- New module in Python 2.3.
- Use it to re-implement `enumerate` (Use this test to check it.)
  `def Enumerate(it): return izip(count(), it)`
- Create a collection of the squares of all the fibonacci numbers which are greater than 1000
  `fibsquares_gt_1000 = imap(lambda x:x*x, dropwhile(lambda x:x<=1000, fibg()))`

Generator Expressions

- Introduced in Python 2.4
- Similar to list comprehensions, but lazy
- `a = (n*n for n in xrange(10))`
  `next(a)`
- Replace the `imap` in the example on the last slide with a generator expression
  `fibsquares_gt_1000 = (x*x for x in dropwhile(lambda x:x<=1000, fibg()))`
- `sum(n*n for n in xrange(10))`

  `When a generator expression appears as the only argument to a function, it does not need an extra set of parentheses. The parentheses of the function call can double up as the parentheses of the generator expression syntax.`

Pervasive laziness in Python 3

- `range`
- `map, filter, zip`
- `dict.keys()` etc
-

State changes

```
p = Happy('Proteus')

p.greet()     # Proteus says "Hellloooo!"
p.sing()      # Lah-de-daaaaah!
p.change()

p.greet()     # Proteus says "Get lost!"
p.sing()      # Proteus is too depressed to sing.
p.change()

p.greet()     # Proteus says "Hellloooo!"
p.sing()      # Lah-de-daaaaah!
```

State Pattern

```
class Happy:
    def __init__(self, name):
        self._name = name
    def greet(self):
        print (self._name, 'says "Hellloooo!"')
    def sing(self):
        print ('Lah-de-daaaaah!')
    def change(self):
        self.__class__ = Sad

class Sad:
    def greet(self):
        print(self._name, 'says "Get lost!"')
    def sing(self):
        print(self._name, "Is too depressed to sing")
    def change(self):
        self.__class__ = Happy
```

Attribute lookup

```
class cls:
    a = 1
```

Create a trivial class. Give it a class attribute called `a`.

```
inst = cls()
```

Instantiate the class

```
dir(cls)
```

`a` is accessible through the class ...

```
dir(inst)
```

... and through the instance.

```
cls.__dict__
```

`a` is present in the class dictionary ...

```
inst.__dict__
```

... but not in the instance dictionary.

Bind the existing attribute `a`, and a completely new attribute `b`, via the instance.

```
inst.a = 22
inst.b = 333
```

Now repeat the green stuff.

```
dir(cls)
dir(inst)
cls.__dict__
inst.__dict__
```

Some `a` or other is accessible through both the class and the instance. `b` is accessible only through the instance.

`b` is present only in the instance dictionary. There are two different `a`s, one in the class dictionary and one in the instance dictionary.

Attributes accessed through class instances are looked up first in the instance itself, then in the instance's class, then in any superclasses. The first found is the one used.

Attributes bound through a class instance are always bound to the instance itself.

Binding Mechanisms

In the surrounding scope

1. `a = ...`
2. `def a(...):`
3. `class a(...):`
4. `import a`
   `from b import a`
   etc.
5. for `a in ...`
6. `except ... as a`
   `with ... as a`
   etc.

Binding Mechanisms

In the local scope

1. `def b(a):`
2. `lambda a: ...`

In the `b` namespace

1. `b.a = ...`
2. `setattr(b, 'a', ...)`
3. `b.__dict__['a'] = ...`

Type/Class Unification

- Initially, user defined types were distinct from the built-in types; it was impossible to derive your own classes from the built-ins.
- All the classes we have written so far are 'classic' classes; these are distinct from the built-in types.
- With 'new-style' (as opposed to 'classic') classes it is possible to derive user-defined classes from the built-ins.
- A new-style class is one which inherits (directly or indirectly) from `object`.

Type/Class Unification

```
class Old: pass

class New(object): pass

oldinstance = Old()
newinstance = New()

type(Old), type(oldinstance)  # classobj, instance
type(New), type(newinstance)  # type,     New

type(Old()) is Old            # False
type(New()) is New            # True
```

I advise you to use new-style classes by default from now on, because

- They are better than classic classes. (For example, the next exercise we do is most easily done with properties which only work fully in new-style classes.)
- New-style classes are the only kinds of classes available in Python 3; making all your classes new-style will leave you with less work to do when you come to migrate your code to Python 3.
-
-

Say "No!" to getters and setters

- In languages such as Java and C++ there is a strong dogma demanding that all data members must be private ...
- ... even if you then make them public, by providing getters and setters for them.
- The purpose of getters and setters is to make all attributes look like functions, even those which really are data.
- This allows you to change your mind about the implementation details, while guaranteeing that you can keep the interface stable.
- In Python, we have the ability to make function attributes look like data attributes. This allows us change our mind about the implementation details, without breaking the interface, even if we have public data attributes.
- It's perfectly OK to let classes have public data in Python.
- Do NOT write trivial getters and setters in Python.

Properties

- Read `help(property)` and solve the following problem without changing the original interface of the class.
- ```
  class Rectangle(object):

      def __init__(self, w, h):
          self.w = w # Yes, just make it public!
          self.h = h
          self.a = w*h

  r = Rectangle(2,3)

  assert r.w ==  2
  assert r.h ==  3
  assert r.a ==  6

  r.w = 4

  assert r.w ==  4
  assert r.h ==  3
  assert r.a == 12 # FAILS HERE
  ```
- Unittest tests
- Guided solution

Decorators

- Static and class methods
- Decorator syntax (2.4)
- Use property as a decorator in our earlier rectangle example. Note the `setter` utility which was added in Python 2.6
- Use the decorator syntax in conjunction with the memoizer.
- Write a decorator for generator functions, which enables `send` to be used right from the start.

RAD

- Develop in HLL, **benchmark** ... deploy or ...
- **Profile**, explore solution space, **benchmark** ... deploy or ...
  - Getting the architecture right is much more significant than microoptimization.
- **Profile**, recode hotspots in LLL, **benchmark** ... deploy or ...
- Recode everything in LLL, **benchmark** ...
- CPU time is (increasingly) much cheaper than programmer time.
  ```

I hope that

- You are comfortable with writing functions and classes in Python.
- You know how to get online help (and do so).
- You appreciate that the interactive, dynamic and introspective nature of Python significantly helps the development process.
- You have learned at least one new programming technique or concept.
- You realize that you should not write Fortran, C++, Perl etc. in Python.
- You had some fun.

Code samples

- Password file sorting
- Sum numbers on lines in file
- Fibonacci implementations
- Memoizer
- Priority queue test suite and implementation
- Enumerate test suite and sample solutions.
- Decorator pattern in Python and C++
- State pattern in Python and C++
- Properties