PWC Python Course - Introduction to Python

Erik Spence

SciNet HPC Consortium

1 December 2014

Welcome to Python!

The slides for this class can be found here:

http://wiki.scinethpc.ca/wiki/index.php/PWC_Python

Feel free to download them and follow along.

Welcome to Python!

This morning's class will cover the following topics:

- Getting started with Python in Eclipse.
- Basic Python data types.
- Compound data types.
- Loops, conditionals.
- Iterators.

The goal of this morning's material is to get everyone up to speed on basic Python programming.

SciNet

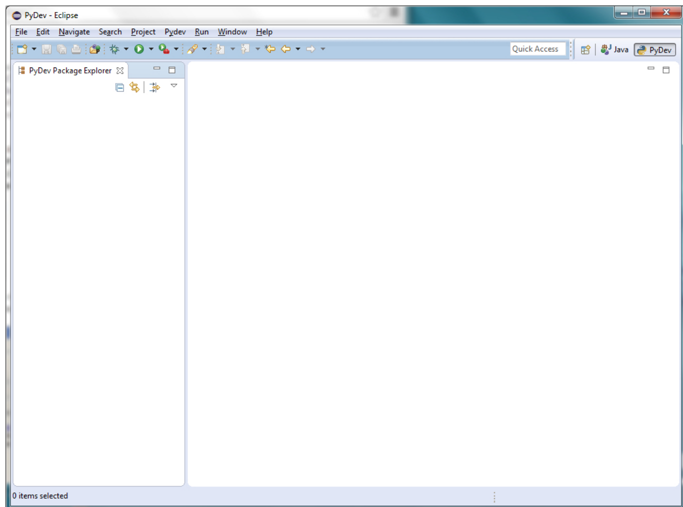
About Python

Some trivia about Python.

- Python combines functional and syntactic aspects of the languages ABC, C, Modula-3, SETL, Lisp, Haskell, Icon, Perl.
- Python is a high-level, interpreted language.
- Python supports many programming paradigms (procedural, object-oriented, functional, imperative).
- Python variables are dynamic, meaning they merely labels for a typed value in memory. They are easily re-assigned to refer to some other memory location.
- Python has automatic memory management, and garbage collection.
- Python is case sensitive.
- Python 3.X is not back-compatible with Python 2.X.

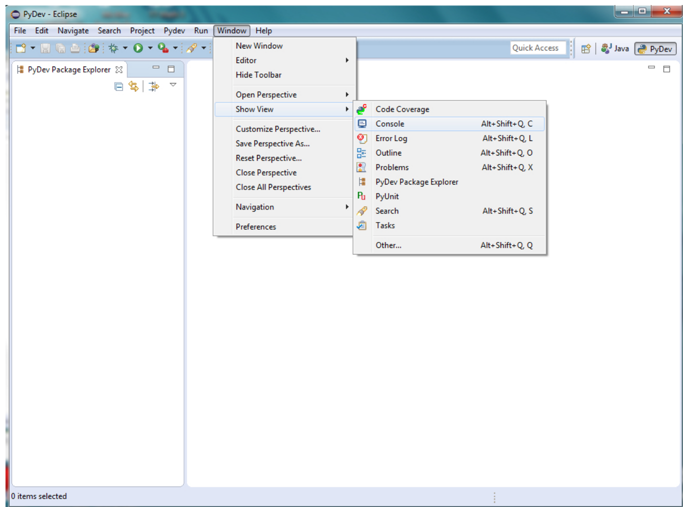
SciNet

Using Python with Eclipse



The first time you open Eclipse, it may look like this.

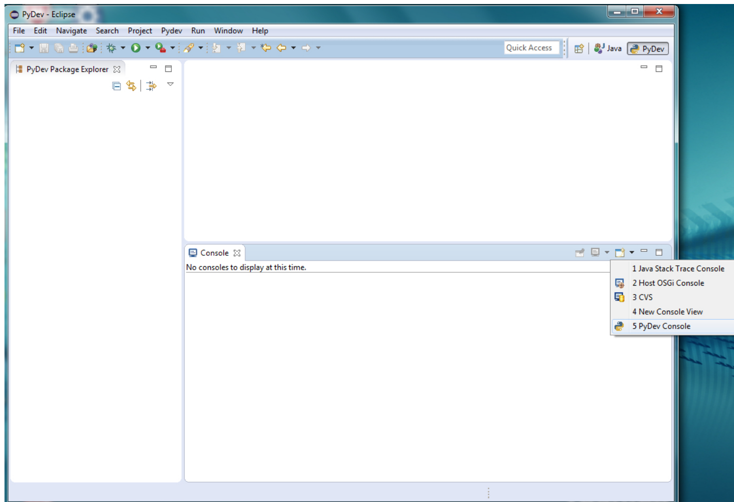
Using Python with Eclipse



We need to open a console view.

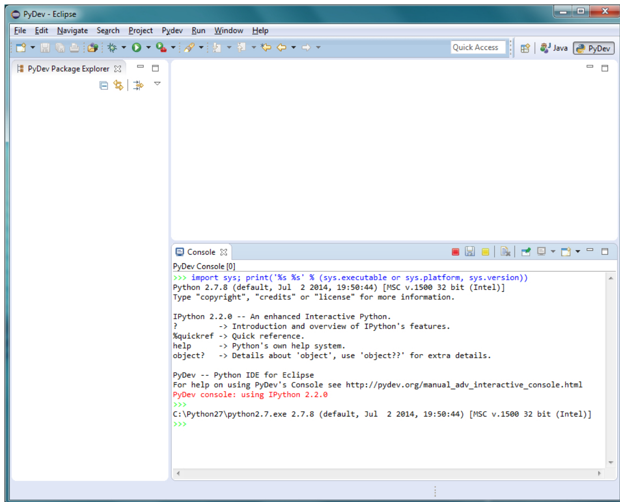
SciNet

Using Python with Eclipse



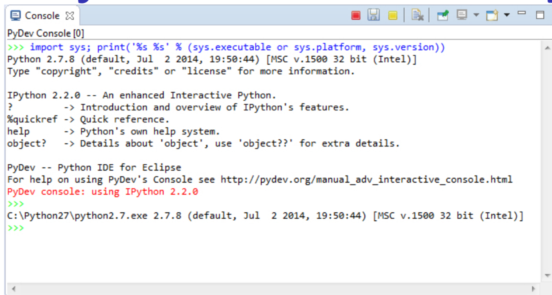
We then open up a PyDev Console.

Using Python with Eclipse



We now have a Python interpreter prompt.

Using the Python interactive interpreter



The interactive Python interpreter allows you to:

- type commands directly into the Python interpreter, to see how they behave, determine the correct syntax and confirm functionality.
- allows rapid code development, since you can quickly test coding ideas, and determine modes of failure.

We'll be solely using the interactive Python interpreter for most of the morning.

Python data types

Python has several standard data types:

- Numbers
- Strings
- Booleans
- Container types
  - Lists
  - Sets
  - Tuples
  - Dictionaries

During this presentation, we are going to cover these data types, as well as iterators.

SciNet

Integers in Python

Python offers two default types of integers:

- "plain integers":
  - All integers are plain by default unless they are too big.
  - These are implemented using long integers in C. This gives them, depending on the system, at least 32 bits of range.
  - The maximum value can be found by checking the sys.maxint value.

```
>>> import sys
>>> print sys.maxint
2147483647
>>> a = 10
>>> type(a)
int
>>> int(10.0)
10
```

SciNet

Integers in Python, continued

Python offers two default types of integers:

- "long integers":
  - Have infinite range.
  - Are invoked using the long(something) function, or by placing an "L" after the number.

```
>>> a = 10
>>> b = 10L
>>> b
10L
>>> type(b)
long
>>> c = long(a)
>>> type(c)
long
```

SciNet

Floats in Python

Python offers two types of floating point numbers:

- "floating point numbers":
  - Based on the C double type.
  - You can specify the exponent by putting "e" in your number.
  - Information about floats on your system can be found in sys.float_info.

```
>>> import sys
>>> print sys.float_info
sys.float_info(max=1.7976931348623157e+308, max_exp=1024,
max_10_exp=308, min=2.2250738585072014e-308, min_exp=-1021,
min_10_exp=-307, dig=15, mant_dig=53,
epsilon=2.220446049250313e-16, radix=2, rounds=1)
>>>
>>> a = 4.5e245
>>> a
4.5e+245
```

Floats in Python, continued

Python offers two types of floating point numbers:

- “complex numbers”:
  - Have a real and imaginary part, both of which are floats.
  - Use z.real and z.imag to access individual parts.

```
>>> a = complex(1.,3.0)
>>> print a
(1+3j)
>>>
>>> b = 1.0 + 2.j
>>> print b.imag
2.0
>>> complex(10.0)
(10+0j)
```

SciNet

Booleans

Python supports standard boolean variables and operations.

```
>>> bool(1)
True
>>> bool(0)
False
>>> bool(3)
True
>>> True + 1
2
>>> False + 1
1
>>>
```

```
>>> a = True
>>> a and False
False
>>> not a
False
>>> a or False
True
>>> a & True
True
>>> a | True
True
```

Strictly speaking, Booleans are a sub-type of plain integers.

Booleans, bitwise operations

Python contains the usual bitwise operations.

Operation	Result
x | y	bitwise OR of x and y
x ^ y	bitwise XOR of x and y
x & y	bitwise AND of x and y
x << n	x shifted left by n bits
x >> n	x shifted right by n bits
~x	x bitwise inverted

SciNet

String manipulation

Strings are delimited by single or double quotation marks (' or "):

```
>>> word = "Hello World"
>>> word
'Hello World'
>>> print word
Hello World
```

The end-of-line character: \n

```
>>> "line 1\nline 2"
'line 1 \nline 2'
>>> print "line 1\nline 2"
line 1
line 2
```

String manipulation, finding characters

Strings come with a number of useful commands built-in:

```
>>> print word
Hello World
>>> print word.count('l')
3
>>> print word.find("W")
1
>>> print word.index("Wo")
1
```

The difference between 'find' and 'index' is when the letters are not found in the string.

What happens when trying to find the position of a letter appearing several times within a string?

SciNet

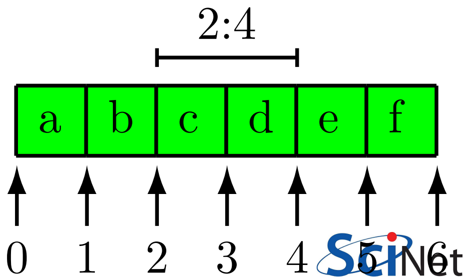
String manipulation, indexing

Selecting characters from within a string:

```
>>> len(word)
11
>>> print word[6]
W
>>> print word[6:7]
W
```

Some notes about indexing in Python:

- Like C++, the first index is 0.
- Read "2:4" as "from the beginning of the second element, to the beginning of the fourth element".



String manipulation, indexing continued

Selecting characters from within a string:

```
>>> print word[:7]
Hello W
>>> print word[7:]
orld
>>> print word[-3:]
rld
>>> print word[:-2]
Hello Wor
```

What is the output of word[-5:-4] and word[-4:-5]?

String manipulation, beginning and ending

Startswith/Endswith:

```
>>> word.startswith("H")
True
>>> word.endswith("d")
True
>>> word.startswith("h")
False
```

Replacing:

```
>>> print word.replace("Hello", "Goodbye")
Goodbye World
>>> print word.replace("l", "?")
He??o Wor?d
>>> print word.replace("l", "?", 2)
He??o World
```

SciNet

String manipulation, upper and lower case

Changing upper and lower case:

```
>>> print word.upper()
HELLO WORLD
>>> print word.lower()
hello world
>>> print word.swapcase()
hELLO wORLD
>>> word.lower().isupper()
False
>>> word.lower().islower()
True
>>> world.upper().isupper()
True
```

The isupper() and islower() commands test the case of the characters in the string.

SciNet

String manipulation, stripping

Strip: to remove characters from both ends of a string:

```
>>> print word.strip("d")
Hello Worl
>>> print "aaaaa Hello Worldaaaaa".strip("a")
 Hello World
>>> print "aaaaa Hello Worldaaaaa".lstrip("a")
 Hello Worldaaaaa
>>> print "aaaaa Hello Worldaaaaa".rstrip("a")
aaaaa Hello World
>>> print "       Hello World       ".strip()
Hello World
```

Using strip(), without an argument, will remove all leading and trailing white space.

String manipulation, testing character types

There are many functions for testing the nature of your string:

```
>>> "3141592".isdigit()
True
>>> "3.141592".isdigit()
False
>>> word.isalpha()
False
>>> "Hello".isalpha()
True
```

- word.isdigit(), are all chars numbers?
- word.isalpha(), are all chars alphabetic?
- word.isalnum(), does it contain digits?
- word.isspace(), does it contain spaces?

SciNet

String manipulation exercise

Quiz: What is a one-line command which determines if a string contains only numbers and a maximum of one dot?

```
>>>
```

String manipulation exercise

Quiz: What is a one-line command which determines if a string contains only numbers and a maximum of one dot?

```
>>>
>>> "3.22143".replace(".", "1", 1).isdigit()
True
>>>
```

SciNet

String manipulation, misc functions

There are too many functions to mention. Here are some more:

```
>>> str(5.31)
'5.31'
>>> print "Hello " + "World" + "!"
Hello World!
>>> word.split(' ')
['Hello', 'World']
>>> word.split('o')
['Hell', ' W', 'rld']
```

The split function splits up a string based on a particular symbol, and returns a list.

SciNet

Lists

The list is a data type not available in a number of lower-level languages:

- A list is a collection of items.
- Each item in the list has an assigned index value.
- Lists are enclosed in square brackets and each item is separated by a comma.
- The items in a list can be of any data type, and mixed types, though as a general rule they are usually all the same type.
- Lists can contain lists.

SciNet

Lists, creation

```
>>> a = [3.123, "Hello World", 3]
>>> print a
[3.123, 'Hello World', 3]
>>> L = ['yellow', 'red', 'blue', 'green', 'black']
>>> print len(L)
5
>>> print L[0]
yellow
>>> print L[1:4]
['red', 'blue', 'green']
>>> print L[2:]
['blue', 'green', 'black']
>>> print L[1:5]
['red', 'blue', 'green', 'black']
>>> print L[1:5:2]
['red', 'green']
>>> print L[-1]
black
```

Lists, manipulation

```
>>> print sorted(L)
['black', 'blue', 'green', 'red', 'yellow']
>>>
>>> L
['yellow', 'red', 'blue', 'green', 'black']
>>> L.sort()
>>> L
['black', 'blue', 'green', 'red', 'yellow']
>>>
>>> L.append('pink')
>>> L
['black', 'blue', 'green', 'red', 'yellow', 'pink']
>>> L.insert(2, 'white')
>>> L
['black', 'blue', 'white', 'green', 'red', 'yellow', 'pink']
```

Lists, removing elements

```
>>> L.remove('green')
>>> L
['black', 'blue', 'white', 'red', 'yellow', 'pink']
>>> tmp = [1, 5, 3, 5]
>>> tmp.remove(5)
>>> tmp
[1, 3, 5]
>>> L.pop()
'pink'
>>> L
['black', 'blue', 'white', 'red', 'yellow']
>>> L.pop(1)
'blue'
>>> L
['black', 'white', 'red', 'yellow']
```

SciNet

Lists, deleting elements

The del command will remove an individual element without returning it.

```
>>> a = [-1, 1, 66.25, 333, 333, 1234.5]
>>> del a[0]
>>> a
[1, 66.25, 333, 333, 1234.5]
>>> del a[2:4]
>>> a
[1, 66.25, 1234.5]
>>> del a[:]
>>> print a
[]
>>> del a
>>> print a
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
NameError: name 'a' is not defined
```

SciNet

Lists, more manipulation

Lists are manipulated with many of the same commands used on strings.

```
>>> L.reverse()
['yellow', 'red', 'white', 'black']
>>> L.count('red')
1
>>> [1, 5, 3, 5].count(5)
2
>>> 'red' in L
True
>>> range(6)
[0, 1, 2, 3, 4, 5]
>>> L + range(3)
['yellow', 'red', 'white', 'black', 0, 1, 2]
```

The range command is often used for indexing loops.

Looping over lists

A common operation is to loop over the elements of a list. Note the colon, and the use of white space in the syntax of the loop.

```
>>> for item in L:
...      print item
...
yellow
red
white
black
>>>
```

Lists of numbers

The range function is commonly used when indices are needed within a loop. It returns a list.

```
>>> for i in range(3):
...     print i
...
0
1
2
>>> for i in range(5,10,2):
...     print i
...
5
7
9
>>>
```

SciNet

List comprehensions

Python allows you to create lists with loops, in what is at first a somewhat strange syntax:

```
>>> S = [x**2 for x in range(10)]
>>> S
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
>>>
>>> t = [x**2 for x in range(10) if x > 5]
>>> t
[36, 49, 64, 81]
```

These are called "list comprehensions". The basic syntax is

[expression(item) for item in list conditional(item)].

SciNet

List comprehensions exercise

Using a list comprehension, create a list whose entries are the first letter of every word in the sentence below.

```
>>> sentence = "Python is easy to learn"
>>>
```

List comprehensions are very powerful, and very fast.

List comprehensions exercise

Using a list comprehension, create a list whose entries are the first letter of every word in the sentence below.

```
>>> sentence = "Python is easy to learn"
>>>
>>> answer = [word[0] for word in sentence.split(' ')]
>>>
>>> print answer
['P', 'i', 'e', 't', 'l']
>>>
```

List comprehensions are very powerful, and very fast.

SciNet

If statements

All code blocks in Python are delineated by white space. The if statement is no exception:

```
>>> for i in range(10):
...       if (i > 4):
...            print i
...
5
6
7
8
9
>>>
```

The brackets in the if statement are optional. The colon is not.

SciNet

If/elif/else statements

Like all if statements, there is an 'elif' (else if) and 'else' option:

```
>>> for i in range(7):
...      if (i > 5):
...           print i
...      elif i < 2:
...           print i - 7
...      else:
...           print 'hello'
...
-7
-6
hello
hello
hello
hello
6
>>>
```

SciNet

Using white space

All code blocks in Python are delineated by white space. But the amount of indentation does not need to be the same from one code block to another. But it must be consistent within the same code block.

```
>>> for i in range(5):
...      if (i > 2):
...           print 'eek'
...      elif i == 2:
...                            print i
...      else:
...       print i - 7
...
-7
-6
2
eek
eek
>>>
```

Using white space, continued

If the code blocks do not line up, you'll get an error message:

```
>>> for i in range(5):
...      if (i > 2):
...           print 'eek'
...      elif i == 2:
...                              print i
...         else:
...          print i - 7
...
  File "<stdin>", line 6
    else:
        ^
IndentationError: unindent does not match any outer indentation
level
>>>
```

While loops

Python actually has only two types of loops, ‘for’ and ‘while’. The while loop behaves in a fashion similar to other languages:

```
>>> i = 3
>>> while (i > 0):
...       print i
...       i -= 1
...
3
2
1
>>>
```

The loop is indented, as expected, and takes a colon, as usual.

SciNet

Loop control statements

Python supports three loop control statements, which modify the normal execution of the loop:

- break: terminates the loop and transfers execution to the statement immediately following the loop, like the traditional break found in C.
- continue: causes the loop to skip the remainder of its body, and then retest the loop condition prior to reiterating.
- pass: is used when a statement is required by syntax, but you do not want any code to execute.

Loop control statements, continued

```
>>> for letter in "Python":
...       if letter == 'h':
...          break
...       print letter
...
P
y
t
>>>
>>> a = 10
>>> if (a > 1):
...   print a
... else:
...   pass
...
10
```

```
>>> for letter in "Python":
...       if letter == 'h':
...          continue
...       print letter
...
P
y
t
o
n
>>>
```

Sets

Sets are similar to lists, but all elements are unique.

```
>>> s = set([1, 3, 2])
>>> s
{1, 2, 3}
>>> s.add(4)
>>> s
{1, 2, 3, 4}
>>>
>>> s.add(2)
>>> s
{1, 2, 3, 4}
>>>
>>> t = {'apple', 'pear', 'apple', 'orange', 'orange'}
>>> t
{'apple', 'orange', 'pear'}
```

Note that sets are automatically sorted.

SciNet

Sets, removing elements

```
>>> s.remove(2)
>>> s
{1, 3, 4}
>>> s.remove(2)
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
KeyError: 2
>>> s.discard(3)
>>> s
{1, 4}
>>> s.discard(3)
>>> s
{1, 4}
>>> s.clear()
>>> s
set()
```

SciNet

Sets, union, intersection

```
>>> RedFruit = set(['strawberry', 'watermelon'])
>>> FruitWithSkin = set(['banana', 'orange', 'watermelon'])
>>> FruitWithSeeds = set(['watermelon', 'orange', 'currant'])
>>>
>>> Fruits = RedFruit | FruitWithSkin | FruitWithSeeds
>>> print Fruits
set(['strawberry', 'watermelon', 'orange', 'currant', 'banana'])
>>>
>>> Fruits - RedFruit
{'orange', 'currant', 'banana'}
>>>
>>> RedFruit & FruitWithSeeds
{'watermelon'}
>>>
```

Tuples

Tuples are useful for representing what other languages call records, related information that belongs together:

- Like lists, a tuple is a sequence of immutable objects.
- Each item has an assigned index value.
- Tuples customarily use parentheses to enclose the elements, though this isn't required.
- Tuples are the default type for comma-separated assignments.
- Tuple elements can include other tuples.
- One-element tuples exist, but need to have a comma in their declaration.
- Empty tuples are also possible.

SciNet

Tuples, creating

```
>>> tup1 = (12, 34.56)
>>> tup2 = "tup2", 36, 21
>>> print tup2
('tup2', 36, 21)
>>>
>>> print tup1[0]
12
>>> print tup2[1:3]
(36, 21)
>>>
>>> tup3 = (95.0,)
>>> tup3
(95.0,)
>>>
>>> tup4 = ()
>>> print tup4
()
```

SciNet

Tuples, index ranges

When you request an index range of a tuple, a tuple is returned.

```
>>> tup3 = tup1 + tup2
>>> print tup3
(12, 34.56, 'tup2', 36, 21)
>>>
>>> print tup2[0]
tup2
>>>
>>> print tup2[0:1]
('tup2',)
>>>
>>> (tup2[0], 52, tup2[2:])
('tup2', 52, (21,))
>>>
>>> (tup2[0],) + (52,) + tup2[2:]
('tup2', 52, 21)
```

SciNet

Tuples, updating

Tuples are immutable, which means you cannot update or change the values of the tuple elements.

```
>>> print tup2
('tup2', 36, 21)
>>>
>>> tup2[1] = 52
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
TypeError: 'tuple' object does not support item assignment
>>>
>>> tmptup = (tup2[0],) + (52,) + tup2[2:]
>>> del tup2
>>> tup2 = tmptup
>>> print tup2
('tup2', 52, 21)
```

Tuples, removing elements

Removing tuple elements is also not possible. The tuple must be rebuilt, as per the previous slide.

```
>>> print tup2
('tup2', 52, 21)
>>>
>>> tmp = tup2[1:]
>>>
>>> del tup2
>>> tup2 = tmp
>>> print tup2
(52, 21)
```

SciNet

Tuples exercise

Loop over the list of products (stored as a list of tuples(name, IsFruit, price), and calculate the sum of the fruit prices.

```
>>> products = [("banana", True, 2), ("potatoe", False, 3),
    ("apple", True, 3), ("pear", True, 4), ("carrot", False, 3)]
>>>
>>> sum = 0
```

Tuples exercise

Loop over the list of products (stored as a list of tuples(name, IsFruit, price), and calculate the sum of the fruit prices.

```
>>> products = [("banana", True, 2), ("potatoe", False, 3),
    ("apple", True, 3), ("pear", True, 4), ("carrot", False, 3)]
>>>
>>> sum = 0
>>> for item in products:
...     if (item[1]):
...         sum += item[2]
...
>>>
>>> print sum
9
>>>
```

Dictionaries

Dictionaries are a Python data type which associates keys to values.

These definitions of the dictionary are all equivalent:

```
>>> a = dict(one = 1, two = 2, three = 3)
>>> b = {'one': 1, 'two': 2, 'three': 3}
>>> c = dict([('two', 2), ('one', 1), ('three', 3)])
>>> d = dict({'three': 3, 'one': 1, 'two': 2})
>>>
>>> e = {}
>>> e['one'] = 1
>>> e['two'] = 2
>>> e['three'] = 3
>>>
>>> e
{'one': 1,  'three': 3, 'two':2}
>>>
```

SciNet

Dictionary key values

Dictionary values have no restrictions. They can be any arbitrary Python object, either standard or user-defined objects.

The same is not true of dictionary keys. These must be strings, numbers or tuples.

```
>>> d = {['three']: 3, 'one': 1, 'two': 2}
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
TypeError: unhashable type: 'list'
>>>
```

More than one entry per key is not allowed.

SciNet

Dictionary key values

If duplicate keys are encountered, the previous value of the key is over-written.

```
>>> d = {'three': 3, 'one': 1, 'two': 2}
>>> d
{'one': 1,  'three': 3, 'two': 2}
>>>
>>> d['three'] = 4
>>> d
{'one': 1,  'three': 4, 'two': 2}
>>>
>>> d['four'] = 4
>>> d
{'four': 4, 'one': 1,  'three': 4, 'two': 2}
```

SciNet

Deleting dictionary keys

```
>>> d
{'four': 4, 'one': 1,  'three': 4, 'two': 2}
>>>
>>> del d['three']
>>> print d
{'four': 4, 'one': 1, 'two': 2}
>>>
>>> d.clear()
>>> d
{}
```

Manipulating dictionary entries

```
>>> MonthNumbers = {'January': 1, 'February': 2, 'March': 3}
>>> MonthNumbers.items()
[('January', 1), ('February', 2), ('March', 3)]
>>>
>>> MonthNumbers.keys()
['January', 'February', 'March']
>>>
>>> MonthNumbers.values()
[1, 2, 3]
>>>
>>> "March" in MonthNumbers
True
>>> "April" in MonthNumbers
False
>>>
```

SciNet

Manipulating dictionary entries

```
>>> MonthNumbers['February']
2
>>> MonthNumbers.get('July')
>>>
>>> MonthNumbers.setdefault('July', -1)
-1
>>> MonthNumbers.setdefault('March', -1)
3
>>>
>>> MonthNumbers
{'February': 2, 'January': 1, 'July': -1, 'March': 3}
>>>
>>> MonthNumbers.get('May', -1)
-1
>>> MonthNumbers.get('March', -1)
3
```

SciNet

Dictionary exercise

Exercise: using a loop, create a dictionary with fruits as keys, and the number of times the fruit is in the list as a value.

```
>>> fruits = {"apple", "pear", "banana", "banana", "pear",
"banana"}
>>> MyDict = {}
>>>
```

Dictionary exercise

Exercise: using a loop, create a dictionary with fruits as keys, and the number of times the fruit is in the list as a value.

```
>>> fruits = ["apple", "pear", "banana", "banana", "pear",
"banana"]
>>> MyDict = {}
>>>
>>> for fruit in fruits:
      MyDict.setdefault(fruit,0)
      MyDict[fruit] += 1
>>>
>>> print MyDict
{'apple': 1, 'banana': 3, 'pear': 2}
>>>
```

Understanding loops

You've probably noticed that you can loop over just about any type of container object:

```
>>> for element in [1, 2, 3]:
...     print element
>>> for element in (1, 2, 3):
...     print element
>>> for key in {'one': 1, 'two': 2}
...     print key
>>> for char in "123":
...     print char
>>>
```

But what's happening under the hood?

SciNet

How 'for' loops work

Behind the scenes, this is what happens:

- the for statement is calling the iter() function on the container object; this returns an iterator object.
- the next() method (or __next__ in Python 3) of the iterator is called, returning the needed value.
- this is repeated until the iterator raises a StopIteration exception.
- once the exception is raised, the for loop ends.

SciNet

Iterables and iterators

In Python, *iterable* and *iterator* have very specific meanings:

Iterable:

- An iterable is an object that has an __iter__ method, which returns an iterator.
- Or an iterable is an object wich defines a __getitem__ method that can take sequential indexes starting from zero.
- An iterable is an object from which you get an iterator.

Iterator:

- An iterator is an object with a next (Python 2) or __next__ (Python 3) method.

If you wish to add iterator behaviour to your classes, the above functions need to be added to your class.

Iterators

A list is iterable because it has the __iter__ method, but it is not an iterator:

```
>>>
>>> L = [1, 2, 3, 4]
>>> L.__iter__
<method-wrapper  '__iter__' of list object at 0x7fbf38e72ea8>
>>>
>>> L.next()
Traceback (most recent call last):
File "<stdin>", line 1 in <module>
AttributeError: 'list' object has not attribute 'next'
>>>
```

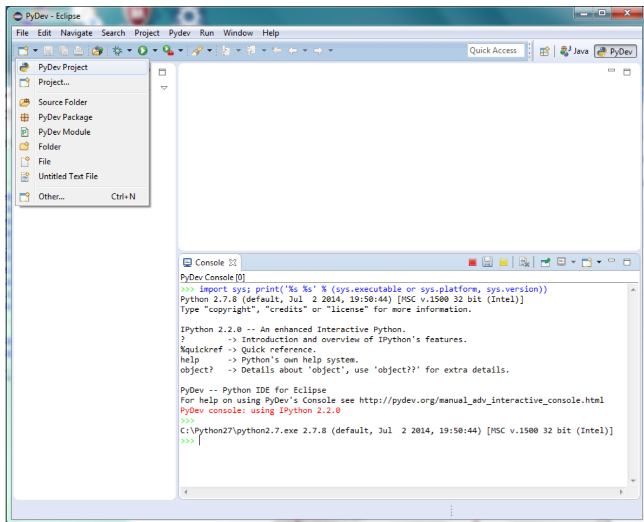
Iterators, continued

When we loop over a list, the list is converted as an iterator:

```
>>> L = [1, 2]
>>> items = iter(L)
>>> items.__iter__
<method-wrapper  '__iter__' of listiterator object at
0x7fbf38e98290>
>>>
>>> items.next
<method-wrapper  'next' of listiterator object at 0x7fbf38e98290>
>>> items.next()
1
>>> items.next()
2
>>> items.next()
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
StopIteration
```

SciNet

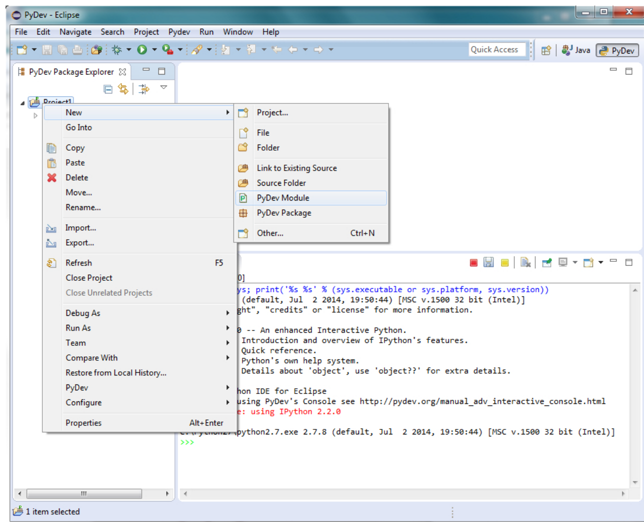
Running scripts within Eclipse



Click on the ‘New’ button, select ‘PyDev Project’.

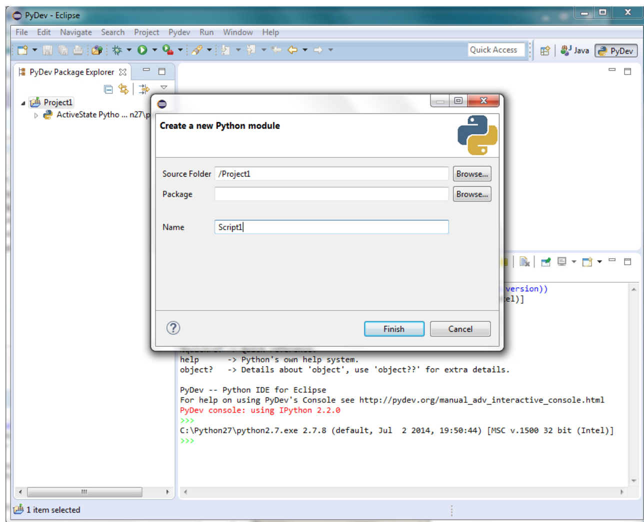
SciNet

Running scripts within Eclipse



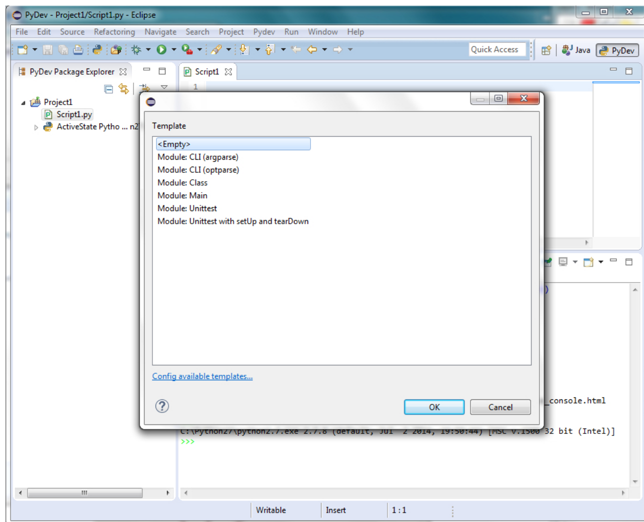
Right click on the project, then select 'New', 'PyDev Module'.

Running scripts within Eclipse



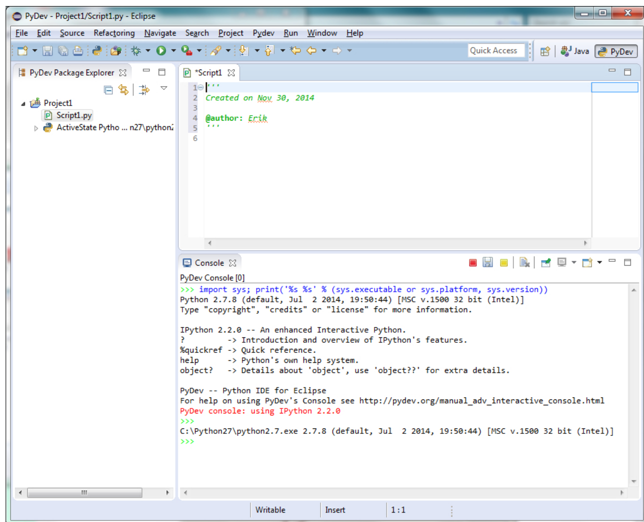
Give your script a name.

Running scripts within Eclipse



Choose an Empty template.

Running scripts within Eclipse



And you're read to go!

Running scripts from within Eclipse

There are a number of ways to run your script from within Eclipse. After you confirm that you have a "Console View" open, and you have saved your code:

- Right-click on the editor tab, and select "Run As", "Python Run".
- If you've run this code previously, press the green right-arrow button on the toolbar.
- Type "%run C:\path\to\Script1.py" from the command prompt.

We'll get more experience running scripts later in the class.

```
>>> %run C:\Users\Erik\workspace\Project1\Script1.py
Hello World!
```

Warning: when I did "%run" the interactive console stopped producing output. If this happens, restart Eclipse.

SciNet

Running scripts from the DOS prompt

Python code can also be executed from the DOS prompt:

- Click the Start menu, and open up the prompt by searching for the 'cmd' command.
- Once you're at the prompt, type the command:

```
C:\Users\Erik>
C:\Users\Erik> cd workspace\Project1
C:\Users\Erik\workspace\Project1> C:\Python27\python.exe Script1.py
Hello World!
C:\Users\Erik\workspace\Project1>
```