Basic Python for Jeppesen Products

covers Python 2.6
used in Jeppesen Products

developed by Jeppesen Training

Dario Lopez-Kästen

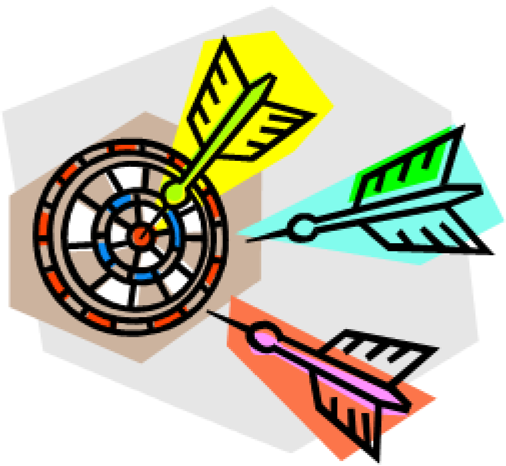
Systems Expert

dario.lopez-kasten@jeppesen.com

@Jeppesen since 2007

CMS2, R21 and Atrium
Python Head Teacher

Course goals



- **Let experienced developers become familiar with the Python Language**
  - Types & Operators
  - Statements, Control Flow & Error Management
  - Writing Object Oriented Code in Python
- **Know how to run Python code in a UNIX environment**
  - In a shell, from Studio
- **Have a solid ground for more advanced courses**
  - PRT I & II, Python in Studio, etc

Prerequisites

- Previous programming knowledge/experience
- Previous experience of **object oriented** programming
- Previous experience with Linux/Unix
  - File system navigation
  - Text editing tools
  - Unix shells

We have a Unix/Linux Quick Reference available - please let the teacher know if you want one!

Please –

Don't be afraid to ask questions if anything is unclear!

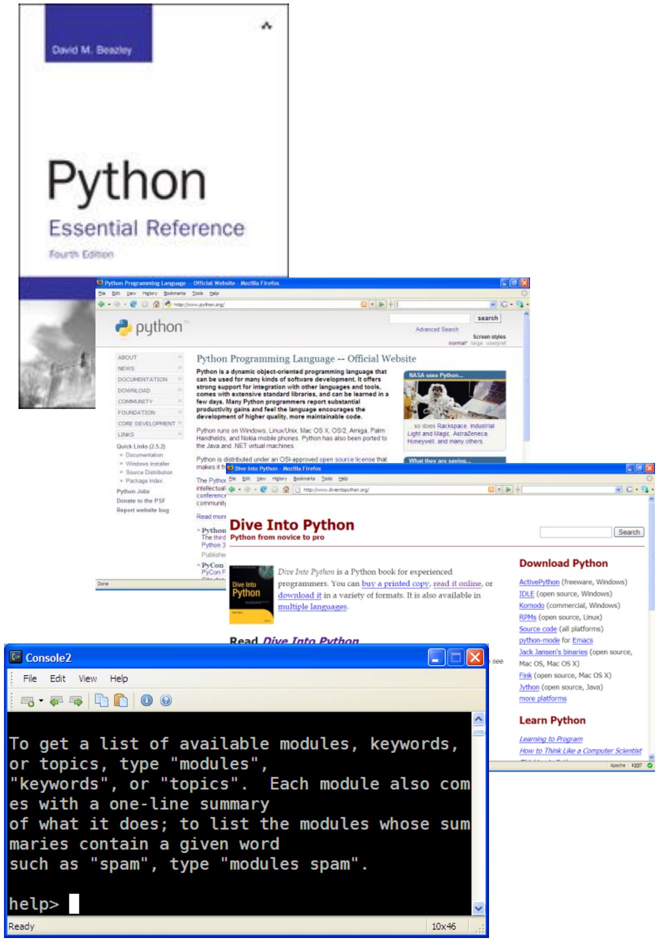
Course material

- **The Course Slides**
  – Quite a lot of them - but not all will be used. Intended to be used as mini-reference.

  – The Course Reference Book
  – Python Essential Reference by David M. Beazley (4th edition).

- **Information on the web**
  – http://www.python.org

- **Built-in help tools**
  – Use the help() command in the Python interpreter shell



Agenda

Day 1

09:00 - 10:15	1. Introduction to Python
10:15 - 10:30	**Coffee break**
10:30 - 12:15	2. Basics of the Python Language
	3. Built-in data types
12:15 - 13:00	**Lunch**
13:00 - 15:00	3. Built-in data types, cont.
	4. Flow control
15:00 - 15:15	**Coffee break**
15:15 - 17:00	5. Functions

All times are approximate – changes may/will occur

Short breaks every ~40 minutes or so

Agenda

Day 2

09:00 - 10:15	Review of day 1 6. Modules & Packages
10:15 - 10:30	**Coffee break**
10:30 - 12:15	7. Object-oriented Python
12:15 - 13:00	**Lunch**
13:00 - 15:00	8. Exceptions 9. Iterators & generators
15:00 - 15:15	**Coffee Break**
15:15 - 17:00	10. Profiling and performance
	11. Testing your code
	12. Some useful built-in modules

All times are approximate – changes may/will occur
Short breaks every ~40 minutes or so

1. Introduction to Python

What is Python?
Where Python Fits In Jeppesen Products
Python Versions
The Python Interpreter
Summary & Exercise 1

What is Python?

– Object-oriented, general-purpose programming language

– Automatic memory management

– Dynamically, implicitly & strongly typed

– Automatically compiles to intermediate byte code

– Interactive interpreter shell, see results immediately

– "Batteries Included" - Huge Standard Library & many 3rd party add-ons

For more detailed, but still high level information about Python's features, visit http://www.python.org/ or http://en.wikipedia.org/wiki/Python_(programming_language).

Where Python fits in

Python is the tool to

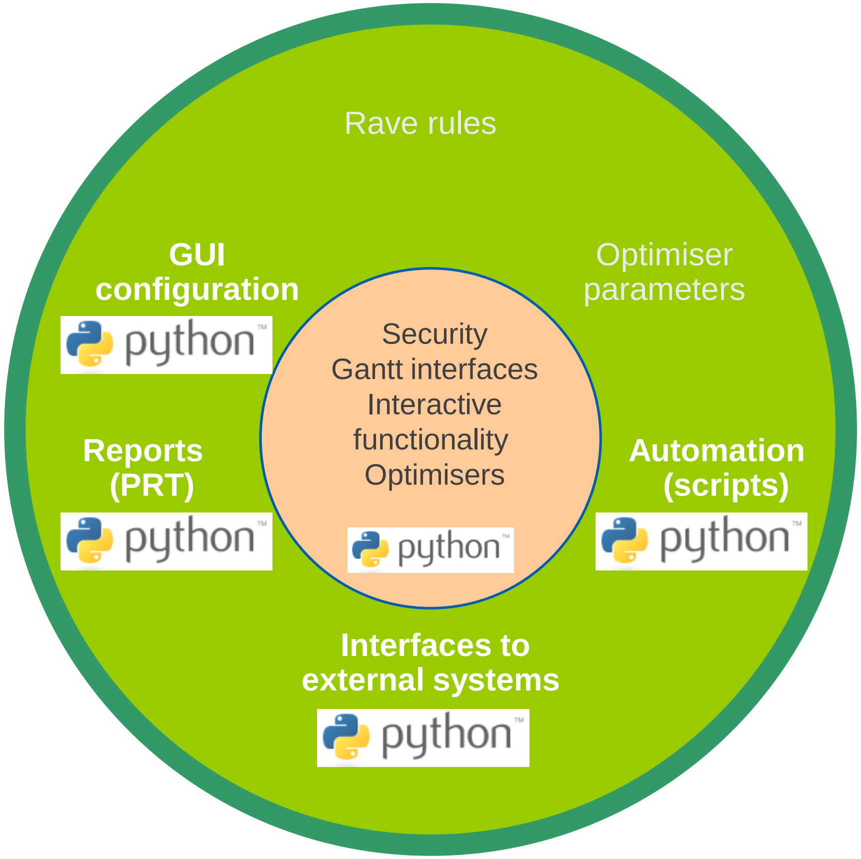
– Author and execute Reports

– Customize, script and automate the behaviour of Jeppesen products

– Programmatically interface with Studio, PRT and other components with APIs for Python.

– Create interfaces to external systems

Jeppesen Products



Python versions

Python 2.x Series

- **2.6** used in current Jeppesen products (Atrium, R21)
- 2.7.10 current stable version, last of v2 series.

NOTE!
You *may* have more than one Python version installed in your system.

Being aware of this avoids confusion

Python 3 – future versions

- 3.5 – current stable version
  - Not used in Jeppesen products (yet)
  - Not fully compatible with 2.6 but features similar to python > 2.6
  - No *major* porting efforts needed

For a complete listing of all previous, current and future releases, visit http://python.org/

The Python interpreter – 1 of 7

The application that executes Python code

– Compiles source code into byte code and regenerates it as necessary (for example when source changes).

– Provides a simple - but useful - shell, embeddable in other applications.

– Normally found as python (or similar, e.g. python2.6) in your $PATH.

The Python interpreter – 2 of 7

Using Python

- From the command line passing a *source file* as argument to Python
- Interactively in the interpreter shell, command by command.
- From the command line, feeding Python commands as *command line arguments.*
- Running a script as an *executable* program, using the Python interpreter as a *shell*.

The Python interpreter – 3 of 7

Interactively, command by command

```
$ python
Python 2.6.6 (r266:84292, Nov 21 2013, 10:50:32)
[GCC 4.4.7 20120313 (Red Hat 4.4.7-4)] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>> s = "Hello world!"
>>> print s
Hello world!
>>> s
'Hello world!'
>>> print 'My first Python Program'
My first Python Program
>>> exit
'Use CTRL-D (i.e. EOF) to exit.'
>>> quit()
$
```

The print statement outputs to standard out

Using a variable as a "statement" accesses its value

This is how you exit the interpreter on Un*x/Linux (varies depending on OS/platform).

The Python interpreter – 4 of 7

Interactively, executing a source file from *within the interpreter*

```
$ python
...
>>> execfile("Ex2.py")
Hello world!
>>> print s
Hello world!
>>> quit()
$
```

Contents of file "Ex2.py":

```python
# sample code
s = "Hello World!"
print s
```

execfile is a *built-in-function* that executes all code in the file "`Ex2.py`" *as if you had typed it directly into the interpreter.*

Defined variables are available in the interpreter after the call to **`execfile`** has finished

You will find all the built-in functions in the Course Book, pp 201-211.

The Python interpreter – 5 of 7

Passing the source file to python on the OS shell

```
[user@host]$ python Ex2.py
Hello world!
[user@host]$
```

Contents of file "`Ex2.py`":

```
# sample code
s = "Hello World!"
print s
```

On the *shell* command line, we pass the file "`Ex2.py`" as an *argument* to the python interpreter.

Python will then read the source file "`Ex2.py`" and execute all the code in it, line by line as if you had typed each line manually into the interpreter.

The Python interpreter – 6 of 7

Passing commands to python on the OS shell

On csh, tcsh and similar shells

```
user@host> python -c 's = "Hello world!" \
? print s'
Hello world!
user@host>
```

On sh, bash and similar shells

```
[user@host]$ python -c 's = "Hello world!"
> print s'
Hello world!
[user@host]$
```

Look at the man page for python(1) for information about all supported flags and arguments; in your shell, issue the following command: `man python`

The Python interpreter – 7 of 7

Running a script as an executable, using the interpreter as a *shell*

```
[user@host]$ ls -l Ex3.py
...
-rw-r--r--  1 user users 49 Aug 16 08:34 Ex3.py
......
[user@host]$ chmod u+x Ex3.py
[user@host]$ ls -l Ex3.py
...
-rwxr--r--  1 user users 49 Aug 16 08:34 Ex3.py
...
[user@host]$ ./Ex3.py
Hello world!
[user@host]$
```

Contents of file "`Ex3.py`":

```
#!/bin/env python
# sample code
s = "Hello World!"
print s
```

The line

`#!/bin/env python`

is used by the unix shell to find a python in the $PATH instead of hard coding a specific python installation location

We also need to make our script ***executable*** – the (**"x"**)**-** so that we can use it as a program.

Summary

About Python

- Python is an easy to learn, high level and garbage collected OO Programming language with implicit and strong typing.

- The Python interpreter provides an interactive shell; also compiles source code into byte code and executes it.

- Python is used in Jeppesen Products to automate tasks, customize GUI and back-end behaviour and to author and run reports.

Exercise 1

Review of Exercise 1

2. Basics of the Python Language

Python syntax
Statements & expressions
Objects & references
Code standards
Summary

Python syntax – 1 of 4

Python programs consist of statements

– *Simple* statements span one *logical* line
– *Compound* statements consist of several simple statements in a *block*

A *logical* line may span several *physical* lines

– Using the continuation character `\` (backslash)
– Inside `"""   """`, `'''   '''`, `[ ]`, `{ }` and `( )`
– **Note:** Physical lines end with a *newline* char, `(\n)`

Blocks are specified by *indentation*

– *Whitespace* (tabs and spaces) is the indentation marker
  - Python standard is to use ***four* (4) space characters per indentation level**
  - Python best practice and ad-hoc standard
– Statements that expect an indentation level end with a colon `(:)`
– *Never mix spaces and tabs* – leads to hard to find syntax & logical errors

Python syntax – 2 of 4

Comments

- Start with a `#` character
- Are always *single line* - no block comment construct.

Identifiers/variables

- Identifiers in Python can be of any length
- Case matters: `MyVar != myvAR`
- Identifier names must start with a *letter* or an *underscore*
- Valid characters for identifier names are: `a-z A-Z 0-9 _`

Regex: `(A..Z,a..z,_)(A..Z,a..z,0..9,_)*`

Python syntax – 3 of 4

By default, Python uses 7-bit US-ASCII for program text

- In Python 2.6, it is an *error* to use an non-ASCII characters in source code without specifying the encoding:

```
SyntaxError: Non-ASCII character '\xe5' in file notencoded.py on line 1, but no
encoding declared; see http://www.python.org/peps/pep-0263.html for details
```

Telling Python to use a proper encoding

- To support, for example, ISO-8859-15 (Latin1 with the € sign) Put the following as the *first* or *second* line in your program:

```
# -*- coding: iso-8859-15 -*-
```

- Python will understand this comment to mean "this source code uses this encoding"

Avoiding problems in Python 2 source code

- Always specify the encoding - never assume a particular encoding
- *Avoid using Unicode in your program text* – at least for Python 2.x

Python syntax – 4 of 4

Examples

```
#!/bin/env python
# -*- coding: iso-8859-15 -*-
print "Hello Göteborg!"
a_really_long_variable_name = "A 0.01€ for your thoughts"
```

```
# An if statement is a compound
# statement
if a < 0:
    print a   # Simple statement
    a = a + 1 # Another one
else:
    print "The value of a+1 " \
       "is now:", a + 1
```

```
# A multi-line logical line
my_list = [
    'Hello',
    'How',
    'are',
    'you',
    'doing',
    '?']
```

Statements & expressions

Expressions return *objects*

- Expressions *always* have a return value in the form of an object
- The *type* of the object depends on the objects involved

Statements perform actions – no return value

- You get a *Syntax Error* when trying to use a statement as an expression

```
# Example expressions
a == b     # return boolean object
a < 10     # return boolean object
dir(a)     # return list object
a + b      # Object type depends
           # on a and b

# Example statements
print a      # prints a value
pass         # the no-op statement
import os    # importing a lib
a = 10

# Errors – statements are not
# expressions -> Syntax errors
a = print 10
b = pass
c = import os
a = (b = 10)
```

Objects & references – 1 of 3

Python objects

- Basic types are built in: *number, string, sequences, mapping objects*
- Objects are created when their program text is executed (top-down)
- New object types are created with *Classes*

References to objects

- May be *variables* or entries in a *container* object
- Each object has a *reference counter*, keeping track of how many times is it being referenced/used
- When an object has no more references, Python detects this and the object is deleted automatically (via *garbage collection*).

Objects & references – 2 of 3

```
>>> import sys
>>> a = []
>>> b = a
>>> l = [a,a]
>>> print sys.getrefcount(a)
5
>>> del l
>>> b = 10
>>> print sys.getrefcount(a)
2
```

- `sys` is a Python *module* (library) (built-in) that provides access to Python System-level runtime properties and features.
- `getrefcount` is a function that returns the number of references there is to a certain object.
- `sys.getrefcount` is how you address the function in the module.

Objects & references – 3 of 3

Assignment Statements

– Python has *plain* assignment and *augmented* assignment

Plain assignment operator, =

– Creates a variable and binds it to an object, or rebinds an existing variable to reference a new object:

```
a = 1      b = SomeObj      x = sin(y)-3      a = 'Hi'
```

Augmented assignment operators

```
+=   -=   *=   /=   //=   %=   **=   |=   >>=   <<=   &=   ^=
```

– *Never creates its target reference* – identifier binding must already exist.

– For mutable objects, augmented assignment *may change the object in place*, rather than create a new object.

Code standards – 1 of 2

– Jeppesen follows *PEP-8 Style guide for Python Code*

http://www.python.org/dev/peps/pep-0008/

The Code Standard is a *Best Practice* followed at Jeppesen

Jeppesen Naming Standard

```
my_variable  # variables

MyClass      # classes

my_function  # functions
             # and methods

my_module    # modules

my_script    # scripts and
             # PRT reports

_name        # private attributes
             # in modules

__name       # private attributes
             # in classes

__name__     # predefined by
             # Python
```

Code standards – 2 of 2

Documenting your code

- A ***docstring*** is a string literal that occurs as the first statement in a module, function, class, or method definition.
- Used by the `help()` command and other Python documentation tools.
- External tools such as *Sphinx* are possible options for utilizing/extracting docstrings to document your code.

For more information on Sphinx, please see http://sphinx-doc.org/

- Proper commenting and documentation of code is important
- **Code is *read* more often than it is written**
- **Maintainability is important** - there is a reality *after* the project too – maintenance, change requests, bug fixing, etc

Summary

Python Syntax

– Valid chars, identifier length, indentation, comments, charsets...

Statements vs. Expressions

– Expressions return objects, statements perform actions

Objects & Their References

– Everything is an object, built-in object types, variables hold references to objects, reference counting, assignments

Code Standards

– PEP-8, Naming conventions, documenting...

3. Built-in data types and their operators

Numbers & Booleans
Sequences & Containers
Strings
Tuples & Lists
Dictionaries & Sets
Files
The None Type
Booleans & Boolean Operators
Mutability of types

Numbers & booleans – 1 of 7

4 data types for numbers

- Actually 5
  - *Boolean* is a number type too ☺ (integer)
- *Plain* Integers (32 / 64 bit)
  - Maximum value of integers depends on architecture. Use `sys.maxint` to find out
- *Long* Integers (unlimited range)
- Float (double in C)
- Complex Numbers (two doubles in C)
- Numbers are *Immutable* → a number object *can not be changed*

Simplest

Most complex

Numbers & booleans – 2 of 7

Notation for Numbers & Booleans

- Plain integers (class `int`)
  ```
  123   -1111   0   0177   0xff12
  ```
- Long integers (class `long`)
  ```
  987654321000L   -1L   -1111111111123l
  ```
- Float (class `float`)
  ```
  1.23   4.0e-10   6.76E100   1E5   1.0
  ```
- Complex numbers (class `complex`)
  ```
  4j+1   3.3j   -77J
  ```
- Booleans (class `bool`)
  ```
  True   False   1   0
  ```

Numbers & booleans – 3 of 7

Numerical operators

- The expected set of operators are available:
  `a**b   a*b   a/b   a//b   a%b   a+b   a-b   +a   -a`

- As well as bitwise operations on integers
  `~a   a<<b   a>>b   a&b   a^b a|b`

- Boolean operators
  `not a     a and b     a or b`

Note:

- `a**b` means a^b, *"a raised to the power of b“*
- `a^b` means "*bitwise a XOR b*"

Numbers & booleans – 4 of 7

General math functions

– Many math functions are available as built-ins.

– Modules ***`math`*** and ***`cmath`*** (***`cmath`*** supports complex numbers) provide access to the mathematical functions defined by the ***C standard***.

Random numbers

– Module ***`random`*** implements *pseudo-random* number generators for various distributions.

Numbers & booleans – 5 of 7

Decimal numbers

- Module ***decimal*** implements decimal floating point arithmetic.
- Offers several advantages over the ***float*** datatype
- For instance, it performs correct *decimal calculations* that are not susceptible to accuracy problems.

Why Decimal is Good

Normal, expected operation:
0.1 + 0.1 + 0.1 – 0.3 = 0

```
>>> from decimal import Decimal
>>> ai = 0.1
>>> bi = 0.3
>>> ai + ai + ai - bi
5.5511151231257827e-017
>>>
>>> ad = Decimal('0.1')
>>> bd = Decimal('0.3')
>>> ad + ad + ad - bd
Decimal("0.0")
>>>
>>> float(bd)
0.29999999999999999
```

More on accuracy problems: http://en.wikipedia.org/wiki/Floating_point#Accuracy_problems

Numbers & booleans – 6 of 7

Type casting

Use the desired type's *class* to convert between number types.

```
>>> int(c)
Traceback (most recent call last):
...
TypeError: can't convert complex to
int; use int(abs(z))
>>> int(abs(c))
3
>>> float(c)
Traceback (most recent call last):
...
TypeError: can't convert complex to
float; use abs(z)
>>> abs(c)
3.3615472627943226
```

```
# integer
>>> i = 10
>>> long(i)
10L
>>> complex(i)
(10+0j)
# float
>>> f = 1.3
>>> int(f)
1
# complex
>>> c = 1.3+3.1J
>>> c
(1.3+3.1000000000000001j)
```

Numbers & booleans – 7 of 7

Mixing numerical types in expressions

Within the scope of the expression:

1. all the objects are *cast* into the *most complex* type involved
2. the calculation takes place
3. the *resulting* type is the *most complex type* involved.

Examples

```
>>> 2 + 2
4
>>> 2 + 2L
4L
>>> 2 + 2.0
4.0
>>> 2 + 2j
(2+2j)
>>> 2 + 2.0j
(2+2j)
>>> True + True
2
>>> 9 / 2
4
>>> 9 / 2.0
4.5
```

Exercise 2.1

~5-10 mins

Sequences & containers – 1 of 5

Container objects

- Objects that can *contain* other objects.
- Containers are *iterable* objects ("objects upon which you can loop").

Sequences - definition

- Containers of an *ordered* sequence of items, indexed by non-negative *integers*.
- Items are accessible by indexing or *slicing.*

Sequence objects

- The built-in types *strings*, *tuples* and *lists* are sequence objects.
- Other types of sequences available from libraries and modules.
- Sequences have some operations in common.

Sequences & containers – 2 of 5

Generic operations on sequences

- Get the length of a sequence `S`

  `len(S)`

- Min and max - *only meaningful if items are of the same type*

  `max(S)`

  `min(S)`

- Sum the items - for supported data types

  `sum(S)`

- Concatenate (not add)

  `S1 + S2`

Sequences & containers – 3 of 5

Generic operations on sequences, cont...

- Repetition

  `S * 5`

- Membership test

  `x in S`

  `x not in S`

- Index/access items; the `i`$_{th}$ element in `S`

  `S[i]`

- To slice - access *segments* or *slices* of a sequence

  `S[i:j]`

Sequences & containers – 4 of 5

Sequence slicing rules

`S[x:y]`

start at item `x`, up to – *but not including* - `item y`

`S[x:]`

the whole sequence, starting at item `x`

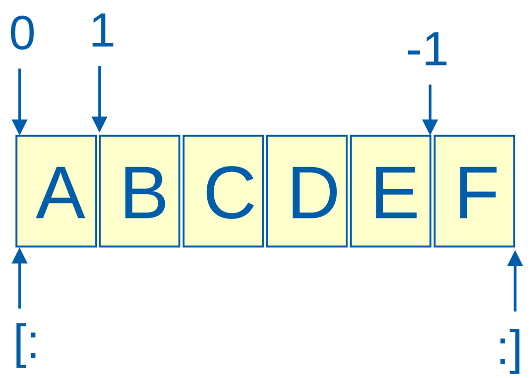
`S[:]`

the whole sequence

`S[:y]`

the whole sequence, up to – *but not including* - item y

Intervals



Sequences & containers – 5 of 5

Sequence slicing rules , cont...

`S[:-y]`

the whole sequence, up to – *but not including* – item y (counting backwards)

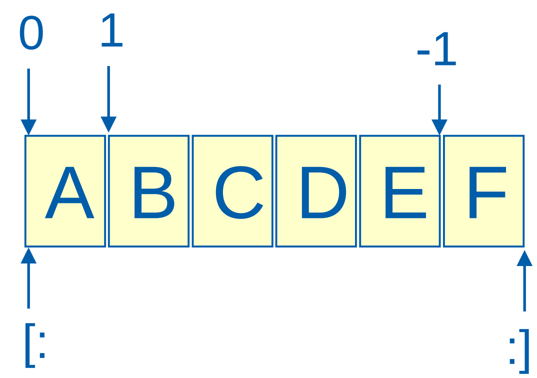
`S[-x]`

get the item found when counting backwards x steps

`S[x:y:z]`

start at item x, up to – *but not including* - item y, fetching every z:th item

Intervals



Strings – 1 of 8

Strings are sequence objects used to store textual data

- Strings are *immutable* – they cannot be changed (*mutated*).
- The *items* of a string (the characters) are strings too, of length 1.
- *Slices* of a string are strings too.
- Python support for unicode strings is built in, notation is

```
u"this is a unicode string"
"this is a normal non-unicode string"
```

- The class names are `str` and `unicode` respectively.
- String objects provide several *methods to operate on themselves* (change case, etc) – these methods return *new* string objects.

Strings – 2 of 8

String literals

Single line strings

```
# Single quotes
'Hello world!'
'I\'m a "virtual" personality!'

# Double quotes
"Hello world!"
"I'm a \"virtual\" personality!"

# Implicit concatenation
'Hello ' 'World!'
"I'm a " '"virtual"' ' personality!'
```

Unicode strings

```
u'Hello World!'
u"I'm a \"virtual\" personality!"
```

Empty strings

```
u''
"“
```

Multi line strings

```
# Single quotes
'''Hello world!

I'm a "virtual" personality!
'''

# Double quotes
"""Hello world!

I'm a "virtual" personality!
"""

# Making it hard
'''Confusing quote 'trick\''''

"""Confusing quote "trick\""""
```

Strings – 3 of 8

String operations

- Concatenation (*slow* for strings!)

```
>>> "hello" + "world"
'helloworld'
```

- Repetition

```
>>> ':-)' * 5
':-):-):-):-):-)'
```

- Indexing

```
>>> 'helloworld'[5]
'w'
```

- Slicing/subscription

```
>>> 'helloworld'[3:5]
'lo'
```

More String operations

- Membership check

```
>>> "hello" in "hello world!"
True
>>> "globe" in "hello world!"
False
```

- Length

```
>>> len('hello')
5
```

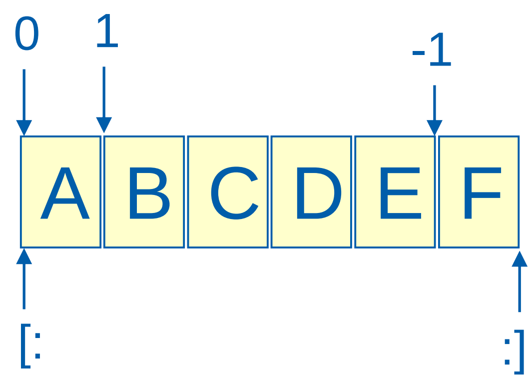
- Max/min values

```
>>> max('helloworld')
'w'
>>> min('helloworld')
'd'
```

Strings – 4 of 8

More examples on slicing strings

```
>>> s = "ABCDEF"
>>> print s[1]
B
>>> print s[-1]
F
>>> print s[0:1]
A
>>> "BC" in s
True
>>> print s[:-1]
ABCDE
>>> s[2:5]
'CDE'
```



Strings – 5 of 8

String formatting/interpolation

- Unique to strings, feature is similar to text format using `sprintf()` in C
- Two ways, standard (positional) and mapping
- `%s` (string formatter) can be used for all data types (like `%D` in PDL) – implicit conversion to `str`

Standard

- The `%` operator is used, with a modifier describing the data type

```
>>> "%s %05d %s" % (44, 43, 'are numbers')
'44 00043 are numbers'
```

Mapping

- Map keywords to a *mapping* object - order does not matter

```
>>> "%(foo)s %(bar)05d %(txt)s" % {'txt':'are numbers', 'foo':43, 'bar':44}
'43 00044 are numbers'
```

Python 2.6 and later

```
>>> "{foo} {bar} {txt}".format(**{'txt': 'are numbers', 'foo':43, 'bar': 44})
'43 44 are numbers'
>>> "{foo} {bar} {txt}".format(txt='are numbers', 'foo'=43, 'bar'=44)
'43 44 are numbers'
```

Strings – 6 of 8

String objects have many useful methods

- Used to perform conversions, find substrings, etc
- Available directly on string objects, no need to import specific libraries to operate on strings
- Methods that "change" strings, actually *return new string objects* – remember: strings are *immutable*

Examples

```
>>> s = "Hello there"
>>> s.upper()
'HELLO THERE'
>>> s
'Hello there'

>>> s.index("el")
1

>>> l = s.split(" ")
>>> print l
['Hello','there']

>>> "-*-".join(l)
"Hello-*-there"
```

Strings – 7 of 8

Caveats for strings

- String objects are *immutable* – this means that a string object cannot be changed.

- Concatenation does not *change* any of the string objects; *new* string objects are created – and the old ones *remain* until they are garbage collected.

- Do this a lot – eg. in a loop – and you will end up consuming your available memory.

```
# The good way
>>> l = ['hello', 'lovely', 'world']
>>> s = ' '.join(l)
>>> s
'hello lovely world'

# The bad way, consumes memory – slow!
>>> s = ''
>>> for w in l: s = s + ' ' + w
>>> s
'hello lovely world'

# The good way - faster
>>> a = 'Two'
>>> b = 'One %s Three'%(a)
>>> b
'One Two Three'

# The bad way - slower
>>> b = 'One
>>> b = b + ' ' + a + ' ' + 'Three'
>>> b
'One Two Three'
```

Strings – 8 of 8

Caveats for strings, cont...

- Use the `join` method of strings
- Avoid concatenation to create new strings
- Use *string interpolation* for templates.

```
# The good way
>>> l = ['hello', 'lovely', 'world']
>>> s = ' '.join(l)
>>> s
'hello lovely world'

# The bad way, consumes memory – slow!
>>> s = ''
>>> for w in l: s = s + ' ' + w
>>> s
'hello lovely world'

# The good way - faster
>>> a = 'Two'
>>> b = 'One %s Three'%(a)
>>> b
'One Two Three'

# The bad way - slower
>>> b = 'One
>>> b = b + ' ' + a + ' ' + 'Three'
>>> b
'One Two Three'
```

Exercise 2.2

~5-10 mins

Tuples – 1 of 3

Tuples are sequence objects, used to store an *ordered* sequence of arbitrary objects

- Tuples are *immutable* – like strings – but individual *items* in tuples may be mutable.
- *Slices* of a tuple are tuples too.
- Like strings, tuples support all standard sequence operations: indexing, slicing, concatenation, membership checks, etc.
- Class name is `tuple`.
- Unlike strings, tuples do not have their own methods to operate on themselves.

Tuples – 2 of 3

Tuple literals

Creating tuples

```
>>> t = (1,'foo', 42, 'bar')
>>> type(t)
<type 'tuple'>

>>> t[0:2]
(1, 'foo')

>>> t[3]
'bar'

>>> a = () # Empty tuple
>>> type(a)
<type 'tuple'>
```

Tuples with one element

```
>>> t = (1)
>>> type(t)
<type 'int'> # did not work!

>>> t = (1,)
>>> type(t)
<type 'tuple'>

>>> t = 1,
>>> type(t)
<type 'tuple'>
```

Tuples – 3 of 3

Tuple operations

– Concatenation
```
>>> (1,2) + ('a', '123')
(1, 2, 'a', '123')
```

– Repetition
```
>>> (1,2) * 5
(1, 2, 1, 2, 1, 2, 1, 2, 1, 2)
```

– Indexing
```
>>> (1, 2, 'a', '123')[2]
'a'
```

– Slicing/subscription
```
>>> (1, 2, 'a', '123')[1:3]
(2, 'a')
```

– Membership check
```
>>> 'a' in (1, 2, 'a', '123')
True
>>> 6 in  (1, 2, 'a', '123')
False
```

More tuple operations

– Length
```
>>> len((1, 2, 'a', '123'))
4
```

– Max/min values
```
>>> max((1, 2, 'a', '123'))
'a'
>>> min((1, 2, 'a', '123'))
1
```

– **Beware**!
```
>>> min((1, 2, 'a',
         '123', 1j+0.3))
Traceback (most recent call last):
  File "<interactive input>", line
1, in ?
TypeError: no ordering relation is
defined for complex numbers
```

Exercise 2.3

~5-10 mins

Lists – 1 of 6

Lists are sequence objects, used to store a sequence of arbitrary objects

- Lists are *mutable* – unlike strings and tuples
- Slices of a list are lists too
- Like *tuples*, lists support all standard sequence operations
- Class name is `list`
- Like *strings*, lists have their own methods to operate on themselves. *New methods and operators* are available because of the mutability.

Lists – 2 of 6

List literals and initial examples

Creating Lists

```
>>> l = [1,'foo', 42, 'bar']
>>> type(l)
<type 'list'>

>>> l[0:2]
[1, 'foo']

>>> l[3]
'bar'

>>> a = [] # Empty list
>>> type(a)
<type 'list'>
```

Type casting

```
>>> t = (1,2,3,4)
>>> l2 = list(t)
>>> l2
[1, 2, 3, 4]

>>> list('hello')
['h','e','l','l','o']
```

Lists – 3 of 6

List operations

- Concatenation
```
>>> [1,2] + ['a', '123']
[1, 2, 'a', '123']
```
- Repetition
```
>>> [1,2] * 5
[1, 2, 1, 2, 1, 2, 1, 2, 1, 2]
```
- Indexing
```
>>> [1, 2, 'a', '123'][2]
'a'
```
- Slicing/subscription
```
>>> [1, 2, 'a', '123'][1:3]
[2, 'a']
```
- Membership check
```
>>> 'a' in [1, 2, 'a', '123']
True
>>> 6 in  [1, 2, 'a', '123']
False
```
- Length
```
>>> len([1, 2, 'a', '123'])
4
```
- Max/min values
```
>>> max([1, 2, 'a', '123'])
'a'
>>> min([1, 2, 'a', '123'])
1
```

Beware!
```
>>> min([1, 2, 'a',
          '123', 1j+0.3])
Traceback (most recent call last):
  File "<interactive input>", line 1, in ?
TypeError: no ordering relation is defined
for complex numbers
```

Lists – 4 of 6

Mutating operations

– Replace a slice with objects from another sequence to replace, insert and remove objects

```
# L1[x:y] = L2
>>> l1 = [1,2]
>>> l2 = [3,4]
>>> l1[1:1] = l2 # Insert
>>> print l1
[1, 3, 4, 2]
>>> l2[:] = [10] # Replace
>>> print l2
[10]
```

```
>>> l1[0:2] = l2 # Replace
>>> print l1
[10, 4, 2]
>>> l1[:-1] = [] # Remove
>>> print l1
[2]
```

Lists – 5 of 6

Mutating methods

- `l.append(x)`
  same as `l += [x]`

  (but use append!)

- `l.insert(i, x)`
  same as `l[i:i] = [x]`

- `l.sort()`
- `l.sort(f, k, r)`
  sort `l` , optionally using functions `f` and `k`

  Note: this is potentially ***very*** slow. Using only `l.sort()` will use Pythons built-in type comparison which is faster.

`l.reverse()`
reverses the items of `l`

Non-mutating methods

- `l.index(x)`
  return (first) position of object `x` in list `l`

- `l.count(x)`
  number of occurrences of `x` in `l`

Hint: The built in function **sorted** creates a sorted list from any iterable object.

Lists – 6 of 6

Augmented assignment and lists

Adding all objects in one list to the end of a list

`l1 += l2`

Delete objects from a list with del

`del l[a:b]` is the same as `l[a:b]=[]`

`del l[a]` removes item a

Exercise 2.4

~5-10 mins

Dictionaries – 1 of 5

Dictionaries

- Mutable *container* objects - contain arbitrary collections of *unordered* key-value pairs
- Dictionaries belong to the family of *mapping* objects – these *map* keys *to* values.
- All hashable (*immutable*) data types can be used as keys - values can be objects of *any* type, mutable or not.
- Contained objects are *indexed* by the *keys*. Dictionaries have fast lookup – constant time.
- Dictionaries are *not sequences*, so none of the sequence operations can be applied to them.
- Class name is `dict`.

Dictionaries – 2 of 5

Creating Dictionaries

```
d = {"key1": 1234,
     list : "a data type",
     len  : "a function"}
```

Type Casting

```
>>> d2 = dict([(1,2),(3,4)])
>>> d2
{1:2, 3:4}
```

Any *iterable* of pairs

Dictionaries – 3 of 5

Non-mutating methods

- Get the value of one key
  `d[key] or d.get(key, default)`

- Get the number of objects in the dictionary
  `len(d)`

Get a list of all keys
  `d.keys()`

Get a list of all entries/items
  `d.items()      # returns key,value pairs`

Check if a key exists
  `d.has_key(key)` or: `key in d`

Dictionaries – 4 of 5

Mutating methods

- **Add or replace an object**
  ```
  d[key] = x
  ```
- **Remove an entry**
  ```
  del d[key]
  ```
- **Add the content of another dictionary**
  ```
  d.update(d2)
  ```

  where `d2` could be a dictionary or an iterable (e.g. list or tuple) of pairs. Python type casts the sequence into a dictionary before the `update` operation.

Dictionaries – 5 of 5

Examples

```
>>> d = {27:56, 12:37, 44:5, 12:73}
>>> len(d)
3
>>> d.keys()
[44, 27, 12]
>>> l = d.keys()
>>> l.sort()
>>> l[0]
12
>>> d[l[0]]
73
```

12:73 Replaces 12:37
both items have the same key (12),
and keys are unique

A dictionary is the equivalent of a *hash tables* in other languages, so the order of the keys in d is arbitrary. We need to *sort* the list of keys to access dictionary items in order.

Exercise 2.5

~5-10 mins

Sets – 1 of 5

Sets

- Container objects that contain *arbitrarily* ordered collections of *unique* objects
- Contain any kind of *hashable* (immutable) object
- There are two types of sets available in Python: `frozenset` – immutable and `set` – mutable
- Class names are `frozenset` and `set.`

Sets – 2 of 5

Creating sets

```
>>> s1 = set()
>>> s1
set([])
>>> type(s1)
<type 'set'>

>>> s2 = set([2,3,4,5])
>>> s2
set([2, 3, 4, 5])
```

```
>>> f2 = frozenset(s2)
>>> f2
frozenset([2, 3, 4, 5])
>>> type(f2)
<type 'frozenset'>
```

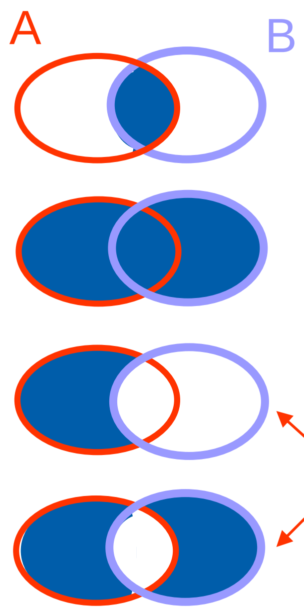
Type casting

```
>>> set((2,3,4))
set([2, 3, 4])

>>> set({1:2,3:4,5:6})
set([1, 3, 5])
```

Sets – 3 of 5

Non-mutating Methods

```
A.intersection(B)
   A & B
A.union(B)
   A | B
A.difference(B)
   A - B
A.symmetric_difference(B)
   A ^ B
```



The operators require two (frozen) sets. The methods accept any iterable object.

Union accepts both `set` and `frozenset` at the same time

A new `set/frozenset` is created.

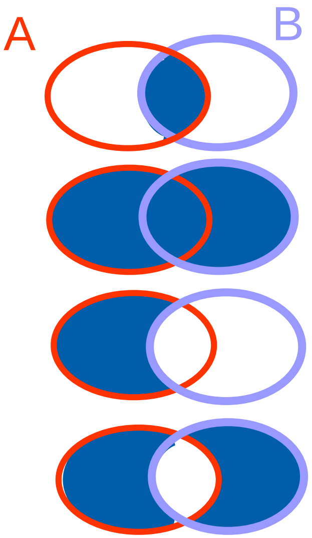
```
A.issubset(B)
   A | B == B

A.issuperset(B)
   A & B == B
```

Checks if A is a subset/superset of B

Sets – 4 of 5

Mutating Methods

```
A.intersection_update(B)
  A&=B
A.update(B)
  A|=B
A.difference_update(B)
  A-=B
A.symmetric_difference_update(B)
  A^=B
```



`difference_update` is only available for `set`

```
A.add(object)
A.remove(object)
A.discard(object)
A.clear()
```

`remove` raises **`KeyError`** if object is missing.

Nothing is returned.

A is changed in place

Sets – 5 of 5

Examples

```
>>> s1 = set([1,2])
>>> L2 = [2,3]
>>> f2 = frozenset(L2)
>>> s1.update(L2)
>>> s1
set([1, 2, 3])
>>> f2 & s1
frozenset([2, 3])
>>> s1.add(f2)
>>> s1
set([1, 2, 3, frozenset([2, 3])])
>>> 1 in s1
True
>>> s1.add(set())
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
TypeError: set objects are unhashable
```

Type of the expression comes from the ***first*** argument to the operator (in this case: `f2`)

`frozenset` is immutable and *hashable*, so it can be an element of a `set`. But a `set` can **not** be an element in `set` or a `frozenset`

Membership testing works for all container classes

Exercise 2.6

~5-10 mins

Booleans & Boolean Operators – 1 of 5

Booleans have only two possible values: True or False

- `>>> int(True)`
- `1`
- `>>> int(False)`
- `0`

– Class name is bool

So you can actually use **`bool`** in all kinds of arithmetic expressions. An **`int`** conversion is done first. Example:

```
>>> True + True
2
```

While interesting, don't do it just because it is possible ☺!

All objects can be *typecast* to bool

`False`

- Numbers that are 0
- Empty objects
- The **`None`** object

`True`

- Numbers that not are 0
- Not empty objects

Note: important not to be confused by, for example, list notation. For instance:

`[[]]`

is a list of one item: an empty list

Booleans & Boolean Operators – 2 of 5

Comparison operators

– They all return a `bool` object

```
a == b
a > b
a < b
a != b
a >= b
a <= b
a is b
not a
```

Do the ***objects*** that **a** and **b** are references to have the ***same value***?

Note: These two do not test the same thing!

Are **a** and **b** references to the ***same object***?

Booleans & Boolean Operators – 3 of 5

Comparison may be applied to all kinds of objects, examples:

Strings	Lexicographically
Lists & Tuples	Object by object from left to right
Dictionaries & Sets	Possible and deterministic
Compound objects	Recursively

Note: comparing *different types* to each other does not make sense, and will vary with the actual Python implementation. Example, how to interpret this?:

```
(10+32j) > 'Hello world?'
```

Booleans & Boolean Operators – 4 of 5

`and` + `or` do not evaluate to bool

Instead:

```
– a and b    # Returns a if a is false else b
– a or b     # Returns a if a is true else b
```

Maybe unexpected!

But very useful!

Example:

```
>>> "Hello" or "Empty"
'Hello'
>>> "" or "Empty"
'Empty'
```

Short-circuit

Righthand argument *only* evaluated *if needed*

Python 2.6, new syntax:

```
>>> res = (true_value if condition else false_value)
```

Parenthesis not *always* needed, but makes it easier to read and avoids problems when they **are** needed

Booleans & Boolean Operators – 5 of 5

Examples

```
>>> a = [1,2]
>>> b = a
>>> a == b, a is b
(True, True) # Same obj
>>> a is [1,2], a == [1,2]
(False, True) # Different obj
>>> [] or 0 or ""
'' # empty string notation

>>> a and 10 or 20
10
```

Similar to `a?10:20`
in C language, as long as "10" is True

Truth table for `a and b or c`

Evaluated strictly left to right,
and *has precedence over* ***or***

```
a and   b   or  c
-------------------------------
a       b       c       =   b
a       b       False   =   b
a       False   c       =   c
a       False   False   =   False
False   b       c       =   c
False   b       False   =   False
False   False   c       =   c
False   False   False   =   False
```

Exercise 2.7

~5-10 mins

Files – 1 of 5

A *file* is a stream of bytes that a program can read/write to

- Python can manage text files, compressed files (zip, gzip, tar, bz2), files containing serialised data (various ways), etc.
- We will focus on simple *text* files to illustrate the concepts.

Files

- File is a built-in object type
- Name of the data type is **`file`.**

File object provides methods for

- Opening files in different *modes*
- Reading and writing to files
- and more...

Files – 2 of 5

Opening a file

`f = open(filename[,mode="r",[bufsize=-1]])`

Closing a file

`f.close()` -garbage collection will do it otherwise
-good standard to do it explicitly

Reading from a file

`f.read()` -reads the *complete* file into a *string*
`f.readline()` -reads *one line* into a *string*
`f.readlines()`-reads the file into *a list of strings*

Lineseparator for text files

– On UNIX, always `'\n'`

– On other platforms - may vary, but see *modes* on next slide

Files – 3 of 5

Modes

A *string* that specifies how the file is to be opened/created

`'r'`	read-only; file must exist
`'w'`	write-only; if file exists, truncate and overwrite, otherwise create and write
`'a'`	append, write-only; if exist, append at the end of file, otherwise create and write
`'r+'`	read/write; file must exist
`'w+'`	read/write; if file exists, truncate and overwrite, otherwise create
`'a+'`	read/write; if file exists, append at the end of file, otherwise create.

Text/binary modes

Append `'t'` to the mode for *text* files, append `'b'` for *binary* files.

Ex: `'rt'` – read-only *text* file

Universal newlines

When reading textifles, use mode `'U'` (equivalent to `'rU'`) to automatically read and interpret `'\n'`, `'\r'` and `'\r\n'` as line separators

Files – 4 of 5

Write to a file

`f.write(s)` - writes a string to the file
`f.writelines(l)` - writes a list of strings to the file.

There are more methods/attributes

`f.flush()` - forces write to the file system.
`f.name` - the name of the opened file.
`f.mode` - the mode the file was opened in: read, write, append, etc.

Files – 5 of 5

Examples

```
>>> f = open("myfile", "w")
>>> f.write("Optimization matters\n")
>>> f.close()
>>> g = open("myfile")
>>> g.readline()
'Optimization matters\n'
>>> g.readline()
''
```

More information on Input & Output, see chapter 9, and on (text) files, see pp158-160 in the Course Book.
See also the “Base I/O Interface” chapter on pp 349 and fwd in the Course Book.

Exercise 2.8

~5-10 mins

The None type

None

- Is an empty placeholder object
- Has *no* methods or operators
- Can be compared to NULL in C
- Evaluates to boolean `False`
- All functions/methods without an explicit return value return `None` by default

```
>>> a = None
```

Mutability of types – 1 of 6

- As we have seen, we can have two or more references to an object
- This is very useful – *but be careful with mutating objects*.
- Lists are mutable objects, we'll use them as example.

Unexpected side effects
Copying a mutable object

```
>>> a = [1, 2, 3]
>>> b = a
>>> a[0] = 5
>>> print b
[5, 2, 3]
```

b and a are the *same* object

Augmented assignment woes
Copying a mutable object, then doubling the original

```
>>> a = [1, 2, 3]
>>> b = a
>>> a += a
>>> print a
[1, 2, 3, 1, 2, 3]
>>> print b
[1, 2, 3, 1, 2, 3]
```

b and a are the *same* object

Assignment woes
Change one reference, reflect the changes in the other

```
>>> a = [1, 2, 3]
>>> b = a
>>> a = a + a
>>> print a
[1, 2, 3, 1, 2, 3]
>>> print b
[1, 2, 3]
```

b and a are *different* objects

Mutability of types – 2 of 6

Copying a container object with references

– Copying all references in the object, shallow

```python
import copy
a = copy.copy(b)
# For lists, a = b[:] works the same way
```

– Copy the *complete* data structure recursively

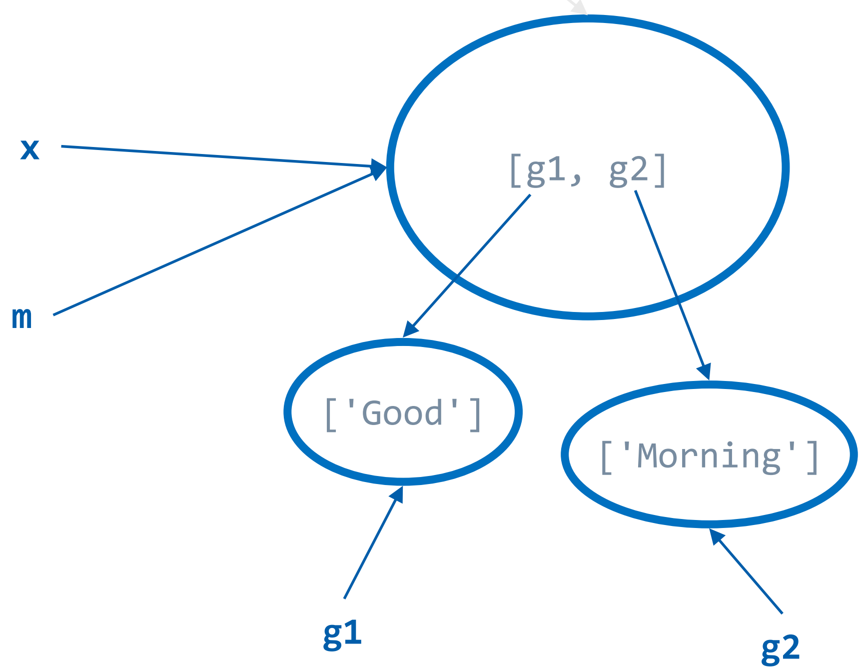
```python
a = copy.deepcopy(b)
```

Mutability of types – 3 of 6

Example using lists

```
>>> g1 = ["Good"]
>>> g2 = ["Morning!"]
>>> x = [g1, g2]
>>> m = x
```

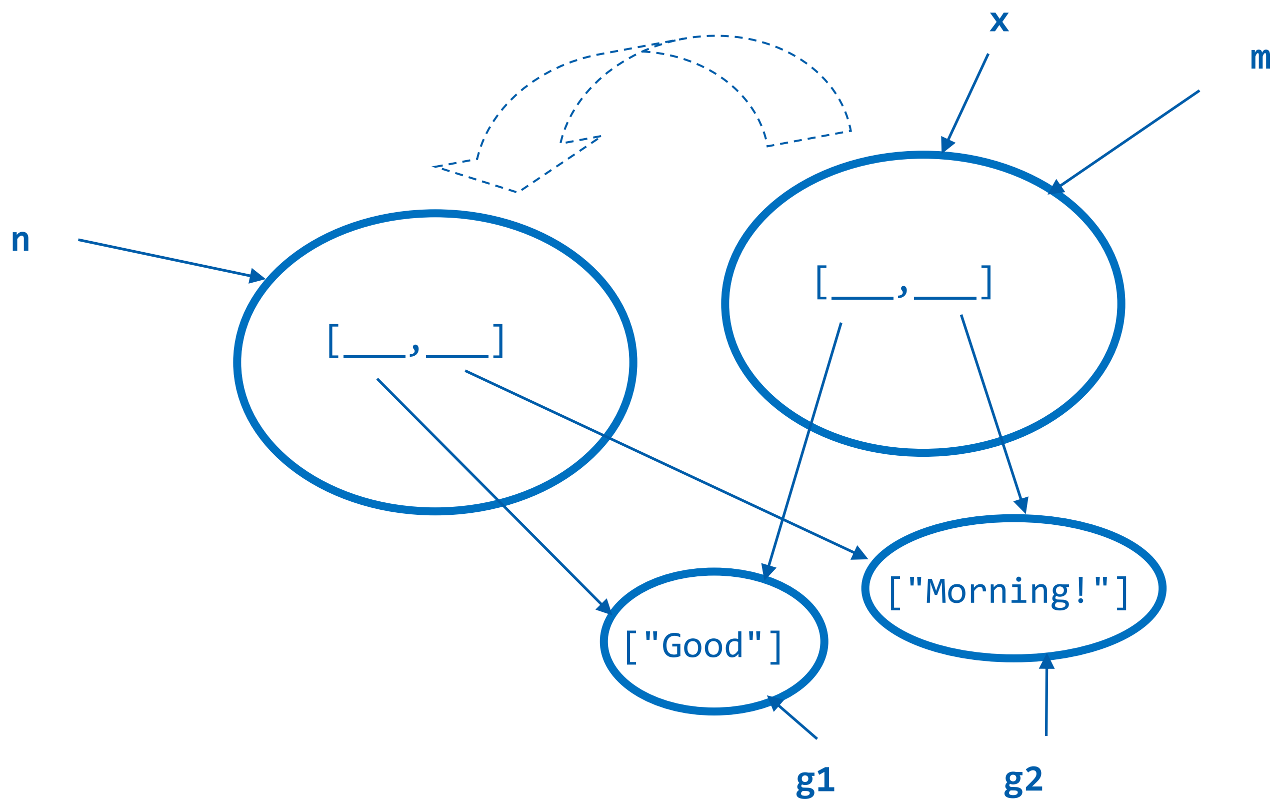
A circle indicates a *mutable* object.



Mutability of types – 4 of 6

"Shallow" Copy

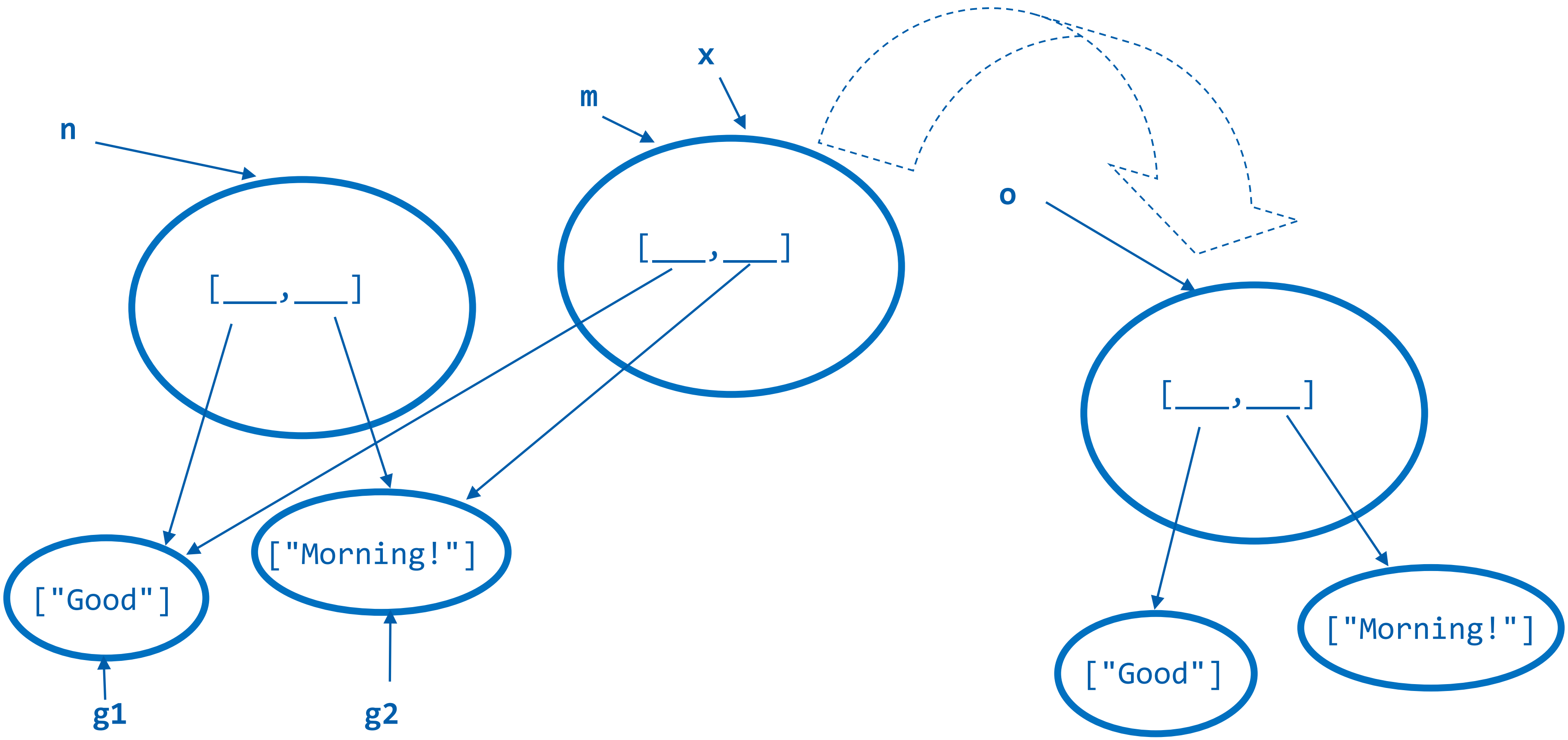
```
>>> n = copy.copy(x)
```



Mutability of types – 5 of 6

Recursive Copy

```
>>> o = copy.deepcopy(x)
```



Mutability of types – 6 of 6

Results

```
>>> g2[:] = ["Bye!"]
>>> x
[["Good"], ["Bye!"]]
>>> m
[["Good"], ["Bye!"]]
>>> n
[["Good"], ["Bye!"]]
>>> o
[["Good"], ["Morning!"]]
```

Exercise 2.9

~30 mins

4. Flow Control

Flow Control
`if-elif-else` & `while-else`
The `for` loop
`break` & `continue`
List comprehensions

Flow Control – 1 of 2

How Python runs a script

- Python code is executed statement by statement, from the top to the bottom.

- Flow control statements let us
  - modify the execution order.
  - define if blocks of statements should be executed or not.

- A block of statements is defined by indentation

Flow control – 2 of 2

Basic statements

- `if-elif-else`
- `for-else`
- `while-else`
- `try-except` (Covered later in "*Exception handling*")

Modifying statements

- `break`
- `continue`

No-operation statement

- `pass`

if-elif-else and while-else

```
if <test1>:
    <block1>
[elif <test2>:
    <block2>]
[else:
    <block3>]
```

- Avoid test expressions like:

  ```
  bool(a) == True
  len(s) > 0
  ```

  instead, use the "truth" value of the object

  ```
  a
  s
  ```

Example:

```
if a:
    <block1>
elif s:
    <block2>
```

```
while <test>:
    <block1>
[else:
    <block2>]
```

- The code in `<block1>` is executed repeatedly as long as `<test>` is `True`
- The code in `<block2>` is executed when `<test>` is/becomes `False`, *unless*
  - a `break` or `return` is executed in `<block1>` or
  - an *exception* occurs in `<block1>`

The for loop

```
 for <target> in <iterable>:
     <block1>
[else:
     <block2>]
```

must be an iterable object (list, tuple, etc)

- <target> is a variable that is bound to the objects in <iterable>, one by one.

- The code in <block2> is executed when the loop exits naturally (we run out of objects in <iterable>), unless
  - a break or return is executed in <block1> or
  - an exception occurs in <block1>

break and continue

- `break` exits the closest enclosing loop, without executing the `else` block
- `continue` jumps back to the header of the current `while/for` statement
- When using one ore more `break` statements inside a loop, you may need to check if the loop exits naturally or prematurely

```
f = open("myfile")
for row in f:
    if row.find("Jeppesen") != -1:
        print row
        break
else: print "Jeppesen not found"
```

A `file` object is also an *iterator* (something you can loop over)

Writing code like this is OK, <u>if</u> the clause body is just one logical line, and it makes to code *more* readable.

List comprehensions

Common pattern

```
rl = []
for x in my_iter:
    if x: rl.append(x+30)
```

optional part

With list comprehension

```
result = [x + 30 for x in my_iter if x ]
```

```
>>> [(a, b) for a in (1,2) for b in (3,4)]
[(1, 3), (1, 4), (2, 3), (2, 4)]

>>> [(a, b) for a in (1,2) for b in (3,4) if (b+a)%2]
[(1, 4), (2, 3)]
```

Exercise 3

Now it is time for exercise 3

5. Functions

Overview
Defining functions
Namespaces
The `return` statement
Arguments
Functions as arguments
`lambda`

Defining functions

Functions are callable (first class) objects.

- They behave and can be used similarly to other objects we have seen

Syntax:

```
def <name>(parameters):
       [<doc_string>]
       <statements>
       [return <value>]
```

- A function **object** is created (at runtime)
- The variable/identifier **`<name>`** is bound to the object
- The `<doc_string>` is documentation.

Namespaces – 1 of 2

- Functions in Python have local namespaces
- Each call to a function creates a namespace
- The namespace contains the arguments and the variables assigned in the function body
- Python looks for variables in three steps (LGB):
  1. **L**ocally (inside the current function)
  2. **G**lobally (in the enclosing module)
  3. **B**uilt-in-names (e.g. "`len`", etc)

Namespaces – 2 of 2

- If you want to assign a global variable inside a function – use the `global` keyword

```
>>> a = b = 10
>>> c = 5
>>> def test():
...     global b
...     a = b = c
>>> test()
>>> (a, b, c)
(10, 5, 5)
```

Local variable **c** not found, look globally

Local variable **a**

- It is possible to "create" a global variable inside a function, before it actually exists in the namespace outside the function
- *avoid this, it is confusing*

```
>>> def f():
...     global x
...     x = 10
...
>>> x
Traceback (...):
...
NameError: name 'x' is not defined
>>> f()
>>> x
10
```

The return statement

The return statement is optional. If omitted, None is returned

```
>>> def func1(): print 'it works'
...
>>> res = func1()
it works
>>> type(res)
<type 'NoneType'>
```

Use a tuple if you want to return many values from a function

```
>>> def test(): return (10, 20, 30)
...
>>> a, b, c = test()
>>> a
10
>>> c
30
```

You can unpack the tuple returned by the function this way

Arguments – 1 of 4

- Function calls specify arguments by *position* or by *name*
- *Default values* for parameters are supported in function definitions
- It is possible to define functions that take *any number of arguments*
- Each parameter provided as argument is a *local variable in the function body*. It is bound to the **object that was passed as argument** – this is called *call-by-sharing*

More information on this subject:
http://effbot.org/zone/call-by-object.htm and http://www.python.org/doc/current/ref/objects.html

Arguments – 2 of 4

```
def myfunc(a, b=1, c=10, d=[]):
    return(a, b, c, d)
```

This is how **default values** to functions are defined

Parameters without default values *must* be specified in the function call

```
>>>   myfunc()
Traceback (most recent call last):
...
TypeError: myfunc() takes at least 1 argument (0 given)
>>> myfunc(10, 5, d=44)
(10, 5, 10, 44)
```

Arguments – 3 of 4

Unspecified number of arguments

- `*<name>` - remaining *positional* arguments as a ***tuple***
- `**<name>` - remaining *named* arguments as a ***dictionary***

Example

```
def myfunc(a, *b, **c):
    print a, b, c
```

Must be in that order

```
>>> myfunc(1, 2, 3, 4, 5, x=6, y=7, z=8)
1   (2, 3, 4, 5)   {'x': 6, 'y': 7, 'z': 8}
```

The opposite is also possible

```
>>> a = range(6) # a tuple
>>> b = {"w":99, "v":88} # a dictionary
>>> myFunc(*a, **b) # pass arguments with this notation
0 (1, 2, 3, 4, 5) {'w': 99, 'v': 88}
```

Arguments – 4 of 4

A word of warning

- Objects for default values are created at the same time the function object is created
- *Mutable objects as default values may have unexpected side effects*

```
def f(a, b=[]):
    b.append(a)
    return b

>>> f(1)
[1]
>>> f(2)
[1, 2]
```

The `[1,2]` is unexpected, we expected `[2]`. The problem is the *mutable* default value to b (a list).

```
def f(a, b=None):
    if b == None:
        b = []
    b.append(a)
    return b

>>> f(1)
[1]
>>> f(2)
[2]
```

Functions as arguments – 1 of 1

Functions are first class objects and can be used arguments to function calls

Lets look at an example using

- **`map`**`(`*`function`*`, iterable)`
  Applies function to each element in the iterable and returns a list with the result

- **`filter`**`(`*`function`*`, iterable)`
  Returns a list containing all objects in iterable for which function is true

Using `map` and `filter` to generate a list of squares of odd integers

```
>>> def f1(x): return x**2
...
>>> def f2(x): return x%2
...
>>> map(f1, \
        filter(f2, xrange(10)))
[1, 9, 25, 49, 81]

# Note:
# x%2 is 1 (True) when odd
```

lambda – 1 of 2

– Sometimes it is convenient to define a function as an *expression*

– It is possible if the function body consists of only a *single return statement*

Without lambda

```
def f(a,b):
    return a + b
```

With lambda

```
f = lambda a,b: a + b
```

The assignment is optional and can be omitted, for instance when using `lambda` with **`map`** or **`filter`**

lambda – 2 of 2

Previous examples using lambda can be... intimidating

```
>>> map(lambda x: x*x,
        filter(lambda x: x%2, xrange(10)))
[1, 9, 25, 49, 81]
```

– Note how we use **`lambda`** as an *expression* only

Often it is possible/better to use list comprehensions

```
>>> [x**2 for x in xrange(10) if x%2]
[1, 9, 25, 49, 81]
```

Exercise 4

Now it is time for exercise 4

All you want to know about functions:

Chapter 6 in the Course Book

6. Modules & packages

Modules overview
Importing modules
Module attributes
Module Examples
Packages
Final advice on modules

Modules overview

- Modules are the largest components you use to build your programs and are created from Python source code files (files containing python code, with the `.py` suffix)

- A *module* is an object that can be seen as a *namespace* containing a set of named *attributes*. The *name of the source file* becomes the *name of the module object*

- Objects in modules are attributes of the module, and are accessed by qualification: `module.attribute.` This is equivalent to using the built-in function `getattr(module, 'attribute').`

- The **`path`** attribute of the **`sys`** module defines module directories. The paths defined in the environment variable `$PYTHONPATH` are added to `sys.path` when python starts

Importing modules – 1 of 4

- A module object is created from a file by the **`import`** statement

- *All* statements in the module file are executed when it is imported the first time, and a precompiled byte-code-file is created

- The attribute `sys.modules` is a dictionary mapping module names to all currently imported module objects

Importing modules – 2 of 4

Creating a module object

```
import <module> [as <alias>]
```

Importing specific objects from a module

```
from <module> import <v1> [as <alias>]
from <module> import <v1>, <v2>, ...
from <module> import *
```

Avoid!
Pollutes the current namespace

- The `from` statements create *variables* in the current name-space, that are *bound to objects in the module object*.

Importing modules – 3 of 4

Contents of file "ex4.py":

```python
s = "Hello World!"
print s
```

```
> python
>>> import ex4
Hello World!
>>> import ex4
>>> ex4.s
Hello World!
>>> Ctrl-D
> ls ex4*
ex4.py ex4.pyc
```

When the **import statement** is invoked (high level):

1. Check in the current python session if a **module object** based on the ex4.py source file already exists. If it does, continue with step 4.
   If not, continue with step 2.

2. Check whether a byte code file does not exist, or whether the source file is newer than the byte code file. If so, read and (re)compile the source file to byte code, execute it, create the module object in the python session and save the byte code to a file **`ex4.pyc`**

3. If the byte code file **does** exists and the source file is **not** newer, read and execute the existing byte code file, create the module object in the current python session. No compiling occurs.

4. Bind the variable **`ex4`** to the newly created module object.

Note: the file name ending ".py" is required for python to be able to import files.

Importing modules – 4 of 4

Common pitfalls

- The **`import`** statement <u>does not re-execute</u> the code for modules which are already imported.

- To consider changes in the module file:
  - use **`reload`**`(`*`module`*`)`
  - or restart the interpreter – preferred way

Use `reload` *only* during development!

Removed attributes of a module file are *NOT removed* from memory by **`reload`** – lingering attributes may lead to hard to find errors.
See page 149, in the Course Book

Module attributes

Python adds some attributes into modules

`__dict__`	a dictionary containing all (other) attributes of the module
`__name__`	name of the module (`__main__` for the "top" module)
`__builtins__`	used by Python to find built in objects
`__doc__`	defined by a string at the top of the module file (does not apply to `__main__`)
`__file__`	name of the module file (does not apply to `__main__`)

There are no “true” global variables in Python.

All variables are *attributes* of a namespace, such as a `function`, `class` or `module` (e.g. the `__main__` module).

Modules example – 1 of 3

example.py

```python
"""Some text describing the module"""
def func():
    "Text describing the function"
    print "Module: %s" % __name__

func()
```

documentation / help texts

```
> python example.py
Module: __main__
```

Modules example – 2 of 3

```
> python
>>> import example as e
Module: example
>>> e.func()
Module: example
>>> import example
>>> example is e
True
```

Note that he module file is **not** re-executed. You only get a (new) *reference* to the module object instead.

Modules example – 3 of 3

```
>>> example.__doc__
'Some text describing the module'

>>> example.__dict__.keys()
['__builtins__', '__name__', '__file__', '__doc__',
'func']

>>> dir(example)
['__builtins__', '__name__', '__file__', '__doc__',
'func']
```

Packages – 1 of 2

Modules can be organised and grouped into packages

– Module packages can be organised in tree structures

– Each sub directory to a `sys.path` directory containing a file named `__init__.py` becomes a module package

Example

`$CARMSYS/lib/python``/carmensystems/basics`

in **`sys.path`**

Packages – 2 of 2

Importing packages

- `__init__.py` is executed when the module object for the package is created – use to initialise the package
- Individual modules from the package can be imported

Examples

```
import carmensystems.basics.abstime
from carmensystems.basics import abstime
```

Final advice on modules

Attribute access

– All attributes of a module are possible to access . Python has no “real” private - as in "protected from access by the system" – attributes

– Private attributes are respected by *convention* - respect this convention. Good, complete and documented public API's is what keeps users from venturing into private areas of the code.

– Use "_" as *prefix* for private attributes that are not intended for use outside of the module, class, etc.

– While it is possible to manipulate attributes of a module from outside it, it should be avoided most of the time.

Exercise 5

In-depth details on Modules and Packages:
Chapter 8 in the Course Book

7. Object-oriented Python

Overview of classes - Defining a class
Simple class example
Constructors & destructors
Attribute access - Inheritance
Class-level methods - Special methods
Final hints & Reminders
Summary

Overview of classes – 1 of 2

Definition of a class

– A class is a user-defined type with its *own namespace* containing arbitrarily named attributes

– When *called* like a function, it will return an new object – an instance of the class

– Functions defined in classes are called *methods*

– Classes can inherit attributes (and thus functionality) from one or more other classes.

Overview of classes – 2 of 2

Definition of a class

– Python supports *multiple inheritance* and *mixins*

– A class may implement a range of *special methods*. These are called *implicitly* when various operations are performed on its instances.

– Classes in Python are *first-class objects* – classes are ordinary Python objects that can be treated and manipulated like such.

Defining a class

The `class` statement

```
class classname(base-classes):
    statement(s)
```

– The `class` statement does not create an actual instance of the class – it only defines a *class object* with a set of attributes *shared* by all its instances.

– Class *instances* are created by *calling* the class, as if it were a function:

```
my_inst = MyClass()
```

classname

- Identifier *bound* to the class object. Created after the class statement finishes execution.

base-classes

- Optional comma-separated sequence of class-objects to inherit attributes from (aka: superclasses, bases, parents)

statements(s)

- The class body, non-empty sequence of statements. Note that *functions* defined in classes are called *methods*

Simple class example

Simple example

- The class subclasses the built-in ***`object`*** base class
- First parameter (`self`) to methods is *always* a reference to the class *instance object*

  [PEP 8] The name `self` is by convention - always use `self`!

- Instance attribute access is done using the **`self`** parameter

```
class SimpleUser(object):
    """Minimalistic class
    featuring user info"""

    def __init__(self):
        # always 'declare'
        # all your instance vars
        self.name = None
        self.age = None
        self.shoe_size = None

    def get_name(self):
        "Return username"
        return self.name
```

PEP 8 can be found at http://www.python.org/dev/peps/pep-0008/

Constructors & destructors

Constructors – well, kind of...

- If defined, the special method `__init__` is called by Python to initialise the instance
- `__init__` behaves in essence like a *constructor* – however it gets called *after* the instance object has been created (see page 55 in the Course Book)

Destructors

- Similarly, the special method `__del__` is called by Python when the instance is *about* to be destroyed

`del x` does *not* directly call `x.__del__()`

`del x` only decreases the reference counter. `x.__del__()` is called once the reference counter for `x` actually reaches zero.

Read more about object creation and destruction on pp 54-55 in the Course Book

Attribute access – 1 of 3

Getting/Setting data in instances

- To ***get*** the value of an attribute
  `my_var = instance.attr`

- To ***set*** a value of some attribute
  `inst.attr = some_val`

Rules of Thumb for attributes

- Initialize *all* your instance attributes in the `__init__` method. Do not add/create attributes to the instance anywhere else.

- Breaking this rule leads to errors that may halt execution of your code (*AttributeError*)

```
# Create a user,  give it a name
>>> my_user = SimpleUser()
>>> my_user.name = 'Guido'
>>> my_user.age = 46
>>> my_user.shoe_size = 42

# Some attributes can be
# accessed in several ways
>>> my_user.name
'Guido'
>>> my_user.get_name()
'Guido'
>>> my_user.age
46
>>> my_user.age = 47
>>> my_user.age
47
```

There is no actual need for *setter* and *getter* methods in Python
– it is not the conventional Python way, experienced python coders will be confused.

Attribute access – 2 of 3

Getting and setting data in class objects

- Let's look at an example where we store data in the class object
- We want to be in control of the number of user instances that have been created and are still alive in the system

Usage

```
>>> u1 = OtherUser('Carmen')
>>> u2 = OtherUser('Guido')
>>> u1.num_users()
2
>>> u2 = u1
>>> OtherUser.num_users()
1
```

```
class OtherUser(object):
    # we use this variable
    # to keep track of users
    _user_count = 0

    def __init__(self, name):
        self.name = name
        # we want to access the class
        # itself, not the instance
        # hence __class__
        self.__class__._user_count += 1

    def __del__(self):
        self.__class__._user_count -= 1

    # Now we create a class-level method
    # it can be called w/o instances
    @classmethod
    def num_users(cls):
        return cls._user_count
```

More on class-level methods and decorators (the @-notation) in coming slides!

*Note: **copying** instances using the `copy` module does not call the `__init__` method, but there are workarounds. Read more about it at https://docs.python.org/release/2.6.6/library/copy.html*

Attribute access – 3 of 3

Assume that you have an instance `a` of a class `A`

- How is the attribute `a.b` looked up?

Python looks for the attribute `b` in

- the class *instance* `a`
- the class *object* `A`
- the objects of the *super class(es)* of `A`
- *Inherited* classes: left to right, depth first
- Note that methods defined in classes are *unbound* and not normally accessible for execution unless the class is *instantiated* first.

Inheritance – 1 of 2

- A class can inherit from zero or more classes - inheritance can be of *any depth*

- *All* attributes of the superclasses are inherited

Let's look at a simple example

- Create a subclass to our `OtherUser` class, call it `Crew`, to both manage information about Crew, and to keep track of how many crew we have active

Inheritance – 2 of 2

Implementing a Crew class

Usage

```
>>> c1 = Crew(fname='Elrey',
              lname='Jeppesen',
              pos='Captain',
              age=40)
>>> c2 = Crew(fname='Amelia',
              lname='Earhart',
              pos='Captain',
              age=40)
>>> c1.num_users()
2
>>> c2.num_users()
2
>>> Crew.num_users()
2
>>> del c2
>>> c1.num_users()
1
>>> c1.name, c1.pos
('Elrey Jeppesen', 'Captain')
```

```
class Crew(OtherUser):
    "crew user"

    def __init__(self, fname,
             lname, pos, age):
        # common pattern to access
        # the super class
        super(Crew,
              self).__init__(
                  name='%s %s'%(fname,
lname))
        self.fname = fname
        self.lname = lname
        self.pos = pos
        self.age= age
```

- The `'name'` attribute comes from the parent class (`OtherUser`)
- The 'super' pattern above is to make sure that the parent class' methods are called properly. For details, see: *http://rhettinger.wordpress.com/2011/05/26/super-considered-super/*

Class-level methods – 1 of 2

– Methods in classes must be *bound to an instance* before they can be called.

– But methods in classes can also be free from instance binding; we call those methods *class-level methods.*

Static Methods

– Can be called on a class or its instances without any instance parameters – no binding to instances nor class

– The class basically acts as a *grouping* entity.

Class Methods

– Can be called on a class or its instances – the method is bound to the class (or the class of the instance) by the first parameter (like methods).

Class-level methods – 2 of 2

Defining Static and Class Methods

- There is a built-in type `staticmethod` that is used to declare static methods.
- But they can also be defined with a decorator – the `@staticmethod` just above the method declaration.
- Same for class methods. Either use the `classmethod` type or the `@classmethod` decorator.

A decorator works like this:

```
# When python sees this code...
@somedec
def foo(...):
    ...
# ...the following will automagically be
# executed (after the function is created)
foo = somedec(foo)
```

```
class Foo:
    # define static method
    # with staticmethod type
    def bar():
        print "static"
    bar = staticmethod(bar)
    # define static method
    # with decorator
    @staticmethod
    def decbar():
        print "more static"
    # define class method
    # with classmethod type
    def foo(cls):
        print "classy"
    foo = classmethod(foo)
    # define class method
    # with decorator
    @classmethod
    def decfoo(cls):
        print "classier"
```

Special methods - 1 of 3

- Special methods allow *modification and customization* of an object's behaviour. You can customize almost everything

- Python supports a large number of special methods for classes - for instance, there is a special method for each mathematical operator:
  `__del__`   `__add__`   `__mul__`   etc...

- All *core* special methods are listed and explained briefly on pages 47-63 in the Course Book.

- Some extension modules may also provide hooks for special methods for the operations that they provide (e.g. the `copy` module does)

Special methods – 2 of 3

Let's create a class which implements a data type that is a FIFO queue

- Methods `get()` and `put(item)` will be defined
- Function `len` will return the number of items
- `q += a` will do the same as `q.put(a)`
- `-q` will do the same as `q.get()` (just to show that it is possible!!!)
- `print` should show the contents of the queue and indicate that it is a queue

Special methods - 3 of 3

Desired behaviour

```
>>> q = Queue([10,2,3])
>>> print q         # __str__
Q[10, 2, 3]

>>> a = -q          # __neg__
>>> b = q.get()
>>> print (a,b,q) # __repr__
(10, 2, Q[3])

>>> q += 4          # __iadd__
>>> q.put('a')
>>> q               # __repr__
Q[3, 4, 'a']
>>> len(q)          # __len__
3
```

```
class Queue(object):
    """ A very simple Queue Class """
    def __init__(self, q=None):
        if q is None: self._q = []
        else:  self._q = list(q)
    def put(self, item):
        self._q.append(item)
    def get(self):
        return self._q.pop(0)
    def __len__(self):
        return len(self._q)
    def __str__(self):
        return "Q"+str(self._q)
    def __repr__(self):
        return "Q"+repr(self._q)
    def __neg__(self):
        return self.get()
    def __iadd__(self, item):
        self.put(item)
        # We MUST return the instance
        # for __iadd__ to work
        return self
```

Final hints & reminders

- It is good practice to create *all attributes that are ever used in a class instance*, in the `__init__` method

- All updates and accesses of **class** attributes through classes and class instances should be done by *calls to methods* to avoid mistakenly using an *instance* attribute.

- If `C` is a class having a normal method `m` and `c` is an *instance* of `C`. Then `C.m(c)` and `c.m()` do the same. Example: `str.upper(s) == s.upper()`

Exercise 6

Now it is time for exercise 6

8. Exceptions

Exception Overview
Defining Exceptions
Raising exceptions
Exception attributes
Final advice

Exception overview – 1 of 4

– A statement like

```
a = 1/0
```

terminates Python:

```
>>> a = 1/0

Traceback (most recent call last):
    File "<stdin>", line 1, in ?
ZeroDivisionError: integer division or modulo by zero
```

Exception overview – 2 of 4

Avoiding program halt – managing exceptions

– When errors occur, it is possible to catch them with `try-except` statements

```
try:
    1/0
except ZeroDivisionError:
    print "Div by Zero Error caught"
    print "Moving on..."
```

– This way, errors can be managed and program execution is not halted.

Exception overview – 3 of 4

Python provides a set of predefined exception classes

- BaseException
  - GeneratorExit
  - KeyboardInterrupt
  - SystemExit
  - Exception
    - StopIteration
    - StandardError
      - ...
      - AttributeError
      - ...
      - Lookuperror
        - IndexError
        - KeyError
  - ...
  - SyntaxError
  - ...

This is only a part of the class tree for exceptions in python 2.6

See Course Book, page 87 and pages 84-89 for more information on Exceptions

Exception overview – 4 of 4

Customized Exceptions

– You can define your own exception classes for when you need to customise error management or increase granularity of the errors caught

– Create a subclass to an existing exception class

– Normally, you use an empty class body:

```
class ExampleError(StandardError):
    "Just an example"
```

A doc string is OK instead of a `pass` statement.

– When an exception is raised, an *instance* of the exception class is created

Exception Syntax

Exception syntax (Python 2.6)

```
try:
    statements
except [expression [as target]]:
    statements
[except [expression [as target]]:
    statements]
[else:
    statements]
[finally:
    statements]
```

The `else` clause will be executed when no error has occurred.

`finally` is a clean-up-handler. It can occur exactly once in the try/finally statement and will always executes, **exception or not**.

Example where this makes sense:

```
f = open(someFile, w)
try:
    operate_on_file(f)
finally:
    f.close() # no matter what, we
              # always close f
```

Exception attributes

Getting more info from exceptions

```
try:
    f = open(file)
except EnvironmentError as err:
    print "Can't open '%s'. Reason: %s" % \
            (err.filename, err.strerror)
    raise
```

- When raised, we bind the exception object raised to the variable `err`, giving us access to the exception attributes

- Attributes and information available will vary across exception types

Raising exceptions

With the `raise` statement you can raise exceptions yourself

Syntax

```
raise [ExceptionClass(args)]
```

Basic raise

```
class Err(StandardError):
    "Just an example"
```

```
>>> raise Err("This kind of error")
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
__main__.Err: This kind of error
```

Propagating Exceptions

```
try:
    statement(s)
except (KeyError, IndexError):
    fix_stuff()
except ExampleError:
    fix_other_stuff()
    raise # Re-raises ExampleError
[except:
    raise] # Redundant
```

To propagate an exception upwards without changes from an exception handler:
Give *no arguments* to raise**.**

Uncaught exceptions will always propagate upwards.

Final advice

- Avoid large try blocks – break them up in smaller logical groups

```
try:
    x = [1,2]
    c = x[3]
    ...
    massive_amount_of_code()
    ...
except KeyError:
    print "oops!"
```

- Avoid too general except statements

```
try:
    do_lots_of_stuff()
except:
    print "uh... err?"
```

- The `try/except/pass` pattern – never, ever use it

```
try:
    single_big_call()
except:
    # previous call
    # should be safe
    pass
```

- **Accessing internal exception info**
  - `sys.exc_info()` gives reference to the current exception object

Note: Try to avoid replacing proper logging with printing `exec_info` to `stdout`.

Exercise 7

More on exceptions: pages 85-89 in the Course Book

9. Iterators & generators

Iterators

Generators

Iterators – 1 of 4

An iterator is an object that

- provides the method `next`, which returns the next item of the iterator.
- raises `StopIteration` exception, when there are no more items.

Functions to use for creation of iterators

- `enumerate` and `itertools.izip`
  - Normal `zip` creates large *lists*, not iterators

Creating your own iterator

- The `iter` *function* creates an iterator from iterable objects (lists, tuples, etc).

Iterators – 2 of 4

Creating iterators with `iter`

`iter(collection)`

- `collection` can be a list, tuple, etc
- `iter` is called *implicitly* by (for instance) the `for` statement
- `iter` calls the special method `__iter__` of the object.

```
>>> it = iter([1,2,3])
>>> while True: print it.next(),
...
1 2 3
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
StopIteration
>>> it == iter(it)
True
```

Iterators – 3 of 4

The `itertools.izip` function

```
from itertools import izip
izip(s1 [, s2 [..., sn]]) ->
     [(s1[0], s2[0],..., sn[0]), (...)]
```

– Like `zip` except that it returns an iterator instead of a list. Used for lock-step iteration over several iterables at a time. Returns tuples, where each tuple contains the *i-th* element from each of the argument sequences.

More on `zip` can be found on https://docs.python.org/2.6/library/functions.html
More on `itertools.izip` can be found on https://docs.python.org/2.6/library/itertools.html

```
>>> l1 = [0, 1]; l2 = ["a","b"]
>>> iz = izip(l1, l2)
>>> print iz
<itertools.izip object at 0x..>
>>> for i in iz: print i
(0, 'a')
(1, 'b')
>>> d = dict(izip(l1, l2)); print d
{0: 'a', 1: 'b'}
```

```
>>> iz = izip(l1, l2) # recreate it
>>> for a, b in iz: print b, a
a 0
b 1
```

Compared to `zip`

```
>>> z = zip(l1, l2); print z
[(0, 'a'), (1, 'b')] # a list
>>>
```

Iterators – 4 of 4

The `enumerate` class

`enumerate(iterable)`

- A data type (class) that, given an `iterable` object, returns an `enumerate` object
- The `enumerate` object yields *pairs* (tuples) containing a count from zero and a value yielded by `iterable`
- `enumerate` is useful for obtaining an indexed list of the items in `iterable`

```
>>> l = ["a","b"]
>>> enumerate(l)
<enumerate object at 0x...>

>>> for a in enumerate(l):
...     print a
(0, 'a')
(1, 'b')

>>> d = dict(enumerate(l))
>>> d
{0: 'a', 1: 'b'}
```

Generators – 1 of 3

- A *generator* is a *function* that contains a `yield` statement
- *Calling* a generator *returns* a **generator object**
- Calling this objects `next()` method executes the generator function to the *next* `yield` statement and *stops execution* until `next()` is called again
- `StopIteration` is raised when the function terminates

```
def myCounter(v=0):
    yield 'start'
    while True:
        yield v
        v += 1

>>> c = myCounter(3)
>>> c
<generator object at 0x...>

>>> c.next()
'start'

>>> c.next()
3

>>> c.next()
4
```

Generators – 2 of 3

Generator expressions

- *Generator expressions* are like list comprehensions, but *generator objects* are created instead.

- Using generator expressions means that no sequence object is created – you save memory.

```
>>> x = ((a,b) for a in (1,2) \
        for b in (3,4) if (b+a)%2)
>>> x
<generator object at 0x...>
>>> len(x)
Traceback (most recent call last):
TypeError: len() of unsized object
>>> x.next()
(1, 4)
>>> x.next()
(2, 3)
>>> x.next()
Traceback (most recent call last):
  File "<console>", line 1, in ?
StopIteration
```

Generators – 3 of 3

Built-in
& Std Lib generators

- There are many built-in callable objects that produce iterators
- Built-ins: `enumerate`, `xrange`, `reversed`
- The module `itertools` produces high-performace tools for iterators

```
>>> g = (a*a for a in xrange(4))
>>> g
<generator object <genexpr> at 0x...>

>>> for it in g: print it,
...
0 1 4 9

>>> print sum(a*a for a in xrange(4))
14
```

More in-depth information on generators can be found in the Course Book, pages 102-108

Exercises 7 & 8

Now it is time for exercise 7 & 8

10. Profiling & performance

Optimization matters
Speed of Python
Measuring performance
`time`
`timeit`
`profile & pstats`
Summary

Optimization Matters

But don't overdo it!

"First make it work. Then make it right. Then make it fast."

- Kent Beck (or his Dad)

Premature optimization is bad

- You end up may optimizing what you won't be using – not optimal at all.

But optimization does matter!

- Once your code behaves as it should – how can you make it faster?

Or does it?

- You don't always need to reach the Speed of Light – *good enough might be sufficient*, given all constraints (time, maintainability, etc)

Speed of Python

Python is usually fast enough

- Python is written in highly optimized C.

If speed is a disappointment

- Re-examine your algorithms and data structures, avoid Anti-patterns and use Best Practices.

Avoid NIH Syndrome

- Python is faster than C when it invokes faster, better libraries
- Use the **StandardLibrary** as much as possible – maybe your data structure is there **already?**

Still not fast enough?

- Check for bottlenecks, *internal and external*
- Use modules *profile* and *timeit* to see how fast you code is running
- Move CPU-intensive code to C-extension modules

Measuring performance

Hotspots

- Areas of your code where the computer spends most of its time.

Modules that time/profile execution time

- The **`time`** & **`timeit`** modules measure precise performance of specific *code snippets*

- The **`profile`** module helps you find hotspots (don't guess!) in your code (not just snippets) over one or more runs

- The **`pstats`** module analyses/presents the results of `profile`

time

- `time` is a general module for manipulating time, time zones, etc.
- function `time.time` returns the time as a floating point number expressed in seconds since the epoch, in UTC

```
def build_string():
    t = time.time()
    my_str = ''
    for a in range(100000):
        my_str = my_str + get_string()

    print "Len: ", len(my_str)
    print "Exec time: ", time.time() - t
```

For more information about the time module, including how to use it programatically, see https://docs.python.org/2.6/library/time.html

timeit – 1 of 3

- Measure execution time for *snippets* of code in an easy way
- Usually called from the command line, but can be accessed programatically – read the docs!
- The timer function is platform dependent – granularity varies

```
$ python -m timeit "import itertools;" \
?   “itertools.repeat(None,1000)"
1000000 loops, best of 3: 0.848 usec per loop

$ python -m timeit “xrange(1000)"
10000000 loops, best of 3: 0.167 usec per loop

$ python -m timeit "range(1000)"
100000 loops, best of 3: 8.59 usec per loop
$
```

python –m<mod>: Execute module <mod> as a script

Setup statements Prepare the timing

"…best of 3:" *Runs timeit(…) 3 times (default)*

For more in-depth information about the timeit module, including how to use it programmatically, see https://docs.python.org/2.6/library/timeit.html

`timeit` – 2 of 3

Example

- Measure how long time it takes to build a huge string from smaller parts.
- Keep the timing "built-in".

Results

- Output the length of the resulting string to stdout
- Print the execution time.

timeit – 3 of 3

```
# -*- coding: iso-8859-15 -*-
# This program builds a big string by
# list assembly (str.join)
import time
def get_string():
    return "abcdefghijklmno" \
           "pqrstuvwxyzåäö"

# Build the huge string
def build_string():
    t = time.time()
    my_list = []
    for a in range(100000):
        my_list.append(get_string())

    print "Len:", len(''.join(my_list))
    print "Exec time: ", time.time() - t

if __name__ == '__main__':
    # Setup the timer object
    import timeit
    ti = timeit.Timer(
           stmt='build_string()',
           setup='from __main__ import ' \
                 'build_string')
    # Time one run
    tt = ti.timeit(number=1)
    print "Timeit: ", tt
```

```
$ python2.4 \
join_string.py
Len: 2900000
Exec time:
0.0350160598755
Timeit:  0.0353429317474
```

profile & pstats – 1 of 4

Features

- Lets you gather performance statistics over one or more runs, with known input to your program
- Allows for calibration, call overhead can be compensated for
- Profile data is collected to a file – analysis is performed with the **pstats** module

Examples

```
# profile an app to stdout
# main entry point is foo()
import profile
profile.run('foo()')

# send stats to file
# foostats.dat
import profile
profile.run('foo()', 'foostats.dat')

# profile from external script
# to file foostats.dat
import profile
profile.run('import mymodule;' \
            'mymodule.foo()',
            'foostats.dat')
```

`profile & pstats` – 2 of 4

Let's profile

- We will profile our previous script
- We will create an external script that will perform the actual profiling
- Data is output to a file for later analysis
- We run `pstats` to analyze the results.

```
import profile

profile.run('import join_string;' \
            'join_string.build_string()',
            'joinstats.dat')
```

profile & pstats – 3 of 4

```
>>> import pstats
>>> j = pstats.Stats('joinstats.dat')
>>> j.sort_stats('cumulative').print_stats()
Tue Aug 12 10:07:53 2008    joinstats.dat

         200008 function calls in 0.420 CPU seconds

   Ordered by: cumulative time

   ncalls tottime percall cumtime percall filename:lineno(function)
        1   0.000   0.000   0.420   0.420 profile:0(import join_string;
                                                    join_string.build_string())
        1   0.270   0.270   0.420   0.420 /.../join_string.py:10(build_string)
        1   0.000   0.000   0.420   0.420 <string>:1(?)
   100000   0.090   0.000   0.090   0.000 /.../join_string.py:5(get_string)
   100000   0.050   0.000   0.050   0.000 :0(append)
        1   0.010   0.010   0.010   0.010 :0(join)
        1   0.000   0.000   0.000   0.000 :0(len)
        1   0.000   0.000   0.000   0.000 :0(setprofile)
        1   0.000   0.000   0.000   0.000 /.../join_string.py:5(?)
        1   0.000   0.000   0.000   0.000 :0(range)
        0   0.000           0.000         profile:0(profiler)


<pstats.Stats instance at 0x2a95808a28>
```

profile & pstats – 4 of 4

Useful commands – examples

– Which algorithms consume most time executing

```
...sort_stats('cumulative').print_stats(10)
```

– Functions that loop a lot and take much time

```
...sort_stats('time').print_stats(10)
```

– Functions that call a particular function "foo"

```
...print_callers('foo')
```

Summary

- Quick'n'Dirty timing with `time` and `timeit` modules
- Use `profile` & `pstats` modules to measure your code – not 100% accurate, but useful nonetheless
- `profile` does not measure external C modules (though it can be made to)
- Still: Gives excellent hints on where your code is spending it's time, but is not the "One Solution"
- Better to profile than not!

11. Testing your code

Testing Overview
`doctest`
Example
`unittest`
Summary

Testing overview

Python has built-in support for testing

– There are two modules, `doctest` and `unittest` that provide support for *unit testing*

Third party tools

– TextTest, by Geoff Bache, is an application-independent open source tool for text-based *functional testing*

– Written in Python :-)

– `http://www.texttest.org`

TextTest complements Python's built in tools

– In this course we will look at `doctest` and `unittest`

doctest – 1 of 6

– Doctest lets you create usage examples in your code's docstrings and checks that the examples work.

– It looks for the interactive Python prompt (`>>>`), the statement on that line, and the output of that statement on the following line

- `>>> print "hello"`
- `hello`

– When everything goes right, the test machinery exits silently, otherwise it reports which tests failed, expected output and actual output.

doctest – 2 of 6

```python
def adder(x,y):
    """
    >>> adder('2','g')
    '2g'
    """
    return x + y


if __name__ == '__main__':
    import doctest
    doctest.testmod()
```

```python
if __name__ == '__main__':
```

This snippet at the bottom is how you make sure to invoke the tests when the script is run standalone.

- This example will provide no output whatsoever.
- The tests are a bit small – we should have more extensive tests:

```
>>> adder(1L,2j)
>>> adder(1,'d')
>>> adder(2)
>>> adder(1,2,3)
>>> adder(1,2,[1,2])
```

... and so on...

doctest – 3 of 6

When something goes wrong

```
**********************************************************************
File "__main__", line 3, in __main__.adder
Failed example:
    adder('2','g')
Expected:
    '2xg'
Got:
    '2g'
**********************************************************************
1 items had failures:
   1 of   1 in __main__.adder
***Test Failed*** 1 failures.
```

```
1: def adder(x,y):
2:     """
3:     >>> adder('2','g')
4:     '2xg'
5:     """
6:     return x + y
```

doctest – 4 of 6

General Purpose unit testing with doctests

- The `unittest` module provides tools for general purpose unit testing, but it requires more work to set up

- Doctests are not originally meant for general purpose unit testing, still it is certainly possible to build quite large test suites with

- For larger tests using doctests, a good approach is to put your doc tests in a separate file and turn your doctests into unit tests.

doctest – 5 of 6

Building large test suites with doctest

- Suppose you are testing module `adder.py`

- In the same directory as `adder.py`, create a directory `test`

- Create a text file containing only your doctest text, call it `test_adder.txt`, save it in `test` dir

- Create your test runner as `test_adder.py` in the `test` dir and put the testing setup code in it.

doctest – 6 of 6

```python
# test_runner.py
import unittest
import doctest

def test_suite():
    tests = (doctest.DocFileSuite('test_adder.txt'),)
    return unittest.TestSuite(tests)

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

Interesting points of the above code

- `doctest.DocFileSuite` *converts* doctest tests from one or more text files to a `unittest.TestSuite`

- A *test suite* is a collection of test cases, test suites, or both. It is used to aggregate tests that should be executed together.

Example – 1 of 3

```
================
Adder tests
================

This tests a simple function, adder, that takes 2
arguments, x, y, and returns x + y

Let us make some preparations:

    >>> from adder import *

Now we have imported the adder function from the adder
module.
It is OK to use from X import * when doing doc tests.

It cannot be called without arguments

    >>> adder()
    Traceback (most recent call last):
    ...
    TypeError: adder() takes exactly 2 arguments (0 given)

It cannot be called with ony one argument

    >>> adder(1)
    Traceback (most recent call last):
      ...
    TypeError: adder() takes exactly 2 arguments (1 given)

It cannot be called with more than two arguments
```

```
>>> adder(1,3,5)
    Traceback (most recent call last):
      ...
    TypeError: adder() takes exactly 2 arguments (3
given)

It can add different number types

    >>> adder(1,2)
    3
    >>> adder(1L,2)
    3L
    >>> adder(1.2,3.4)
    4.5999999999999996
    >>> adder((1j+8), (2j+3))
    (11+3j)
    >>> adder(8L, 3J)
    (8+3j)

We cannot add nor concatenate numbers and other types.
We test only for strings.

    >>> adder(3,'3')
    Traceback (most recent call last):
    ...
    TypeError: unsupported operand type(s) for +: 'int'
and 'str'

However, adder can concatenate - unexpected! :-)

    >>> adder('c','3')
    'c3'
    >>> adder([23,45,67], [1,2,3])
    [23,45,67,1,2,3]

End of test.
```

Example – 2 of 3

```
======================================================================
FAIL: Doctest: test_adder.txt
----------------------------------------------------------------------
Traceback (most recent call last):
  File "C:\Program Files\Python2.4\lib\doctest.py", line 2157, in runTest
    raise self.failureException(self.format_failure(new.getvalue()))
AssertionError: Failed doctest test for test_adder.txt
  File "C:\...\academy\test_adder.txt", line 0

----------------------------------------------------------------------
File "C:\...\academy\test_adder.txt", line 60, in test_adder.txt
Failed example:
    adder([23,45,67], [1,2,3])
Expected:
    [23,45,67,1,2,3]
Got:
    [23, 45, 67, 1, 2, 3]


----------------------------------------------------------------------
Ran 1 test in 0.032s

FAILED (failures=1)
```

Example – 3 of 3

Results of a successful run of tests

```
----------------------------------------------------------------------
Ran 1 test in 0.015s

OK
```

`unittest` – 1 of 2

- The `unittest` module is the Python version of the xUnit unit-testing framework (JUnit for Java is another version).
- `unittest` code is put into a different source file than the code being tested – a module of its own.
- In your test module, import the module(s) you wish to test.
- Define one or more subclasses of `unittest.TestCase` and write your tests as methods of these subclasses – *test-case methods.*
- The methods names *have* to start with `test`
- Run the tests with a call to `unittest.main()`

unittest – 2 of 2

```python
import unittest
import adder

class AdderTest(unittest.TestCase):
    def testAddStrings(self):
        self.assertEqual(adder.adder('2','g'), '2g')

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

- Looking at the code, we see that the tests operates more or less the same as the doctests – we make a function call with known input and compare the results with some expected values with `assertEqual`
- The page `https://docs.python.org/2.6/library/unittest.html` has more detailed information of `unittest`

Summary

Testing is important – unit tests should be written while (at the same time as) you develop your code

- Python provides two frameworks for unit testing, **`unittest`** and **`doctest`**
- **`doctest`** provides a very easy way of documenting your code and testing it at the same time – test as you go
- **`unittest`** requires a bit more of setup but let's you control, group and manage your tests better
- It is possible to combine the two – **`docttest`** provides functionality to convert **`doctests`** to **`unittests`**

12. Built-in modules

`sys` – `os`
`time` & `datetime`
`subprocess` – `weakref`
`tempfile` – `getopt`
`tkinter`
Other modules of interest

Built-in modules

The Python language comes with many built-in modules

- We have already mentioned some modules such as `math`, `os`, `sys`, `itertools` and `copy`
- In this section we will *very briefly* glance at some useful modules: a bit more about `os`, and then modules `sys`, `subprocess`, `tempfile`, `getopt`, `time`, `datetime`, `weakref`, `tkinter`

sys module

The `sys` module handles system-related issues.

- `stdin, stdout, stderr`
- `maxint`
- `path`
- `modules`
- `getrefcount(object)`
- `argv`

`os` module – 1 of 2

`os` provides file, environment and process handling

Examples on file and environment handling

- `os.unlink(path)`
- `os.stat(path), os.access(path,mode)`
- `os.listdir(path)`
- `os.makedirs(path)`
- `os.path.exists(full_path)`
- `os.path.dirname(path)`
- `os.path.expandvars("$HOME/.cshrc")`
- `os.environ["CARMUSR"]`

os module – 2 of 2

```
>>> os.listdir(os.environ["CARMUSR"])
['CVS', 'CONFIG_extension', 'ADM', 'Resources',
'apc_scripts', 'bin', 'crc', 'crg', 'data', 'lib',
'matador_scripts', 'menu_scripts', 'packages',
'select_filter', 'preferences', 'images', 'macros',
'UPD_LOGG', 'current_carmsys']

>>> os.path.basename(os.environ["HOME"])
studentNN
```

time and datetime

- For manipulation of and calculactions with dates and times

- **Note:** The Jeppesen products provide specialised modules for this purpose too. You will normally use these instead of Python's:

  `AbsTime, RelTime, AbsDate, Dates`

Module `time` is nevertheless useful, for instance:

`t = time.time()` - get current time

`t = time.clock()` - measure CPU time

```
... <do something> ...
td = time.clock()-t
print "Time needed: %.4f" % td
```

`os` module – revisited

– The **`os`** module provides many easy and ready to use functions

```
ret = os.system(<unix-command>)
```

starts a shell command and waits until it is ready. Commands may be any unix executable binary or script:

```
popen* , spawn*, exec*
```

… but to get all possibilities you should use the **`subprocess`** module and the classes it provides...

`subprocess` – run subprocesses

Class `Popen`

When calling

- start with or without a shell
- change current directory
- set environment variables
- define functions to be executed before/after
- define `stdin, stderr, stdout`.

While running

- check status
- wait until ready
- write to `stdin`
- read from `stderr, stdout`

After termination

- access to `stderr, stdout, returncode`

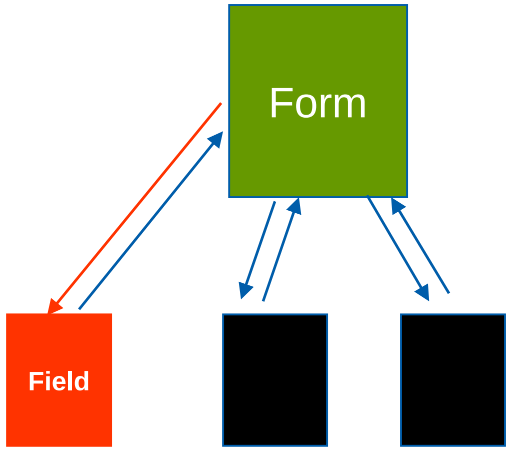
subprocess usage

```
>>> import subprocess as sp
>>> s = sp.Popen(["find",".","-name","*.py"],
                 stdout=sp.PIPE,
                 cwd="/carm/academy/NiceToHaveIQ");
>>> print s.wait()
0
>>> for row in s.stdout.xreadlines(): print row,
...
./MyReload.py
./PyIde.py
./RaveEvaluator.py
```

weakref module

Problem

- Sometimes, your design calls for objects that have to know of and reference each other. This leads to a *rerefence loop*.
- These objects keep each other alive, even after all other references to them are gone, and will consume memory even though not used anymore
- The CFH (*Carmen form handler*) has such a design, and it is important to avoid reference loops when using it.



weakref

Solution

- The `weakref` module provides you with a solution
- It lets objects reference each other, but they will *not* keep each other alive.
- Objects with weak references to each other can be garbage collected when all other objects forget about them.

```
>>> import weakref
>>> c = Counter()
>>> w = weakref.ref(c)
>>> w() is c
True
```

Counter class from earlier

The reference counter is not increased

`tempfile`

This module makes it easy and safe to create temporary files

- The only module function you probably need here is
  `f, path = mkstemp(suffix="", prefix="", dir=None)`

You should *not* use – these are deprecated

- `os.tempnam()`
- `os.tmpnam()`
- `tempfile.mktemp()`

tkinter

Lets you build a GUI

- BUT: Does not work well in the Studio process
- Instead you will use Cfh (Carmen Form Handler) or Wave Forms

Requires a Studio process

```
import Tkinter, sys
Tkinter.Label(text="Test of TkInter Gui").pack()
Tkinter.Button(text="Exit", command=sys.exit).pack()
Tkinter.mainloop()
```

Other modules of interest

`re`

- regular expressions

`pickle`

- store data structures on file

`array`

- as list, but for a single data type

`operator`

- operators as functions

`pwd`

- provides access to the Unix password database, on (Unix systems).
  See https://docs.python.org/2.6/library/pwd.html

Exercise 9

Now it is time for exercise 9

13. What's next?

PRT I & II courses

– The report generator
– The Rave API

Python in Studio course

– Examples of what you learn
  - Special > Rave > Rave Evaluator …
  - Special > Misc > Show Working Window Colours
  - Special > Search > for Objects by Rave expressions

Rave and Python for Rostering Optimization course

The end

Q / A

Evaluation

Welcome back to Jeppesen Training!