Structuring Python Code

Jonathan Dursi

SciNet HPC Consortium

December 2, 2014

Structuring code: Functions, Classes, Modules, Packages, Testing, and Python/Eclipse

Outline for Today:

- Functions
  - Python Functions
  - Using Eclipse for Refactoring
- Objects (Ramses)
- Advanced Functions
  - Functions as Objects
  - Generators, Closures
- Modules
- Testing
  - Unittest, Nose, Doctest
- Packages

Part 1 - Functions

- Defining Functions
- Docstrings, help, and pydoc; Ex1
- Scoping - LEGB
- Parsing; Ex2
- Recursion; Ex3
- Keyword parameters
- Functions as arguments; Ex4
- Eclipse and refactoring
- Testing parameters with asserts
- Ex 5

Defining Functions

In an interactive python session (eclipse interpreter, or IDLE), let's type in the following.

```
def addition(x,y):
    """This function returns the sum of x and y"""
    sumxy = x + y
    return sumxy

print addition(3,5)
```

```
## 8
```

Defining Functions - Structure

Let's look at a couple of things here:

```
def addition(x,y):
    """This function returns the sum of x and y"""
    sumxy = x + y
    return sumxy

print addition(3,5)
```

- `def` keyword introduces the definition of a function.
- Function is a code block, so:
- Usual python indentation requirement
    - Indentation as syntax is clearly and unquestionably a good thing, and I will not hear otherwise.

Defining Functions - Variables

Let's look at a couple more things:

```python
def addition(x,y):
    """This function returns the sum of x and y"""
    sumxy = x + y
    return sumxy

print addition(3,5)
```

- Introduction of variables in the usual pythonic way
- `return` statement for returning values.
- `return` statement is optional; if absent, function returns `None`.

Defining Functions - Docstrings and help

```python
def addition(x,y):
    """This function returns the sum of x and y"""
    sumxy = x + y
    return sumxy

help(addition)
```

```
## Help on function addition in module __main__:
##
## addition(x, y)
##     This function returns the sum of x and y
```

- String immediately following definition becomes the docstring for the function; can be accessed with `help()` and other methods.
- By convention, three quotes, so can easily be made multi-line.

Defining Functions - Docstrings and help

```
## Help on function addition in module __main__:
##
## addition(x, y)
##     This function returns the sum of x and y
```

- Because of automatic documentation of the function's signature, particularly useful to give arguments descriptive names.
- Helps explain their use, minimizes additional documentation you have to write.
- Python *can't* "see" the return type of the function: explicitly documenting what it returns in the docstring is usually necessary.

Defining Functions - Docstrings and help

```python
def addition(x,y):
    """This function returns the sum of x and y"""
    sumxy = x + y
    return sumxy

help(addition)
```

```
## Help on function addition in module __main__:
##
## addition(x, y)
##     This function returns the sum of x and y
```

- If you are unwilling to write even a single-line description of your function for your colleagues, look deep inside yourself and ask yourself why.

Arguments: pass by reference or by value?

The natural first question of any programmer who works with both C/C++ and Fortran code.

- The answer is. . .
- Well, let's see.

```
def mutateInteger(i):
    """Takes an integer argument and doubles it."""
    i = i * 2

def mutateListItem(l):
    """Takes an list argument and doubles second item."""
    l[1] = l[1] * 2

def mutateList(l):
    """Takes an list argument replaces it."""
    l = [1,2,3]
```

Arguments: it's complicated.

```
def mutateInteger(i):
    """Takes an integer argument and doubles it."""
    i = i * 2

def mutateListItem(l):
    """Takes an list argument and doubles second item."""
    l[1] = l[1] * 2

def mutateList(l):
    """Takes an list argument replaces it."""
    l = [1,2,3]

i = 1;      print i,'->',; mutateInteger(i);  print i;
l=[2,4,6]; print l,'->',; mutateListItem(l); print l;
l=[2,4,6]; print l,'->',; mutateList(l);     print l;
```

```
## 1 -> 1
## [2, 4, 6] -> [2, 8, 6]
## [2, 4, 6] -> [2, 4, 6]
```

t

Arguments are “pass by assignment”

```
## 1 -> 1
## [2, 4, 6] -> [2, 8, 6]
## [2, 4, 6] -> [2, 4, 6]
```

- Function dummy arguments are assigned as labels to the passed arguments: another reference to the object.
- If the object allows mutation, can *change* it. Can’t mutate tuples, strings, primitive types.
- *Replacing* it doesn’t work: just makes local dummy argument reference a new, different, local, thing.
- We’ll be able to understand this a little more after discussing objects in python.
- What happens if you pass `mutateListItem()` a tuple?

Multiple Return Values

Modifing function parameters is less necessary in python.

Commonly used to have function return several values.

Can easily return multiple values from a function using tuples, explicitly or implicitly:

```
def minmeanmax(items):
    """Returns summary statistics of a sequence - min,mean,max."""
    minval = min(items)
    meanval= sum(items)/len(items)
    maxval = max(items)
    return (minval, meanval, maxval)   # explicit
```

Multiple Return Values

```python
def minmeanmax(items):
    """Returns summary statistics of a sequence - min,mean,max."""
    minval = min(items)
    meanval= sum(items)/len(items)
    maxval = max(items)
    return minval, meanval, maxval    # implicit tuple

tup = minmeanmax(range(-50,51))
print tup
low, middle, high = minmeanmax(range(1,200))
print "low = ", low, "mid = ", middle, "hi = ", high
```

```
## (-50, 0, 50)
## low =  1 mid =  100 hi =  199
```

Tuple implicitly created from comma list of values, implicitly unpacked into seperate variables after second function call.

Exercise - functions (5-10 min)

- In eclipse, start a new file.
- Define a function `isSquare(n)` which returns true or false depending on whether or not n is a square.
- Using `isSquare()`, write a function which returns the sum all the square numbers from a up to but not including b.
- (Bonus points: do same for all triangle numbers, $T = \frac{n(n+1)}{2}$.)
- Sum of squares in [1,100): 285
  - [1, 4, 9, 16, 25, 36, 49, 64, 81]
- Sum of squares in [1537,2089): 10855
- Sum of triangles in [1,100): 455
  - [1, 3, 6, 10, 15, 21, 28, 36, 45, 55, 66, 78, 91]
- Sum of triangles in [1537,2089): 18040

Scoping

Python has fairly sensible scoping rules for functions that will be familiar to most of us. Let's go back to that IDLE session with addition:

```
def addition(x,y):
    """This function returns the sum of x and y"""
    sumxy = x + y
    return sumxy

print addition(3,5)
print sumxy
```

```
## 8
## Traceback (most recent call last):
##   File "<string>", line 7, in <module>
## NameError: name 'sumxy' is not defined
```

Scoping

```
## 8
## Traceback (most recent call last):
##   File "<string>", line 7, in <module>
## NameError: name 'sumxy' is not defined
```

- Local variable `sumxy` does not exist outside the function
  - (unlike other code blocks, like say if-blocks)
- Important reason for functions, higher-level structures: encapsulation.
  - Don't "leak" state.

Scoping Rules: Local, Enclosing, Global, Builtin

- LEGB lookup priority:
  - Local variables
  - Enclosing blocks
  - Global variables
  - Builtin python functions/values.
- `sumxy` was neither L,E,G, nor B.
- Note that it is very, very easy to hide (shadow) other definitions, including your own or python functions.
- Python won't warn you of this, but IDEs often will.

Global considered harmful

Global variables are, of course, evil, and we will speak no further of them here.

Python and Parsing

Going back to our addition example, what do the following do?

- `print addition('Hello ','World!')`
- `print addition(1.5, 3.7)`
- `print addition(4, "World!")`

Python and Parsing

```
def addition(x,y):
    """This function returns the sum of x and y"""
    sumxy = x + y
    return sumxy

print addition(3, 4)
print addition('Hello ','World!')
print addition(1.5, 3.7)
print addition(4, "World!")
```

```
## 7
## Hello World!
## 5.2
## Traceback (most recent call last):
##   File "<string>", line 9, in <module>
##   File "<string>", line 3, in addition
## TypeError: unsupported operand type(s) for +: 'int' and 'str'
```

Python and Parsing

Python performs only the most cursory examination of code before execution.

- Means you can do a lot of neat things without writing much boilerplate
- Means a lot of errors can't possibly be caught until runtime
  - Ticking timebombs in your code.

What does this do? (note typo, and print as function)

```
def printParity(n):
    if n % 2 == 0:
        print("Even!")
    else:
        prnit("Odd!")

printParity(6)
printParity(4)
printParity(5)
```

Python and Parsing

```
def printParity(n):
    if n % 2 == 0:
        print("Even!")
    else:
        prnit("Odd!")

printParity(6)
printParity(4)
printParity(5)
```

```
## Even!
## Even!
## Traceback (most recent call last):
##   File "<string>", line 9, in <module>
##   File "<string>", line 5, in printParity
## NameError: global name 'prnit' is not defined
```

Exercise - Broken Code (5 min)

Take a few minutes to come up with the most broken code (as scored by either the obvious-wrongness of the code, or the spectacularity of the resulting error messages) you can which will still run correctly in some cases.

Winner gets lasting fame, extra 2 minutes of coffee break.

Python and Parsing - Need to Test

JIT compiling represents a tradeoff: flexibility (ease to get working code running) vs level of insight into the code.

This isn't a problem, necessarily, but it means you need some other tool than a compiler to make sure all code makes sense.

Various testing frameworks exist which can help with this - will see some this afternoon.

Asserting parameter validity

- Even without a test suite, the beginning of a function is an excellent place to check parameters being passed in are valid.
- Fail as *early* as possible, makes the underlying mistake easier to find.
- Can use the rather blunt instrument of an assertion:

```python
def additionAssert(x,y):
    """This function returns the sum of x and y"""
    assert type(x) == type(y)
    sumxy = x + y
    return sumxy

print additionAssert(4, "World!")
```

```
## Traceback (most recent call last):
##   File "<string>", line 7, in <module>
##   File "<string>", line 3, in additionAssert
## AssertionError
```

Asserting parameter validity

- Or, less drastically and more pythonically, raise an exception:

```
def additionException(x,y):
    """This function returns the sum of x and y"""
    if not type(x) == type(y):
        raise ValueError("Mismatched types")
    sumxy = x + y
    return sumxy

print additionException(4, "World!")
```

```
## Traceback (most recent call last):
##   File "<string>", line 8, in <module>
##   File "<string>", line 4, in additionException
## ValueError: Mismatched types
```

Recursion

Recursion is handled in python by applying recursion.

Python won't automatically optimize out tail recursion, so be careful, but can still be useful.

Legally-mandated Fibonacci sequence example:

```
def fibonacci(n):
    if n < 3:
        return 1
    else:
        return fibonacci(n-1) + fibonacci(n-2)

print fibonacci(7)
```

```
## 13
```

Exercise - Recursion (5-10 min)

Pick one of the following:

- Define a function which recursively determines if a word is a palindrome.
  - One character words are necessarily palindromes.
- Fast exponentiation: $\boldsymbol{base^{pow}}$ in as few multiplications as possible.
  - For even pow, calculate $\boldsymbol{base^{pow/2}}$, multiply it by itself.
  - For odd pow, calculate $\boldsymbol{base \cdot base^{pow-1}}$.

Keyword parameters

Let's consider another way of calling our exponentiation function:

```
def fastExponentiation(base, power):
    if power == 0:
        return 1
    if power % 2 == 1:
        return base * fastExponentiation(base, power-1)
    else:
        halfpow = fastExponentiation(base,power/2)
        return halfpow*halfpow

print fastExponentiation(2,12)
print fastExponentiation(power=12, base=2)
```

```
## 4096
## 4096
```

Keyword parameters

We aren't restricted to passing in arguments in order they are listed in function; we can pass in arguments in any order, as long as we specify them.

Can also define optional parameters with default parameters:

```
import math

def myLogarithm(x, base=10):
    """Calculates the logarithm of x, with the given base.
       base: defaults to 10."""
    return math.log(x) / math.log(base)

print myLogarithm(100)
print myLogarithm(16,2)
print myLogarithm(16,base=4)
```

```
## 2.0
## 4.0
## 2.0
```

Keyword parameters

Optional or keyword-specified arguments have to come last. Why?

```
def myLogarithm(x, base=10):
    """Calculates the logarithm of x, with the given base.
       base: defaults to 10."""
    return math.log(x) / math.log(base)

print myLogarithm(100)
print myLogarithm(16,2)
```

Does `help(myLogarithm)` tell me the default values of parameters?

Keyword parameters

Note: be very careful about using something mutable as a default argument, and then mutating it.

```
def appendTo(item, to=[]):
    to.append(item)
    return to

print appendTo(1)
print appendTo(2, [2])
print appendTo(3)
```

```
## [1]
## [2, 2]
## [1, 3]
```

Functions as arguments

Functions, once created, are things that you can assign to variables, etc:

```python
def addition(x,y):
    return x+y

print addition(2,3)

sumThemUp = addition
print sumThemUp(2,3)
```

```
## 5
## 5
```

Functions as arguments

And that means that they're easy to pass into functions as arguments

```python
def addition(x,y):
    return x+y

def subtraction(x,y):
    return x-y

def applyAction(x,y,f):
    return f(x,y)

print applyAction(5,2,addition)
print applyAction(5,2,subtraction)
```

```
## 7
## 3
```

Exercise - Simple Quadrature (10-15 minutes)

- Write a simple numerical integrator function:
  - Takes a starting and ending value
  - And a function y = f(x)
  - and an optional number of steps
- and outputs a simple trapezoid-rule approximation to the integral,

$$\int_a^b f(x)dx \approx \sum_{i=1}^{N} h\frac{f(x_{i+1}) + f(x_i)}{2}$$

$$h = \frac{b-a}{N}; \quad x_i = a + (i-1)h$$

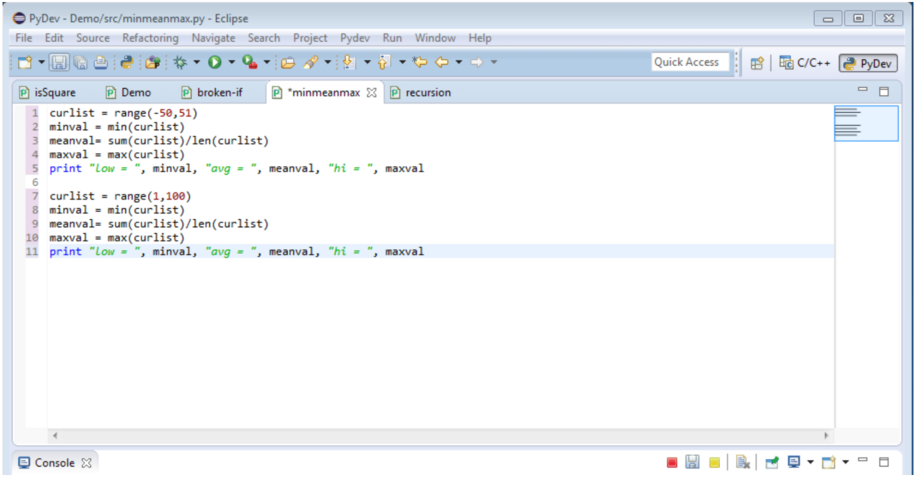
Eclipse refactoring

Eclipse, as with most IDEs, makes it **very** straightforward to pull functions out of code.

Whenever you see repeated code, it is a sign that some refactoring needs to happen to re-use that repeated code.

Happens all the time when doing scripting, extending scripts.

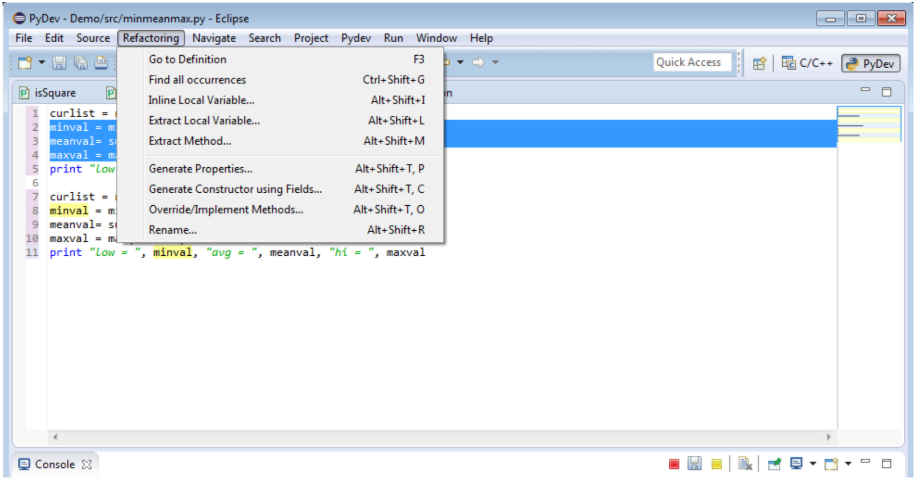
Eclipse refactoring


PyDev - Demo/src/minmeanmax.py - Eclipse
File Edit Source Refactoring Navigate Search Project Pydev Run Window Help
Quick Access
C/C++
PyDev
isSquare
Demo
broken-if
*minmeanmax
recursion
Console

```
curlist = range(-50,51)
minval = min(curlist)
meanval= sum(curlist)/len(curlist)
maxval = max(curlist)
print "low = ", minval, "avg = ", meanval, "hi = ", maxval

curlist = range(1,100)
minval = min(curlist)
meanval= sum(curlist)/len(curlist)
maxval = max(curlist)
print "low = ", minval, "avg = ", meanval, "hi = ", maxval
```

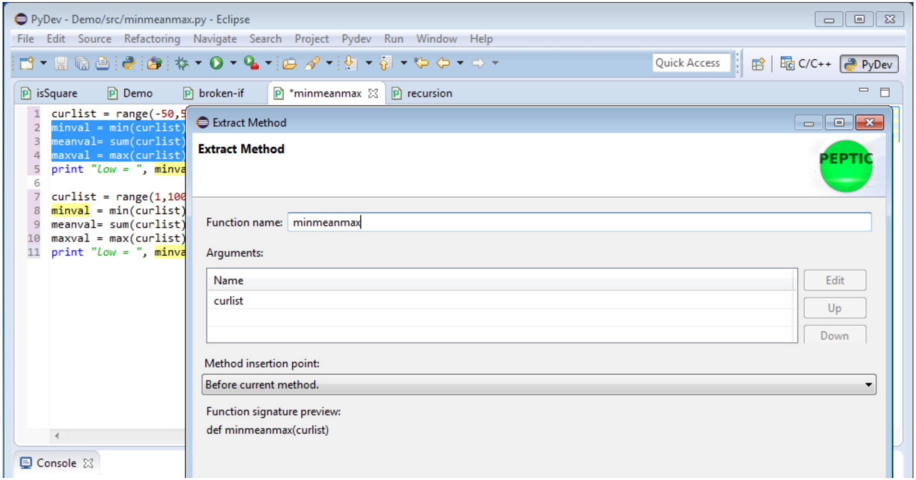
Eclipse refactoring


PyDev - Demo/src/minmeanmax.py - Eclipse
File Edit Source Refactoring Navigate Search Project Pydev Run Window Help
Go to Definition F3
Find all occurrences Ctrl+Shift+G
Inline Local Variable... Alt+Shift+I
Extract Local Variable... Alt+Shift+L
Extract Method... Alt+Shift+M
Generate Properties... Alt+Shift+T, P
Generate Constructor using Fields... Alt+Shift+T, C
Override/Implement Methods... Alt+Shift+T, O
Rename... Alt+Shift+R
Quick Access
C/C++
PyDev
isSquare
Console

```
curlist = 
minval = m
meanval= s
maxval = m
print "low

curlist = 
minval = m
meanval= s
maxval = m
print "low = ", minval, "avg = ", meanval, "hi = ", maxval
```

Eclipse refactoring


PyDev - Demo/src/minmeanmax.py - Eclipse
File Edit Source Refactoring Navigate Search Project Pydev Run Window Help
Quick Access
C/C++
PyDev
isSquare
Demo
broken-if
*minmeanmax
recursion
Console
Extract Method
PEPTIC
Edit
Up
Down

```
curlist = range(-50,5
minval = min(curlist)
meanval= sum(curlist)
maxval = max(curlist)
print "Low = ", minva

curlist = range(1,100
minval = min(curlist)
meanval= sum(curlist)
maxval = max(curlist)
print "Low = ", minva
```

Extract Method

Function name: minmeanmax

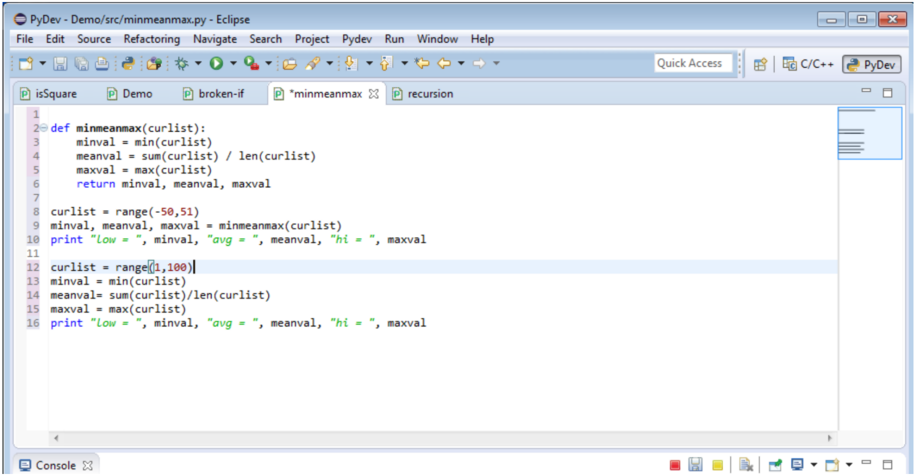
Arguments:

Name
curlist

Method insertion point: Before current method.

Function signature preview:
def minmeanmax(curlist)

Eclipse refactoring


PyDev - Demo/src/minmeanmax.py - Eclipse
File Edit Source Refactoring Navigate Search Project Pydev Run Window Help
Quick Access
C/C++
PyDev
isSquare
Demo
broken-if
*minmeanmax
recursion
Console

```

def minmeanmax(curlist):
    minval = min(curlist)
    meanval = sum(curlist) / len(curlist)
    maxval = max(curlist)
    return minval, meanval, maxval

curlist = range(-50,51)
minval, meanval, maxval = minmeanmax(curlist)
print "low = ", minval, "avg = ", meanval, "hi = ", maxval

curlist = range(1,100)
minval = min(curlist)
meanval= sum(curlist)/len(curlist)
maxval = max(curlist)
print "low = ", minval, "avg = ", meanval, "hi = ", maxval
```

Exercise - refactor a script (15-30 min)

Python is a great language to use to script to glue together a bunch of steps in a pipeline. But a linear script can grow and take on new tasks and eventually become an unreadable mess.

Such an unreadable mess awaits you in `script/runscript.py` (along with the "programs" it runs, `runthermalsimulation.py` and `runturbulentsimulation.py`.)

Using eclipse, refactor this mess into something maintainable and useful. There's a lot that can potentially be done here.

Advanced Functions

Outline

- Revisiting argument passing
- The `*args` in `**kwargs` fall mainly on the blargs
- Functions as Objects
- Higher level functions: Map/Reduce/Filter
- Lambdas; Ex 1
- Generators

More on argument passing

- Every argument is an object
- *args and **kwargs

More on argument passing

- Everything is an object
- This includes ints, floats, . . .
- “Passing by assignment”: local function argument becomes label for incoming object.
    - Some classes have methods for mutating the content of an object (*eg*, lists, dicts).
    - Some don’t (eg, strings, tuples, primitive types).
- May or may not be able to modify arguments.
- Either way, can’t simply replace them - just change what local label points to.

`*args and **kwargs`

What does the following do?

```
def printArgsKwargs(firstarg, *args, **kwargs):
    print 'firstarg: ', firstarg
    print 'args:     ', args
    print 'kwargs:   ', kwargs

printArgsKwargs(1,'two',3.0,greeting='Hello',greetee='World')
```

*args and **kwargs

```
def printArgsKwargs(firstarg, *args, **kwargs):
    print 'firstarg: ', firstarg
    print 'args:     ', args
    print 'kwargs:   ', kwargs

printArgsKwargs(1,'two',3.0,greeting='Hello',greetee='World')
```

```
## firstarg:  1
## args:      ('two', 3.0)
## kwargs:    {'greeting': 'Hello', 'greetee': 'World'}
```

- args: tuple of all (remaining) positional arguments.
- kwargs: dictionary of all (remaining) keyword arguments
- The relevant syntax here is * and **; args and kwargs could be anything (but convention is args and kwargs).

*args and **kwargs

These are very useful in two cases in particular:

- You don't know how many arguments you'll have

```
def mySum(*args):
    return sum(args)

print mySum(1,2,7,18.3,-5)
print mySum(3,8,6)
```

```
## 23.3
## 17
```

- (But are you sure you don't just want to use a list?)

*args and **kwargs

These are very useful in two cases in particular:

- You're writing a wrapper to other functions and you want to pass arguments through without explicitly handling them all:

```python
import subprocess

def getDirectory(extension, **kwargs):
    output = subprocess.check_output(['dir','*'+extension],
                                     shell=True,**kwargs)
    return output

print getDirectory('.py')
print getDirectory('',cwd='..')
```

- Note 'unpacking' the dictionary...

*args and **kwargs

The * or ** syntax can be used to unpack a tuple or a dictionary in an argument list:

```
def printArgsKwargs(firstarg, *args, **kwargs):
    print 'firstarg: ', firstarg
    print 'args:     ', args
    print 'kwargs:   ', kwargs

t = (1,'two',3.0)
printArgsKwargs(*t, greeting='Hello',greetee='World')
```

```
## firstarg:  1
## args:      ('two', 3.0)
## kwargs:    {'greeting': 'Hello', 'greetee': 'World'}
```

Functions as Objects

- Everything is an object - including functions.
- That's why it's so easy to pass functions as arguments.
- Let's take our favourite function, or any you have to hand:

```
def addition(x,y):
    """This function returns the sum of x and y"""
    sumxy = x + y
    return sumxy

print dir(addition)
print addition.__doc__
print addition.func_name
```

```
## ['__call__', '__class__', '__closure__', '__code__', '__defaults_
## This function returns the sum of x and y
## addition
```

Functions as Objects

- Doc strings can be accessed programatically via `function.__doc__`
- Function name can be accessed programatically via `function.func_name`
- `__call__` is an alias to the function method itself
- Other attributes can be seen by interactively typing `dir(function)`

Higher level functions - `map`, etc.

Being able to pass in various functions to various routines is very handy.

Sort is a classic example: sort by various criteria. Absolute values?

```
data = range(-10,10,2)
print sorted(a, key=abs)
```

```
## Traceback (most recent call last):
##   File "<string>", line 2, in <module>
## NameError: name 'a' is not defined
```

Higher level functions - `map`, etc.

Higher level functions allow us to control the application of our functions.

- `map(f,sequence)`: applies `x -> f(x)` to each element in the sequence, return the transformed sequence back.
- `reduce(f,sequence)`: combines the sequence to one item, repeatedly applying `partial, next -> f(partial, next) to sequence left to right`
- `filter(f,sequence)`: generates a new sequence consisting only of sequence items where `f(item) == True`.

All of the above can be done with loops: but applying directly over a list is faster, **and** fewer lines of code.

map

Let's try converting a list of angles in degrees to the sine of the angles:

```
import math
def sinDegree(x):
    return math.sin(math.pi*x/180.)

a = range(0,91,30)
print a
print map(sinDegree, a)
```

```
## [0, 30, 60, 90]
## [0.0, 0.49999999999999994, 0.8660254037844386, 1.0]
```

reduce

Reduce walks the sequence left to right, combining the running total so far with the next item and updating the total.

```
a = [1, 3, -7, 16, 0, 5, 2]

def mySum(a,b):
    return a+b

def myProd(a,b):
    return a*b

print reduce(min, a)
print reduce(mySum, a)
print reduce(myProd, a)
```

```
## -7
## 20
## 0
```

filter

Filter pulls out those items in the sequence from which the function passes something that can be interpreted as True.

```
import math

def isSquare(n):
    m = int(math.sqrt(n))
    return m*m == n

print filter(isSquare, range(100))
print sum( filter(isSquare, range(100)) )
```

```
## [0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
## 285
```

Lambdas

Sometimes, (*eg*, interactively) it is a pain to have to define a function for map/filter/reduce that you'll never use again.

Lambdas are "anonymous" functions that can be defined in place for a particular purpose.

Eg, that product function:

```
a = [1, 3, -7, 16, 5, 2]

def myProd(x,y):
    return x*y

print reduce(myProd, a)
print reduce(lambda x,y:x*y, a)
```

```
## -3360
## -3360
```

Lambdas

The syntax of lambda is:

```
lambda arg1,arg2,... : value_to_return
```

Note that this *only* works for simple expressions - no if statements, multi-line expressions.

Can use it wherever a function would go (map, filter, sorted, reduce, . . .)

Lambdas

If you like, you can use

```
f = lambda x,y:x*x+y
```

as the pleasingly and pointlessly cryptic equivalent of

```
def f(x,y):
    return x*x + y
```

Lambdas (but `operator`)

Many of the lambdas that you might want to put in to a filter/map/etc function are already defined in the `operator` module:

```
import operator

inventory = [('Lemon',2),('Orange',7),('Pear',3),('Bananas',0),('Pi
print map( lambda x:x[0] , inventory )
print map( operator.itemgetter(0), inventory )

a = [1, 3, -7, 16, 5, 2]
print reduce( lambda x,y:x*y, a )
print reduce( operator.mul, a )
```

```
## ['Lemon', 'Orange', 'Pear', 'Bananas', 'Pineapple']
## ['Lemon', 'Orange', 'Pear', 'Bananas', 'Pineapple']
## -3360
## -3360
```

Lambdas (but colleagues)

Lambdas are cool for interactive use, but they're definitely harder to read than the equivalent `def`'ed function - not least of which because you can document those.

Think of your colleagues – including yourself, two months later.

Suggested approach for using lambdas:

- Write a lambda.
- Write a comment, explaining the lambda.
- Pick out the most important word in the comment.
- Write a `def`'ed function, named that important word.
- Delete the lambda.

Exercise - map reduce; word count (20-30 min)

In the file `mapreduce.py`, there is the outline of some code to do wordcounts of a text.

Fill it in so it works:

- Write a function to map over the word list to strip out punctuation, convert to lowercase.
- Write a function to filter out words that are empty after we've stripped out punctuation.
- Write a function to reduce the wordlist to the dictionary of counts.

Note: there's a trick to doing the reduce. . .

Bonus points: (free extra mug of coffee) - write a more concise way of outputting the most-often-occuring word and/or deal with ties.

Generators

Because functions are objects, they can have various methods and functionalities.

In particular, it's quite easy to make an iterable from a function; instead of `returning` a value, we `yield` it:

Generators

```
import math

def squares(start,end):
    n = start
    while n < end:
        m = int(math.sqrt(n))
        if m*m == n:
            yield n
        n = n + 1

def sumSquares(start, end):
    sumsq = 0
    for square in squares(start,end):
        sumsq = sumsq + square
    return sumsq

print sumSquares(0,100)
```

```
## 285
```

Generators

Or even just (since sum takes an iterable):

```
import math

def squares(start,end):
    n = start
    while n < end:
        m = int(math.sqrt(n))
        if m*m == n:
            yield n
        n = n + 1

print sum(squares(0,100))
```

```
## 285
```

Generators

What's the point?

- Many very nice tools for applying functions to/over lists.
- Allows you to write nice simple code that looks like it's handling lists:
  - Without ever explicitly generating the list.
  - "Lazy" evaluation.
- Maybe the list is huge, and would take a lot of memory.
- Maybe calculating each item in the list is super expensive (processing a large file) and you don't know ahead of time how many files you'll need to go through to get the answer.
- This allows you to "hide" iteration inside code that looks like you're just doing list manipulations - sum, filter, etc.

itertools

The `itertools` module has a group of very handy functions for creating iterators out of other sequences, again without ever explicitly storing or generating the whole sequence:

- product

```
import itertools

a = [1,2,3]
b = [4,5]
for i in itertools.product(a,b):
    print i
```

```
## (1, 4)
## (1, 5)
## (2, 4)
## (2, 5)
## (3, 4)
## (3, 5)
```

itertools

The itertools module has a group of very handy functions for creating iterators out of other sequences, again without ever explicitly storing or generating the whole sequence:

- permutations/combinations

```
import itertools

a = [1,2,3]
for i in itertools.permutations(a):
    print i
```

```
## (1, 2, 3)
## (1, 3, 2)
## (2, 1, 3)
## (2, 3, 1)
## (3, 1, 2)
## (3, 2, 1)
```

Modules: Bundling Code and Data

Outline

- We're already module-writing experts
- `__main__`
- Command line arguments - `sys.argv`, `argparse`
- Module contents
- Module docstreams
- `pydoc`
- Bytecode compilation

Psst - we have been creating modules this whole time

Let's take one of the existing files we have, and try running the functions inside:

```
import minmeanmax

## (-50, 0, 50)
## low =  1 mid =  100 hi =  199

minmeanmax.minmeanmax([1,2,3])

## (1, 3, 5)
```

Can access the functions exactly as through system modules. Can also use `from ... import ...`, etc.

Importing

When a module is imported, the source is read in and parsed (unless bytecode compiled - more later) and executed.

- That's how the functions come to be declared, etc.
- And that's why importing minmeanmax resulted in immediate output.

But in many cases we may want to have some lines of code that run and use the declared functions – not as initialization, but as a test, or to demonstrate their use.

We want it both ways – importing the file as a module should not run that test/demo code, but running it standalone should.

__main__

We can test if the context in which this file is being read in is the main program (eg, we're running it standalone) or if it is being pulled in as an imported module:

```
def minmeanmax(items):
    """Returns summary statistics of a sequence - min,mean,max."""
    minval = min(items)
    meanval= sum(items)/len(items)
    maxval = max(items)
    return (minval, meanval, maxval)

if __name__ == "__main__":
    tup = minmeanmax(range(-50,51))
    print tup
    low, middle, high = minmeanmax(range(1,200))
    print "low = ", low, "mid = ", middle, "hi = ", high
```

`__main__`

Running the code above from the command line, *eg*

```
C:\...> python minmeanmax.py
```

or from within eclipse will now produce output as before; importing it as a module will not.

We can further customize running the program directly from the commandline by taking and interpreting command-line arguments, using (eg) system modules `sys`, or `argparse`.

sys.argv

Command line arguments can be seen via `sys.argv`, as in C.

(There's no `argc`; why?)

```
import sys

print sys.argv

## C:..> python foo.py a b c
## ['foo.py', 'a', 'b', 'c']
```

as with C, the program name is in `argv[0]`, and the rest follows.

sys.argv

Let's try using that with minmeanmax:

```
import sys

def minmeanmax(items):
    """Returns summary statistics of a sequence - min,mean,max."""
    minval = min(items)
    meanval= sum(items)/len(items)
    maxval = max(items)
    return (minval, meanval, maxval)

if __name__ == "__main__":
    numargs = map(float, sys.argv[1:])
    low, mean, high = minmeanmax(numargs)
    print "low = ", low, "mid = ", mean, "hi = ", high

## C:..> python minmeanmax.py 1 -7 13. 8.2 0 15
## low =  -7.0 mid =  5.03333333333 hi =  15.0
```

argpargse

Let's take a look at the file arguments.py:

```
import argparse

if __name__ == "__main__":
    parser = argparse.ArgumentParser()
    parser.add_argument("ngrid",  help="size of grid", type=int)
    parser.add_argument("pbgd",   help="background pressure", type=
    parser.add_argument("Re",     help="Reynolds Number", type=floa
    parser.add_argument("outfile", help="output file name prefix", 
    parser.add_argument("-t", "--turbmodel", help="turbulence model
    parser.add_argument("-f", "--velfalloff", help="velocity fallof
    parser.add_argument("-x", "--minx", help="lower x limit of grid
    parser.add_argument("-X", "--maxX", help="upper X limit of grid
    parser.add_argument("-v", "--verbose", help="increase output ve

    args = parser.parse_args()
    print args
```

argpargse

Let's take a look at the file `arguments.py`:

```
import argparse

if __name__ == "__main__":
    parser = argparse.ArgumentParser()
    parser.add_argument(...)
    ...

    args = parser.parse_args()
    print args
```

Basic workflow:

- Create an argument parser,
- Add arguments ("-h/–help" is built in),
- Parse the arguments, return the results.

argparse

```
    parser.add_argument("ngrid",
            help="size of grid", type=int)
    parser.add_argument("pbgd",
            help="background pressure", type=float)
    parser.add_argument("outfile",
            help="output file name prefix", type=str)
```

Can add positional, required arguments

- Give them types, conversion is done for you
- Give them help strings - will show up in `--help` output.

argparse

```
    parser.add_argument("-t", "--turbmodel",
        help="turbulence model", type=int, choices=[1,2,3], default=
    parser.add_argument("-f", "--velfalloff",
        help="velocity falloff", type=float, default=2.)
    parser.add_argument("-v", "--verbose",
        help="increase output verbosity", action="store_true")
```

Can add optional arguments, specified by flags

- `-t 2` or `--turbmodel 2` or `--turbmodel=2`
- Can have default values
- Can have restricted set of valid values
- Can just be flags; if present, true otherwise false.

Referencing module contents

Python namespaces, as you've seen, can be hierarchical.

`import [modulename]` brings those objects in to the current environment, under the `[modulename]` namespace.

Some large packages (*eg*, `numpy`, `scipy`, `matplotlib`) have many levels of namespace.

Shortcircuiting namespaces

You can shortcircuit some of this namespace hierarchy, by, for instance:

```
from sys import argv
```

you can save yourself the horror of occasionally typing an additional "`sys.`" by putting argv directly in the current namespace. More recklessly, you could do

```
from sys import *
```

which will put *everything* in sys in your namespace. Which is fine, because you know absolutely everything in that package and know there could be no possible collisions with names you are using in your program, right?

Shortcircuiting namespaces

Medically speaking, two seconds typing the occasional additional `sys.`:

- Increases strength and dexterity in your fingers,
- Is mild areobic activity which builds cardiovascular strength,
- Demonstrates a prudent and professional approach to rules and order.

whereas using `from [module] import *` repeatedly

- Causes increased difficulty sleeping,
- Raises blood pressure dangerously high when spending hours debugging namespace issues,
- Often associated with personality types who just want to watch the world burn.

Module contents with `dir`

Once a module is loaded, you can view its contents with `dir`. This will show:

- Functions
- Data
  - Module-global *mutable* data is of course evil (evil!)
  - "constants" can be very useful: e.g., `__version__ = 1.0.3`, local constants, etc
  - docstrings

Viewing module documentation with `pydoc`

You can view all of the functions, attributes, and doc strings of a module from the commandline with pydoc:

```
C:\...> python -m pydoc .\minmeanmax.py
```

Gives you a Unix-like "manual page" for the module; can also generate HTML.

Writing one or two lines of documentation for each function makes that information available through code inspection, `help()`, and `pydoc`.

Modules have docstrings, too!

Adding a docstring at the beginning of a module will show up in the same places as for a function: `help`, `pydoc`, the code itself, etc.

As with function docstrings, highly recommended.

Reloading while developing

Python is smart enough not to import a module again once it's been imported.

- Re-initialization could break earlier state
- Also, just slow.

But sometimes you'd like to force a re-import. Say, you have an interactive session open doing testing on module code while you're developing a module.

You can effect a partial re-import of the module via:
`reload([modulename])`

There a number of cases where this will not do everything you want, but for sufficiently simple modules will usually work.

Safest is to exit and restart interactive session.

Bytecode compilation

Those .pyc files you've been seeing as you run examples are byte-code "compiled" (really, only slightly more than pre-parsed) source code.

Python will refer to those .pyc files if they are newer than the .py file - saves parsing.

If you're distributing a large module, you can pre-compile it from the command line, as:

```
C:\...> python -c py_compile .\minmeanmax.py
```

Bytecode compilation

This will *not* speed up the **execution** of your code at all.

It can, however, greatly speed initial **startup** of the code for large modules - no parsing.

Compiling with `-O` will strip out asserts, etc.

Exercise: Refactor into modules (10-15 min)

Continue your refactoring of the runscript program:

- Extract routines into one or more modules, which are then imported by the script
- Don't forget module docstrings, `__version__` attributes, anything else you find useful

Packages and `__init__.py`

Modules are single files of code, data.

It doesn't take long before functionality exceeds what can be sensibly managed within one file.

A python *package* is a directory which contains:

- (Presumably) Modules
- (Possibly) Other packages
- A special module, `__init__.py` which:
  - Identifies the directory as a package
  - Contains initialization code that is run at import time.
  - Special variable **all** which lists modules to load

Testing

The importance of Testing

All code that is under development and isn't actively tested is broken in important but unknown ways.

Code that *is* actively tested is broken in *fewer* and *better known* ways.

- And the same difficult bugs don't reoccur, because they're tested for.

These are all properties well worth having, and so code should really be tested.

Python has several test frameworks which make it very easy to write, run, and maintain test cases.

Testing

We'll start to take a look at differential.py:

```
"""Differential operators on uniform grid."""

def dfdx(f, dx):
    """Takes as input a series of values and a grid spacing.
       Returns the 2nd-order central 1st difference, 1-sided at end
    l = len(f)
    y = l*[0]
    for i in range(1,l-2):
        y[i] = (f[i+1]-f[i-1])/dx
    y[0]  =  (f[1]-f[0])/dx
    y[-1] =  (f[-2]-f[-1])/dx
    return y

#....
```

Testing

But let's start looking at a simpler example, simpletest.py:

```python
"""Let's make sure we can square numbers."""

def square(n):
    return n*n
```

and look at unittest for testing it.

unittest

Unittest is a built-in python module for:

- Creating and tearing down any “fixtures” needed for testing
- Creating test cases to test the code
- Arranging those test cases into test suites
- Running those test suites.

unittest

Unittest is a built-in python module for:

- Creating and tearing down any "fixtures" needed for testing
- Creating test cases to test the code
- Arranging those test cases into test suites
- Running those test suites.

unittest

```python
import unittest
import simpletest

class TestSquare(unittest.TestCase):
    def test_four(self):
        self.assertEqual(16, simpletest.square(4))

    def test_negative_one(self):
        self.assertEqual(1, simpletest.square(-1))

    def test_negative_two(self):
        self.assertAlmostEqual(4, simpletest.square(-2),
                               places=4)

if __name__ == '__main__':
    unittest.main()
```

unittest

```
C:..> python testsq.py  -v
test_four (__main__.TestSquare) ... ok
test_negative_one (__main__.TestSquare) ... ok
test_negative_two (__main__.TestSquare) ... ok

-------------------------------------------------------------------
Ran 3 tests in 0.000s

OK
```

`unittest`

A couple things to note:

- Use otherwise unreasonably long function names for tests - that's the information you get if a test passes or fails.
- Tests can go in the module itself — often very handy — but don't need to.
- Tests are declared by subclassing a `unittest.Testcase` class.
- There are command line options - you can pull out individual tests to run, discover tests in a directory to run, etc.

unittest

```
    self.assertEqual(16, simpletest.square(4))
    self.assertEqual(1, simpletest.square(-1))
    self.assertAlmostEqual(4, simpletest.square(-2),
                              places=4)
```

- Tests fail with asserts if quantites are not equal.
- Also “almost equal” - very handy for numerical computation.
  - define number of decimal places (not sig. figs.) must agree to.
  - absolute, not relative error
  - But can always calculate relative error, compare to zero.

`unittest`

- Can also test to ensure that the routine *fails* in the way we expect.
  - `assertRaises`: check to see that it raises an exception
  - How should `square` fail, and when?
- Can do comparison (`assertLessEqual`)
- Compare data structures (`assertListEqual`, `assertTupleEqual`, etc)

unittest

- Can set up and tear down big data structures (or servers, or..) we'll use for several tests.
- `setUp`, `tearDown` methods.
- Let's start testing the differential operator module.

Exercise (20 min)

Pick your favourite routine we've written (say, the integrator), make it a proper module, and write a series of testcases for it.

- Make sure it gets the right answer in a variety of cases which span important functionality.
- If it has corner cases, make sure they are tested.
- Make sure it fails as expected under different circumstances.
  - What happens when it gets invalid inputs?