Brief introduction to Python
(with a Structural Bioinformatcs biass)

Jordi Villà i Freixa

Universitat de Vic - Universitat Central de Catalunya
Study Abroad

jordi.villa@uvic.cat

course 2023-2024

Disclaimer

This material was originally created by Michael A. Johnston and myself as the course material for a general introduction to Python at the MSc on Bioinformatics for Health Sciences at the Universitat Pompeu Fabra. The material has not been updated much since then, although the original syntax on Python 2.x has been ported to Python 3.x. However, it is likely that some deprecated material is still present here. I would appreciate if you let me know in case you detect some.

FACULTAT DE CIÈNCIES, TECNOLOGIA I ENGINYERIES
UVIC | UVIC-UCC

Programming

- Programs operate on various "data types": integers, strings, doubles
- Concept of variable and assignment:

```
Age = 3
```

- Expresions create and process data:

```
x>3
y=x*2
```

- Control of flow: conditioning testing (if, else) and iterations (for, while loops)
- Procedural programming: using functions to divide your program into logical chunks

Basic programming concepts! Only syntax change.

Python

- Dynamical, interpreted, object oriented programming language
- Software quality: designed to be readable, coherent and maintainable
- Developer productivity: very compact code (20-33% of the size of the corresponding java/C code): less code $\rightarrow$ less to debug $\rightarrow$ less to maintain $\rightarrow$ less to learn

Some help:
http://greenteapress.com/thinkpython/thinkpython.html

The subject

About making it quicker for you and others to write, maintain and extend programs. To do so:

- Reduce the time spent in programming & debugging: OOP, testing
- Make it easy to extend your program: code reuse (OOP)
- Reduce the time for others to understand your program: documentation, program readability

The Python interpreter

```
Python 2.6.6 (r266:84292, Jan  3 2011, 14:28:29)
[GCC 4.2.1 (Apple Inc. build 5664)] on darwin
Type "help", "copyright", "credits" or "license" for more info
>>> print (1 + 1)
2
```

Alternatively, you can store code in a file and use the interpreter to execute the contents of the file. Such a file is called a script. For example, you could use a text editor to create a fle named `dinsdale.py` with the following contents:

```
print (1 + 1)
```

By convention, Python scripts have names that end with .py.

IDLE or other IDEs

- IDLE is the Python Integrated Development Environment: http://docs.python.org/library/idle.html
  - First step in making it easier to write Python code
  - Syntax highlighting
  - Code completion
  - Inline documentation
  - Many other useful features
- Eclipse is... "an open extensible IDE for anything and nothing in particular". Extension to Python through PyDEV
- iPython aims... "to create a comprehensive environment for interactive and exploratory computing"
- but check also other possibilities: http://wiki.python.org/moin/PythonEditors

Structure of a program

- Programs are composed of *modules*
- Modules contain statements:
  - Function definitions
  - Control statements (if, while, etc)
  - Variable assignments
- Statements contain expressions:

```
x<3
a=x*x+2
```

- Expressions *create and process objects*

Python keywords

```
and       del     from    not     while
as        elif    global  or      with
assert    else    if      pass    yield
break     except  import  print
class     exec    in      raise
continue  finally is      return
def       for     lambda  try
```

Numbers

- Types: integer, floating point, long integers, bool (True, False)
- Basic expression operators & precedence http://www.ibiblio.org/g2swap/byteofpython/read/operator-precedence.html
- Conversion: mixed types are converted up, e.g., Integers → floating point

```
40+3.14
```

Dynamic typing

Variable types are decided at runtime

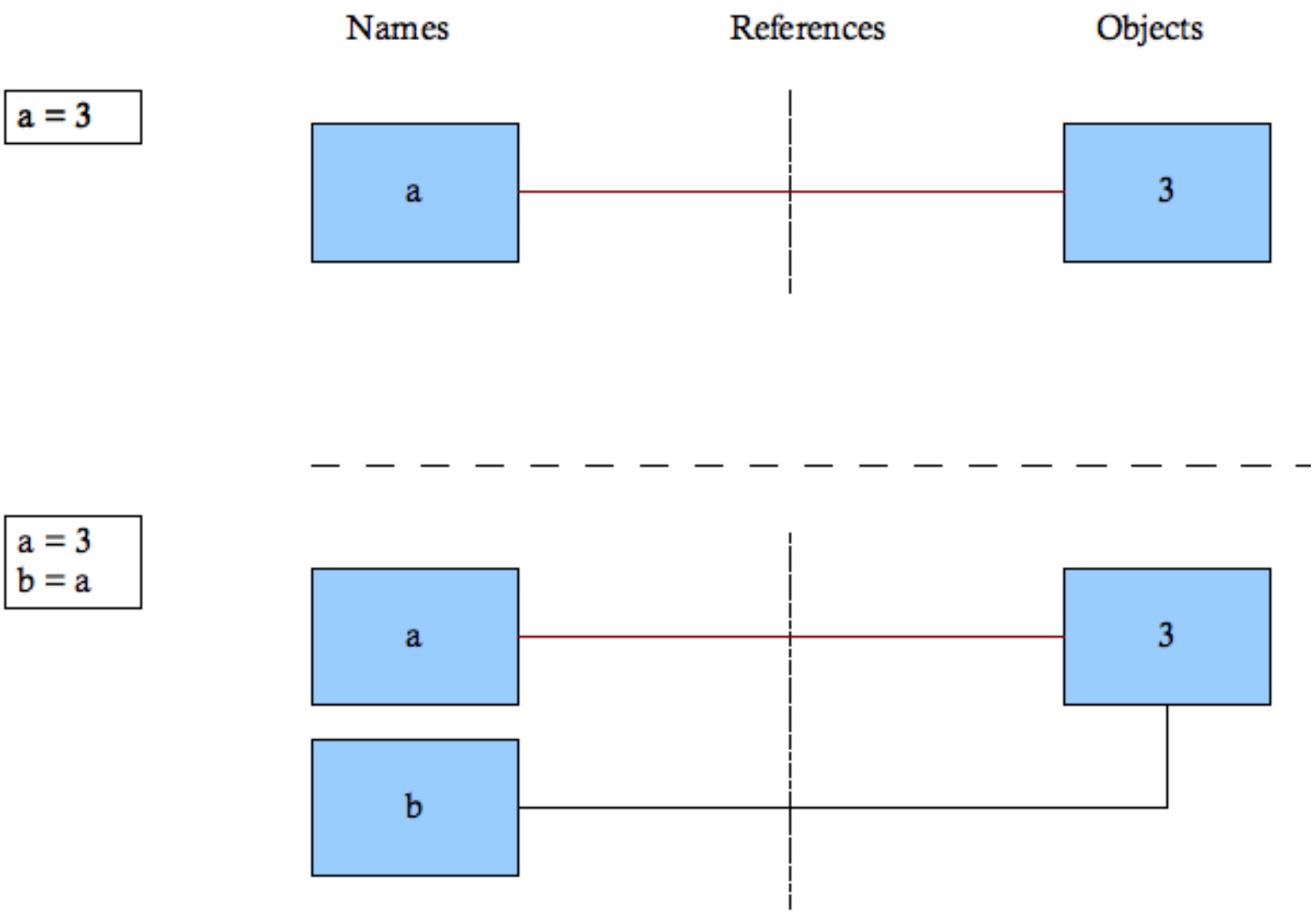
- Variables are created when you assign values to them
- Variables *refer* to an object, e.g., a number
- The object has a type; the variable does not
- When a variable appears in an expression, it is immediately replaced by the object it refers to

Example

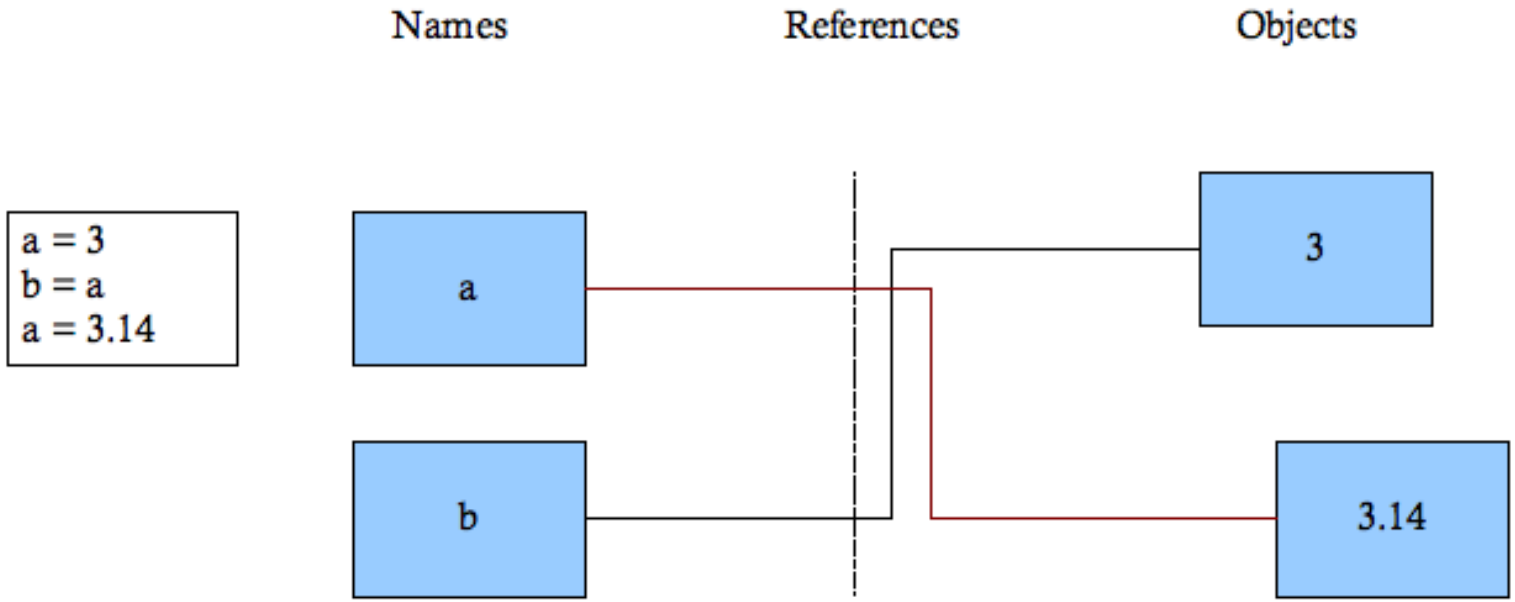
`a=3`

- Crerate an object of type integer that represents the number 3
- Create variable a if it does not exist yet
- link the variable a to the new object 3

Dynamic typing

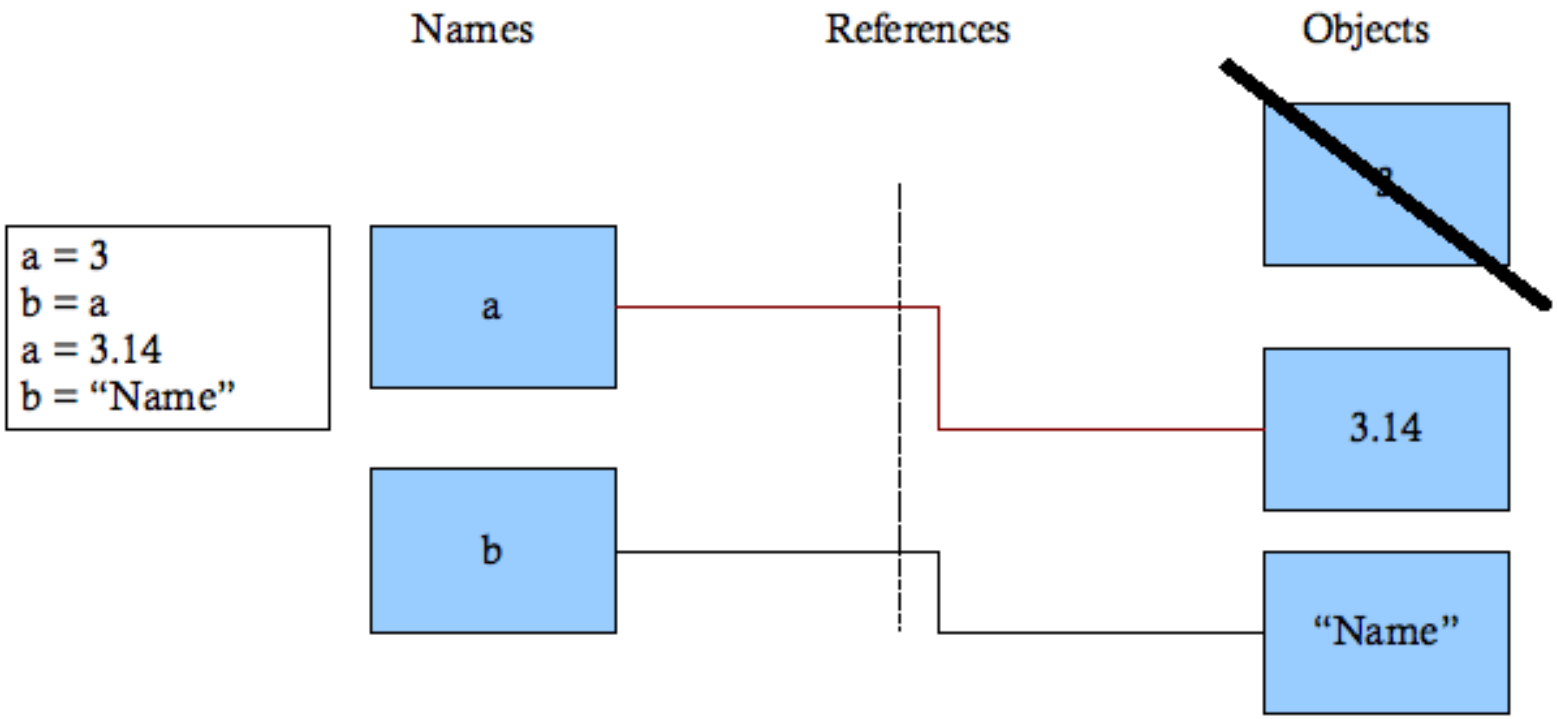


Dynamic typing



Garbage collection

When no variables are left that reference an object, it is destroyed (automatic memory management)



Modules

- Every file containing python code whose name ends in `.py` is a module
- A module usually contains a number of items e.g. Variables and functions which you can access. These items are called attributes
- You load a module using the import statement
- Just like a number a module is an object
- You can reload a module after changing it using the `reload()` function
- You access modules attributes using the `.` operator: `myModule.myAttribute`
- Modules are the highest level way of organising your program
- Large programs have multiple module files each of which contains related code

Documentation

- Documentation is one of the core parts of good programming
- Python contains an inbuilt documentation mechanism using "doc strings"
- For modules the doc string is the first string in the module file.
- Doc strings must be enclosed in triple quotes.
- A modules doc string is accessible through an attribute called `__doc__`

```
>>> import os
>>> os.access.__doc__
'access(path, mode) -> 1 if granted, 0 otherwise
Test for access to a file.'
>>>
```

More docstring examples

```
def phase_of_the_moon():
    "This function returns a slightly randomized
    integer that shuffles data around in a way
    convenient for the XYZ class."
    # Working code here.
    return value

def something():
    "This is a first paragraph.

    This is a second paragraph.  The intervening
    blank line means this must be a new paragraph."
    #   ...
```

Module attributes

- `__doc__` is one of four special module attributes
- The others are:
  - `__name__` - The module name
  - `__file__` - The modules file name (complete path)
  - `__builtin__` - Ignore for now.
- All special names in python begin and end with `__`
- You can see all the attributes of a module using the `dir()` function, which returns a list data type - more on lists later

`dir` returns a list of the attributes and methods of any object: modules, functions, strings, lists, dictionaries...

The import search path

```
>>> import sys
>>> sys.path
['', '/usr/local/lib/python2.2', '/usr/local/lib/pytho
'/usr/local/lib/python2.2/lib-dynload', '/usr/local/li
'/usr/local/lib/python2.2/site-packages/PIL',
'/usr/local/lib/python2.2/site-packages/piddle']
>>> sys
<module 'sys' (built-in)>
>>> sys.path.append('/my/new/path')

import sys, os
print ('sys.argv[0] =', sys.argv[0] )
pathname = os.path.dirname(sys.argv[0])
print ('path =', pathname)
print ('full path =', os.path.abspath(pathname))
```

FACULTAT DE CIÈNCIES, TECNOLOGIA I ENGINYERIES
UVIC | UVIC-UCC

Function basics

http://docs.python.org/library/functions.html

- We have already seen two functions - `reload()` & `dir()`
- Functions are defined using the `def` statement
- The `return` statement sends a functions result back to the caller.
- All code that is in the function must be indented
- The function ends when the indentation level is the same as the def statement that created it.
- The functions arguments are given in brackets after the name
  - Note you do not declare types in the argument list!
  - You can use any object as the arguments to a function: e.g. Numbers, modules and even other functions!

An example function

```
def mult(a, b):
    if b == 0:
        return 0
    rest = mult(a, b - 1)
    value = a + rest
    return value
print "3 * 2 = ", mult(3, 2)
```

Recursivity

Example

Write a function for the factorial of a number

Example

Write a function for counting down from a given integer

Factorial

```
def factorial(n):
    if n <= 1:
        return 1
    return n * factorial(n - 1)

print "2! =", factorial(2)
print "3! =", factorial(3)
print "4! =", factorial(4)
print "5! =", factorial(5)
```

Countdown

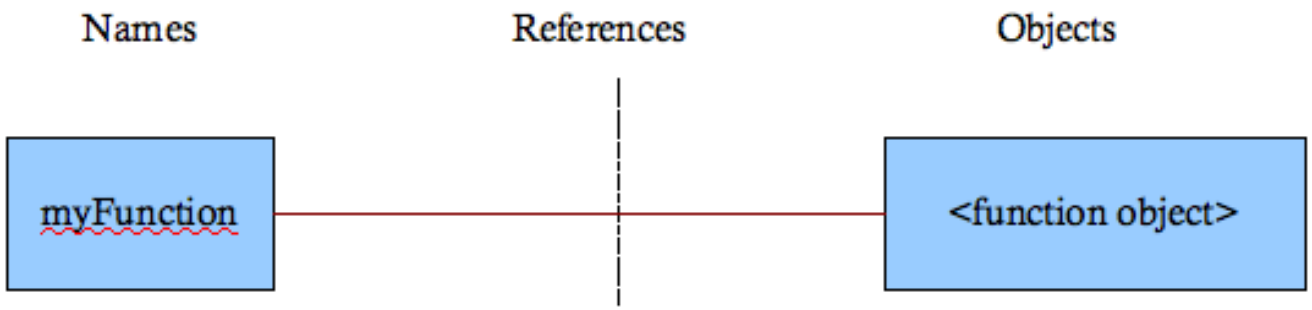
```
def count_down(n):
    print n
    if n > 0:
        return count_down(n-1)

count_down(5)
```

More on functions

- The function is not created until `def` is executed
- Like numbers and modules, functions are objects
- When `def` executes it creates a function object and associates a name with it.

```
def myFunction():
```



Argument values

```
def ask_ok(prompt, retries=4, complaint='Yes or no, pl
    while True:
        ok = raw_input(prompt)
        if ok in ('y', 'ye', 'yes'):
            return True
        if ok in ('n', 'no', 'nop', 'nope'):
            return False
        retries = retries - 1
        if retries < 0:
            raise IOError('refusenik user')
        print (complaint)
ask_ok('Do you really want to quit?')
ask_ok('OK to overwrite the file?', 2)
ask_ok('OK to overwrite the file?', 2, 'Come on, only
```

Lambda forms

Lambda forms can be used wherever function objects are required. They are syntactically restricted to a single expression.

```
>>> def make_incrementor(n):
...     return lambda x: x + n
...
>>> f = make_incrementor(42)
>>> f(0)
42
>>> f(1)
43
```

https://www.w3schools.com/python/python_lambda.asp

Function documentation

- Like modules functions can also have doc strings.
- The doc string is the first string after the function definition.
- It must be enclosed in triple quotes ”’ ”’.
- It is accessible through the attribute `__doc__`.

Objects and attributes

- In python everything is an object.
  - Numbers
  - Functions
  - Modules
- In python all objects have attributes
- The `dir()` function lists the attributes of any object
- Remember objects also have types
  - Functions are of type function
  - Integers have type int etc.
- Use the `type()` function to get an objects type.

Example

Create a module called `firstExercise.py`. Define the following functions and variables in the module:

- A function called `objectDocumention` which takes one argument and returns the doc string of the argument.
- A function called `objectName` which takes one argument and returns its `__name__` attribute.
- A function called `multiply(a, b)` which returns $a \times b$. Try passing objects other than numbers.
- A function called `integerMultiply(a,b)` which converts its arguments to integers before multiplying them. Hint: Use the function `int()` to convert objects to integers. Try with mixed numbers and strings

Load the module from the interactive shell and test it.

Example

Write a program (Python script) named `madlib.py`, which asks the user to enter a series of nouns, verbs, adjectives, adverbs, plural nouns, past tense verbs, etc., and then generates a paragraph which is syntactically correct but semantically ridiculous

Coercion

- Converting an object from one type to another is called coercion

```
>>> x=2
>>> y=3.
>>> coerce(x,y)
(2.0, 3.0)
```

- However not all objects can be coerced.
- When performing numeric operations the object with the smaller type is converted to the larger type.
- When using `and` or `or` the left hand operand is converted to a bool.
- The standard coercion functions for the types we have seen so far are `int()`, `float()`, `str()`, `bool()`

Bool conversions

- Any non-zero number or non-empty object converts to `True`
- A zero number or an empty object is `False`.

Operator overloading

- Operators that perform different actions depending on the types of the operands are said to be *overloaded*
- *
  - Multiplies when the operands are both numbers
  - Replicates when one is a number and the other a string
- +
  - Adds when the operands are both numbers
  - Concatenates when the operands are both strings.
- Many operators in python are overloaded.
- Notice that when the operands do not support the operator python raises an error. There is no point in checking your self.
- Also when the operators meaning is ambiguous an error is raised: using + with a string and a number - addition or concatenation?

Some other terminology

- Assigning an object to a name e.g. `a = 3`, `firstFunction = secondFunction`, is often called *binding*.
- Changing what a name refers to is called *rebinding*.
- `a = 3` Binds the name a to the object 3
- `a = "aString"` Rebinds the name a to the object "aString"

Strings

- A string is an ordered collection of characters.
- They are immutable i.e. They cannot be changed.
- You can create strings using
  - Double quotes - ""
  - Single quotes - ''
  - Triple quotes ''' ''' - i.e. Doc strings.
- Double and single quotes are the same
- Triple quotes create block strings which can span multiple lines.

```
hello = "This is a rather long string containing\n\
several lines of text just as you would do in C.\n\
    Note that whitespace at the beginning of the line
 significant."

print (hello)
```

Basic String Operations

- We've already seen * (replicate) and + (concatenate)
- Since strings are ordered collection of characters we can access their components by *indexing*
- The first character in the string has position 0.
- The position of the last character is equal to the number of characters in the string -1.
- `[]` is the index operator
  - `aString = "Genial"`
  - `aString[1]`
  - You can also index from the end using negative numbers
    - `aString[-1]` (This is the position = number of characters in the string -1)
    - `"Genial"` = length is 6
    - `"Genial"[-1]` is position `6 - 1 = 5` (`"l"`)

Slicing

- Slicing takes specified parts of a string and creates a new string.
- `[start:end]` Take from position start up to but not including position end
- `Astring[1:3]`
- If `start` is blank i.e. `[:end]`. It means from the first position
- If `end` is blank i.e. `[start:]`. It means go to the last position
- Extended slicing `[start:end:step]`
- `[1:10:2]` - Get the characters from 1 to 10 taking steps of 2.

String examples

```
>>> word = 'Help' + 'A'
>>> word
'HelpA'
>>> '<' + word*5 + '>'
'<HelpAHelpAHelpAHelpAHelpA>'

>>> 'str' 'ing'                    #  <-  This is ok
'string'
>>> 'str'.strip() + 'ing'    #  <-  This is ok
'string'
>>> 'str'.strip() 'ing'      #  <-  This is invalid
  File "<stdin>", line 1, in ?
    'str'.strip() 'ing'
                  ^
SyntaxError: invalid syntax
```

String examples

```
>>> word[0] = 'x'
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
TypeError: object does not support item assignment

>>> word[-2:]    # The last two characters
'pA'
>>> word[-100:]
'HelpA'
>>> word[-10]    # error
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
IndexError: string index out of range
```

H	e	l	p	A

String examples

```
string1 = "A, B, C, D, E, F"

print ("String is:", string1)
print ("Split string by spaces:", string1.split())
print ("Split string by commas:", string1.split( "," )
print ("Split string by commas, max 2:", string1.split
```

Removing leading and/or trailing characters in a string:

```
string1 = "\t  \n  This is a test string. \t\t \n"
print ('Original string: "%s"\n' % string1)
print ('Using strip: "%s"\n' % string1.strip())
print ('Using left strip: "%s"\n' % string1.lstrip())
print ('Using right strip: "%s"\n' % string1.rstrip())
```

Lists

- Lists contain ordered collections of any type of object: Numbers, strings, other lists.
- List Properties:
  - Mutable
    - Can change the object at any position
    - Can add and remove items from a list (more later)
  - Heterogenous
    - Can contain a mixture of data
- Creating a list
  - `myList = []`
  - `myList = [3, 4, "Jordi"]`
  - `myList = ["aString", [3, 4, "Jordi"]]`

List Operations

- A list like a string is a sequence. All the operators that work on strings work on lists (more *overloading*)
  - `*` (replication)
  - `+` (concatenation)
  - `[]` (indexing)
  - `[:]` (slicing)
- In addition a list is mutable - you can assign to list positions
  - Index assignment: `myList[3] = "Hello"`
  - Slice assignment: `MyList[0:3] = [0,1]` (Two steps: Deletion - the slice on the left is deleted; Insertion - the slice on the right is inserted in its place.

List Operations

- Trying to access a position that does not exist in a sequence is an error
- The function `len()` returns the number of items in a sequence.
- There are two more sequence operators
  - `x in sequence` evaluates as True if the object x is in the sequence or false if its not. e.g. `3 in [1,2,3]`, `"J" in "Jordi"`
  - `x not in sequence`, the opposite of in.

Examples with lists

```
>>> q = [2, 3]
>>> p = [1, q, 4]
>>> len(p)
3
>>> p[1]
[2, 3]
>>> p[1][0]
2
>>> p[1].append('xtra')
>>> p
[1, [2, 3, 'xtra'], 4]
>>> q
[2, 3, 'xtra']
```

Shallow vs Deep list copy

Shallow Copy: (copies chunks of memory from one location to another)

```
a = ['one','two','three']
b = a[:]
b[1] = 2
print (id(a), a #Output: 1077248300 ['one', 'two', 'th
print (id(b), b #Output: 1077248908 ['one', 2, 'three'
```

Deep Copy: (Copies object reference)

```
a = ['one','two','three']
b = a
b[1] = 2
print (id(a), a #Output: 1077248300 ['one', 2, 'three'
print (id(b), b #Output: 1077248300 ['one', 2, 'three'
```

FACULTAT DE CIÈNCIES, TECNOLOGIA I ENGINYERIES
UVIC | UVIC-UCC

The del statement

```
>>> a = [-1, 1, 66.25, 333, 333, 1234.5]
    >>> del a[0]
    >>> a
    [1, 66.25, 333, 333, 1234.5]
    >>> del a[2:4]
    >>> a
    [1, 66.25, 1234.5]
    >>> del a[:]
    >>> a
    []
```

if statement

```
if test1:
        <statements1>
elif test2:
        <statements2>
else:
        <statements3>
```

- All code that exists in the if statement must be indented (there are no braces etc.)
- Expression is any python expression that evaluates to a boolean i.e `True` or `False`

Example if statement

```
>>> x = int(raw_input("Please enter an integer: "))
Please enter an integer: 42
>>> if x < 0:
...         x = 0
...         print ('Negative changed to zero')
... elif x == 0:
...         print ('Zero')
... elif x == 1:
...         print ('Single')
... else:
...         print ('More')
...
More
```

While loops

```
while test:
        <statements>
```

- Repeatedly executes <statements> until test is true
- Example:

```
>>> # Fibonacci series:
... # the sum of two elements defines the next
... a, b = 0, 1
>>> while b < 10:
...     print (b)
...     a, b = b, a+b
...
1
1
2
3
```

for loop

```
for <target> in <object>:
        <statements>
```

- When python runs this loop it assigns the elements in `<object>`, one by one to the variable `<target>`
- Remember `<target>` is only a reference to an item in the sequence. Rebinding `<target>` does not change the item in the sequence.
- To change the elements of a list you need to use the `range()` function.

Example

try changing the characters of `"Peter"` to `"Roman"` by different methods (use while, for, ...)

for loop examples

```
>>> # Measure some strings:
... a = ['cat', 'window', 'defenestrate']
>>> for x in a:
...     print (x, len(x))
...
cat 3
window 6
defenestrate 12

>>> for x in a[:]: # make a slice copy of the entire l
...     if len(x) > 6: a.insert(0, x)
...
>>> a
['defenestrate', 'cat', 'window', 'defenestrate']
```

`a.insert(len(a), x)` is equivalent to `a.append(x)`

for loop examples

```
>>> a = ['Mary', 'had', 'a', 'little', 'lamb']
>>> for i in range(len(a)):
...         print (i, a[i])
...
0 Mary
1 had
2 a
3 little
4 lamb
```

Loop statements

- `break` Jumps out of the innermost loop. Use when you want a loop to end immediately due to some condition being reached
- `continue` Jumps to the top of the innermost loop. Use when you dont want to execute any more code for this iteration
- `pass` for empty loops
- `else` block, Executed if a loop was not exited due to a break statement

Some examples

```
>>> for n in range(2, 10):
...     for x in range(2, n):
...         if n % x == 0:
...             print (n, 'equals', x, '*', n/x)
...             break
...     else:
...         # loop fell through without finding a fact
...         print (n, 'is a prime number')
...
2 is a prime number
3 is a prime number
4 equals 2 * 2
5 is a prime number
6 equals 2 * 3
7 is a prime number
8 equals 2 * 4
```

List comprehensions

```
>>> li = [1, 9, 8, 4]
>>> [elem*2 for elem in li]
[2, 18, 16, 8]
>>> li
[1, 9, 8, 4]
>>> li = [elem*2 for elem in li]
>>> li
[2, 18, 16, 8]
```

look at it from right to left. li is the list you're mapping

```
>>> params = {"server":"mpilgrim", "database":"master"
>>> ["%s=%s" % (k, v) for k, v in params.items()]
['server=mpilgrim', 'uid=sa', 'database=master', 'pwd=
>>> ";".join(["%s=%s" % (k, v) for k, v in params.item
'server=mpilgrim;uid=sa;database=master;pwd=secret'
```

Examples list comphrehensions

```
>>> vec1 = [2, 4, 6]
>>> vec2 = [4, 3, -9]
>>> [x*y for x in vec1 for y in vec2]
[8, 6, -18, 16, 12, -36, 24, 18, -54]
>>> [x+y for x in vec1 for y in vec2]
[6, 5, -7, 8, 7, -5, 10, 9, -3]
>>> [vec1[i]*vec2[i] for i in range(len(vec1))]
[8, 12, -54]
>>> [str(round(355/113.0, i)) for i in range(1,6)]
['3.1', '3.14', '3.142', '3.1416', '3.14159']
```

Files

- The file object in python represents a file that you can read from and write to
- Unlike the other python objects you can not use operators on them e.g. `+`, `*`, `[]` etc.
- Creation

```
myFile = open(location)
```

- Some Methods
  - `read()`
  - `readline()`
  - `readlines()`
  - `write()`
  - `writelines()`
  - `close()`

File manipulation examples

http://docs.python.org/library/stdtypes.html?highlight=tell#file.tell
http://docs.python.org/tutorial/inputoutput.html

```
fileHandle = open ( 'test.txt', 'w' )
fileHandle.write ( 'Testing files in Python.\neasily'
fileHandle.close()
fileHandle = open ( 'test.txt', 'a' )
fileHandle.write ( '\n\n\nBottom line.' )
fileHandle.close()
fileHandle = open ( 'test.txt' )
print (fileHandle.read())
fileHandle.close()
```

File manipulation examples

```
fileHandle = open ( 'test.txt' )
print (fileHandle.readline())
print (fileHandle.tell()) # position within the file
print (fileHandle.readline())
fileHandle = open ( 'test.txt' )
print (fileHandle.read ( 1 ))
fileHandle.seek ( 4 )
print (FileHandle.read ( 1 ) )
fileHandle = open ( 'testBinary.txt', 'wb' )
fileHandle.write ( 'There is no spoon.' )
fileHandle.close()
fileHandle = open ( 'testBinary.txt', 'rb' )
print (fileHandle.read())
fileHandle.close()
```

More sophisticated file manipulation

http://docs.python.org/library/glob.html

```
import os, glob, shutil
file_ext = raw_input("Extension for the files:\n")
file_count = raw_input("Files count in each new dir:\n
file_count = int(file_count)
dir_base_name = raw_input("name base for dirs:\n")
filenames = glob.glob(('*.' + file_ext))
filenames.sort()
dir_number = 0
while filenames:
    dir_number += 1
    new_dir = dir_base_name + str(dir_number)
    os.mkdir(new_dir)
    for n in range(min(file_count, len(filenames))):
        src_file = filenames.pop(0)
        shutil.copy(src_file, new_dir)
```

Methods

- We have seen that everything in python is an object and that all objects have attributes. The attributes can have different types e.g string, int, function
- Another type of attribute an object can have is called a *method*
- An objects methods are special functions that operate on the object itself.
- invoked with `object.method()` the method does something with object
- Some objects like modules have no methods or very rarely used methods e.g. Functions and numbers.
- Lists and strings have many very commonly used methods.

Example: String methods

- Here are some string methods
  - `capitalize`
  - `count`
  - `find`
  - `index`
  - `split`
- Some methods take arguments, others don't.
- Check `https://docs.python.org/3/library/string.html` for a full description of the string methods.
- Check `https://docs.python.org/3/tutorial/datastructures.html#more-on-lists` for a description of list methods.

Object attributes

- We have seen that objects can have many attributes and that all attributes are objects. (Remember `dir()`)
- Generally an object's attributes are divided into two types
  - Callable - They can perform some action and return a result: Functions, methods
  - Not callable - Everything else (strings, lists, numbers etc.)
- You can check if an object is callable using the `callable()` function.
- Another useful function is `getattr()`
- `getattr()` returns an attribute of an object if you know its name as a string.

```
>>> li = ["Larry", "Curly"]
>>> getattr(li, "pop")
<built-in method pop of list object at 010DF884>
>>> value = obj.attribute
>>> value = getattr(obj, "attribute")
```

FACULTAT DE CIÈNCIES, TECNOLOGIA I ENGINYERIES
UVIC | UVIC-UCC

Augmented assignment

- Based on C
- Short hand for writing common expressions
  - Traditional: `X = X + Y`
  - Augmented: `X += Y`
  - `X *= Y`, `X -=Y`, `X /= Y` etc.
- Less typing
- Automatically chooses optimal method
  - `L = L + [3,4]`
  - `L.extend([3,4])`
  - `L += [3,4]` - Automatically chooses extend

String formatting

%

- Format operator.
- You place a string to the right of the operator with conversion targets embedded in it.
- A conversion target is a % followed by a letter. The letter indicates the conversion to be performed
- On the right of the format operator you place, in parentheses, one object for each conversion target in the string.
- Python inserts each object into the string, the first at the first conversion target etc, performing the necessary conversion first.
- `"Name %s. Age %d" % ("Joe", 52)`

Extended formatting

- Since all basic objects in python have a string description usually `%s` is all thats needed
- However with numbers more control is often required.
- `%d`, `%e`, `%E`, `%f`
- Extended formatting syntax
  - `%[flags][width][.precision]code`
  - Flags
    - `-` left justify
    - `+` add plus for positive numbers
    - `0` pad with zeros
  - Width is the maximum width the conversion can have
  - `.precision` is the number of places after the decimal point.

String formatting vs. concatenating

```
>>> uid = "sa"
>>> pwd = "secret"
>>> print (pwd + " is not a good password for " + uid
)
secret is not a good password for sa
>>> print ("%s is not a good password for %s" % (pwd,
secret is not a good password for sa
>>> userCount = 6
>>> print ("Users connected: %d" % (userCount, ))
Users connected: 6
>>> print ("Users connected: " + userCount)
Traceback (innermost last):
  File "<interactive input>", line 1, in ?
TypeError: cannot concatenate 'str' and 'int' objects
```

See also http://docs.python.org/tutorial/inputoutput.html

Tuples

- A tuple is an immutable list with no methods
  - Ordered collection of arbitrary objects
  - Creation
    - `()` e.g. `(3, "Name")`
    - `,` e.g. `3, "Name"` (Not advisable)
    - A tuple with a single element is a special case: `(40,)` - require a trailing comma
  - Can be operated on by all the immutable sequence operators
    - `*`, `+`, `[]`, `[:]`, in
  - Accessed by position starting from 0
  - Use `len()` to get length of a tuple
- Note than only the tuple is immutable. Mutable objects in a tuple are still mutable.
- Tuples provide integrity (one needs to be sure that something cannot be changed)

Using tuples to assign values

```
>>> v = ('a', 'b', 'e')
>>> (x, y, z) = v
>>> x
'a'
>>> y
'b'
>>> z
'e'
```

`v` is a tuple of three elements, and `(x, y, z)` is a tuple of three variables.

Sequence conversion

- Like `int()`, `float()` etc. there are functions for converting objects to lists & tuples.
- `list()`
- `tuple()`
- These functions can only coerce objects that are also sequences i.e. strings, lists, tuples
- `list(3)` - will not work
- `list("3")` - will work

Sequence functions

- `filter()`
  - Filters the elements of a sequence based on a function and produces a new sequence
- `map()`
  - Applies a function to every element of a sequence and returns a list of the results.
  - Can be used with multiple lists
- `reduce()`
  - Applies a function to the items of a sequence from left to right to reduce the list to a single value.
  - Calls the function using the first two values of the sequence. Then on the result and the third item etc.
- `zip()`
  - Takes any number of lists as arguments
  - Returns a list of tuples where the first contains the first element of each sequence, the second the second element of each etc.

```
>>> foo = [2, 18, 9, 22, 17, 24, 8, 12, 27]
>>>
>>> print (filter(lambda x: x % 3 == 0, foo))
[18, 9, 24, 12, 27]
>>>
>>> print (map(lambda x: x * 2 + 10, foo))
[14, 46, 28, 54, 44, 58, 26, 34, 64]
>>>
>>> print (reduce(lambda x, y: x + y, foo))
139
```

Example of the use of `filter`

```
>>> def odd(n):
...     return n%2
...
>>> li = [1, 2, 3, 5, 9, 10, 256, -3]
>>> filter(odd, li)
[1, 3, 5, 9, -3]
>>> filteredList = []
>>> for n in li:
...     if odd(n):
...         filteredList.append(n)
...
>>> filteredList
[1, 3, 5, 9, -3]
```

- odd returns 1 if n is odd and 0 if n is even.
- `filter` takes two arguments, a function (odd) and a list (li). It loops through the list and calls odd per element

Example of the use of zip

```
>>> mat = [
...         [1, 2, 3],
...         [4, 5, 6],
...         [7, 8, 9],
...       ]
>>> zip(*mat)
[(1, 4, 7), (2, 5, 8), (3, 6, 9)]

names = ["Jesus","Marc","Michal","Graham"]
places = ["Spain","USA","Poland","UK"]
combo = zip(names,places)
who = dict(combo)
```

Examples of the use of map

```
>>> print (map(lambda w: len(w),
              'It is raining cats and dogs'.split()))
[2, 2, 7, 4, 3, 4]

>>>map(f, sequence)
>>>[f(x) for x in sequence]

>>>map(f, sequence1, sequence2)
>>>[f(x1, x2) for x1, x2 in zip(sequence1, sequence2)]
```

Exercises

Example

Write a code that computes the prime numbers up to 50 (hint: use the filter function)

Example

Write a code that writes a value table $(x, f(x))$ for $f(x) = sin(x)$ (hint: use the map function)

Example

Write a code that calculates the geometric mean of a given list of values (hint: use the reduce function)

Dictionaries

- Dictionaries are mappings
  - Unordered collection of objects (Python 3 includes order)
  - Access items via a key (case sensitive)
  - Equivalent to hashes in perl
  - Very fast retrieval
  - Mutable
- Creation
  - `{}` - an empty dictionary
  - `{'age': 40, 'name': "unknown"}`

Example dictionaries

```
>>> d = {"server":"mpilgrim", "database":"master"}
>>> d
{'server': 'mpilgrim', 'database': 'master'}
>>> d["server"]
'mpilgrim'
>>> d["database"]
'master'
>>> d["database"] = "pubs"
>>> d
{'server': 'mpilgrim', 'database': 'pubs'}
>>> d["uid"] = "sa"
>>> d
{'server': 'mpilgrim', 'uid': 'sa', 'database': 'pubs'
>>> del d['uid']
>>> d["mpilgrim"]
Traceback (innermost last):
```

Dictionary operations

- Accessing
  - Dict[key]
  - `len()` - Returns the number of stored entries
- Assignment
  - `Dict[key] = object`
- Removal
  - `del Dict[key]`
  - The del statement can be used with lists or attributes etc.
- Construction
  - `dict(zip(keys, values))`

Dictionary methods

- `has_key()`
- `keys()`
- `values()`
- `copy()` ...

Note on function arguments

```
>>> range(3, 6)                 # normal call with separat
[3, 4, 5]
>>> args = [3, 6]
>>> range(*args)                # call with arguments unpa
[3, 4, 5]
```

```
def cheeseshop(kind, *arguments, **keywords):
    print ("-- Do you have any", kind, "?")
    print ("-- I'm sorry, we're all out of", kind)
    for arg in arguments: print arg
    print ("-" * 40)
    keys = keywords.keys()
    keys.sort()
    for kw in keys: print (kw, ":", keywords[kw])
cheeseshop("Limburger", "It's very runny, sir.",
           "It's really very, VERY runny, sir.",
```

Naming convention

- docstrings: http://www.python.org/dev/peps/pep-0257/, and general text: http://www.python.org/dev/peps/pep-0008/.
- Function names should describe what the function does.
  - The more general the better though there is a balance.
  - Name should be enough to give an idea of what it does.
  - General does not mean short! Use full words
- Arguments names should be as general as possible. object, aString, aFunction, comparisonFunction.
- A variabe name should describe what it is.
  - Use full words.
  - You should not use reserved words (see page 10).
  - Names beginning and ending in two __ are system defined names and have a special meaning for the interpreter

finding substrings

```
>>> dna = """ttcacctagtctaggacccactaatgcagatcctgtg
tgtctagctaagatgtattatatctatattcactgggcttattgggccaa
tgaaaatatgcaagaaaggaaaaaaaaagatgtagacaaggaattctattt"""
>>> E='gat'
>>> dna.find(E)
48
>>> dna.index(E)
48
```

Try looking for a non-existing substring with both methods

Example

Write a function that returns the list of codons for a DNA sequence and a given frame

First view at regular expressions

https://docs.python.org/3/howto/regex.html

```
>>> import re
>>> m = re.search('(?<=abc)def', 'abcdef')
>>> m.group(0)
'def'
>>> m = re.search('(?<=-)\w+', 'spam-egg')
>>> m.group(0)
'egg'
>>> m = re.match(r"(\w+) (\w+)", "Isaac Newton, physic
>>> m.group(0)       # The entire match
'Isaac Newton'
>>> m.group(1)       # The first parenthesized subgrou
'Isaac'
>>> m.group(2)       # The second parenthesized subgro
'Newton'
>>> m.group(1, 2)    # Multiple arguments give us a tu
```

Writing regex

compile() Compile a regular expression pattern into a regular expression object, which can be used for matching using its `match()` and `search()` methods

search() Scan through string looking for a location where the regular expression pattern produces a match, and return a corresponding MatchObject instance.

match() If zero or more characters at the beginning of string match the regular expression pattern, return a corresponding MatchObject instance

split() Split string by the occurrences of pattern

`http://docs.python.org/dev/howto/regex.html`

Regular expressions

A regular expression is a pattern that a string is searched for. Unix commands such as "rm *.*" are similar to regular expressions, but the syntax of regular expressions is more elaborated. Several Unix programs (grep, sed, awk, ed, vi, emacs) use regular expressions and many modern programming languages (such as Java) also support them. In Python, a regular expression is first compiled:

```
keyword = re.compile(r"the ")
keyword.search(line)
not keyword.search(line)
keyword = re.compile(variable)
keyword = re.compile(r"the ",re.I) #for insensitive se
```

re.finditer()

```
import re
import urllib2

html = urllib2.urlopen('http://cbbl.imim.es').read()
pattern = r'\b(the\s+\w+)\s+'
regex = re.compile(pattern, re.IGNORECASE)
for match in regex.finditer(html):
    print ("%s: %s" % (match.start(), match.group(1)))
```

Example

Given a string of A, C, T, and G, and X, find a string where X matches any single character, e.g., CATGG is contained in ACTGGGXXAXGGTTT.

Example

Write a regular expression to extract the coding sequence from a DNA string. It starts with the ATG codon and ends with a stop codon (TAA, TAG, or TGA).

Regular expressions

```
>>> import re
>>> re.findall(r'\bf[a-z]*', 'which foot or hand fell
['foot', 'fell', 'fastest']
>>> re.sub(r'(\b[a-z]+) \1', r'\1', 'cat in the the ha
'cat in the hat'
>>> 'tea for too'.replace('too', 'two')
'tea for two'
```

Regular expressions

```
#!/usr/bin/env python
import re

# open a file
file = open("alice.txt","r")
text = file.readlines()
file.close()

# compiling the regular expression:
keyword = re.compile(r"the ")

# searching the file content line by line:
for line in text:
    if keyword.search (line):
        print (line,)
```

Regular expressions

```
#!/usr/bin/env python
import re

# open a file
file = open("alice.txt","r")
text = file.readlines()
file.close()

# searching the file content line by line:
keyword = re.compile(r"the ")

for line in text:
    result = keyword.search (line)
    if result:
        print (result.group(), ":", line,)
```

http://docs.python.org/library/re.html

Write scripts that

Example

Retrieve all lines from a given file that do not contain "the ". Retrieve all lines that contain "the " with lower or upper case letters (hint: use the ignore case option)

Example

Retrieve lines from a long sequence (eg, CFTR) that contain a given codon, and then a given first and third letter for each triad

http://www.upriss.org.uk/python/session7.html#chars

Example

Write a script that asks users for their name, address and phone number. Test each input for accuracy, for example, there should be no letters in a phone number. A phone number should have a certain length. An address should have a certain format, etc. Ask the user to repeat the input in case your script identfies it as incorrect.

Classes: Some defs

Namespace mapping from names to objects. There is absolutely norelation between names in different namescapes (different local names in a function invocation, for example; that is why we prefix them with the name of the function, for example).

Scope textual region of a Python program where a namespace is directly accessible.

Attributes anything you can call in the form:`object.attribute` (data and methods).

Instance objects created by *instantiation* of classes.

`http://docs.python.org/tutorial/classes.html`

Global vs local variables

```
#!/usr/local/bin/python
"""http://www.wellho.net/resources/ex.php4?item=y105/l
# Variable scope

first = 1

def one():
        "Double a global variable, return it + 3."
        global first
        first *= 2
        result = first+3
        return result

print one.__doc__
print one()
print one()
```

A first example of a class

```
#! /usr/bin/python
"""house.py -- A house program.  """

class House(object):
    """Some stuff """
my_house = House() # class instantiation
my_house.number = 40 # data attribute
my_house.rooms = 8
my_house.garden = 1

print "My house is number", my_house.number
print "It has", my_house.rooms, "rooms"
if my_house.garden:
        garden_text = "has"
else:
        garden_text = "does not have"
```

A second example of a class

```
#! /usr/bin/python
"""house2.py -- Another house.
"""

class House(object):
        def __init__(self, number, rooms, garden):
                self.number = number
                self.rooms = rooms
                self.garden = garden

my_house = House(20, 1, 0)

print "My house is number", my_house.number
print "It has", my_house.rooms, "rooms"
if my_house.garden:
        garden_text = "has"
```

Adding methods

```
#! /usr/bin/python
"""square.py -- Make some noise about a square.
"""

class Square:
        def __init__(self, length, width):
                self.length = length
                self.width = width

        def area(self):
                return self.length * self.width

my_square = Square(5, 2)
print my_square.area()
```

http://www.ibiblio.org/g2swap/byteofpython/read/oops.html

Some terminology

- A class creates a new type where objects are instances of the class.
- The 'functions' that are part of an object are called methods.
- The fields and methods are called 'attributes'.
- You can examine all the methods and attributes that are associated with an object using the dir command : `print (dir(some_obj))`
- Fields are of two types - they can belong to each instance/object of the class or they can belong to the class itself. They are called instance variables and class variables respectively.

Arrays and classes

```
#! /usr/bin/python
"""person.py -- A person example.
"""
class Person(object):
        def __init__(self, age, house_number):
                self.age = age
                self.house_number = house_number

alex = []
for i in range(5):
        obj = Person(i, i)
        alex.append(obj)

print "Alex[3] age is", alex[3].age
print
```

Examples

Example

Write a simple program that reads from a CSV file containing a list of names, addresses, and ages and returns the name, address and age for a particular person upon request.

Example

Extend the above program to include e-mail addresses and phone numbers to the student's data. (Hint (in Python 2.7 syntax!): http://www.upriss.org.uk/python/session13.html)

Syntax errors

http://docs.python.org/tutorial/errors.html

```
>>> while True print ('Hello world')
  File "<stdin>", line 1, in ?
    while True print ('Hello world')
                   ^
SyntaxError: invalid syntax
```

Syntax errors

```
>>> 10 * (1/0)
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
ZeroDivisionError: integer division or modulo by zero
>>> 4 + spam*3
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
NameError: name 'spam' is not defined
>>> '2' + 2
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
TypeError: cannot concatenate 'str' and 'int' objects
```

Handling exceptions

```
#!/usr/bin/env python
#
# Program to read and print a file

import sys

try:
    file = open("alice.txt","r")
except IOError:
    print ("Could not open file")
    sys.exit()

text = file.readlines()
file.close()

for line in text:
```

Exceptions

```
... except (RuntimeError, TypeError, NameError):
...     pass

>>> def this_fails():
...     x = 1/0
...
>>> try:
...     this_fails()
... except ZeroDivisionError as detail:
...     print ('Handling run-time error:', detail)
...
Handling run-time error: integer division or modulo by
```

A useful case

```
import getopt, sys

def main():
    try:
        opts, args = getopt.getopt(sys.argv[1:], "ho:", ["h
    except getopt.GetoptError:
        # print help information and exit:
        usage()
        sys.exit(2)
    output = None
    for o, a in opts:
        if o in ("-h", "--help"):
            usage()
            sys.exit()
        if o in ("-o", "--output"):
            output = a
    #...
if __name__ == "__main__":
```

Exceptions

```
>>> def divide(x, y):
...     try:
...         result = x / y
...     except ZeroDivisionError:
...         print ("division by zero!")
...     else:
...         print ("result is", result)
...     finally:
...         print ("executing finally clause")
...
>>> divide(2, 1)
result is 2
executing finally clause
>>> divide(2, 0)
division by zero!
executing finally clause
```

User defined exceptions

```
>>> class MyError(Exception):
...     def __init__(self, value):
...         self.value = value
...     def __str__(self):
...         return repr(self.value)
...
>>> try:
...     raise MyError(2*2)
... except MyError as e:
...     print ('My exception occurred, value:', e.valu
...
My exception occurred, value: 4
>>> raise MyError('oops!')
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
__main__.MyError: 'oops!'
```

User defined exceptions

```
class Error(Exception):
    """Base class for exceptions in this module."""
    pass

class InputError(Error):
    """Exception raised for errors in the input.

    Attributes:
        expr -- input expression in which the error oc
        msg  -- explanation of the error
    """

    def __init__(self, expr, msg):
        self.expr = expr
        self.msg = msg
```

BioPython

Set of modules and packages for biology (sequence analysis, database access, parsers...):
http://biopython.org/DIST/docs/tutorial/Tutorial.html
http://biopython.org/DIST/docs/api/

Examples

```
>>> from Bio.Seq import Seq
>>> my_seq = Seq("AGTACACTGGT")
>>> my_seq
Seq('AGTACACTGGT', Alphabet())
>>> print (my_seq)
AGTACACTGGT
>>> my_seq.alphabet
Alphabet()
>>> my_seq.complement()
Seq('TCATGTGACCA', Alphabet())
>>> my_seq.reverse_complement()
Seq('ACCAGTGTACT', Alphabet())
```

A couple of simple exercises

Example

Search for CFTR nucleotide sequences in the NCBI server. Save the sequences as FASTA and GeneBank. Using the SeqIO parser extract the sequences from the files and print them on screen.

http://biopython.org/DIST/docs/api/Bio.SeqIO-module.html#parse

Example

Download an alignment for the CFTR protein entries from PFAM (use the seed for ABC transporters). Using the AlignIO parser, extract the sequences from FASTA or Stocholm formatted files downloaded from PFAM.

http://biopython.org/DIST/docs/api/Bio.AlignIO-module.html#parse

```
from Bio.Align.Generic import Alignment
from Bio.Alphabet import IUPAC, Gapped
alphabet = Gapped(IUPAC.unambiguous_dna)

align1 = Alignment(alphabet)
align1.add_sequence("Alpha", "ACTGCTAGCTAG")
align1.add_sequence("Beta",  "ACT-CTAGCTAG")
align1.add_sequence("Gamma", "ACTGCTAGDTAG")

align2 = Alignment(alphabet)
align2.add_sequence("Delta",  "GTCAGC-AG")
align2.add_sequence("Epislon","GACAGCTAG")
align2.add_sequence("Zeta",   "GTCAGCTAG")

my_alignments = [align1, align2]
```

See, better, MultipleSeqAlignment

Converting between sequence alignment formats

```
from Bio import AlignIO
count = AlignIO.convert("PF05371_seed.sth", "stockholm
                        "PF05371_seed.aln", "clustal")
print ("Converted %i alignments" % count)
```

```
from Bio import AlignIO
alignments = AlignIO.parse(open("PF05371_seed.sth"),
                           "stockholm")
handle = open("PF05371_seed.aln","w")
count = AlignIO.write(alignments, handle, "clustal")
handle.close()
print ("Converted %i alignments" % count)
```

```
from Bio import AlignIO
alignment = AlignIO.read(open("PF05371_seed.sth"),
                         "stockholm")
print (alignment.format("clustal"))
```

Performing alignments

BioPython provides tools for command line execution. For example:

```
>>> import os
>>> import subprocess
>>> from Bio.Align.Applications import ClustalwCommand
>>> help(ClustalwCommandline)
>>> c_exe = r"/Applications/clustalw2"
>>> assert os.path.isfile(c_exe), "Clustal W missing"
>>> cl = ClustalwCommandline(c_exe, infile="cftr.fasta
>>> return_code = subprocess.call(str(cl),
...                               stdout = open(os.devnull),
...                               stderr = open(os.devnull),
...                               shell=(sys.platform!="win32"
```

http://docs.python.org/library/subprocess.html

http://jimmyg.org/blog/2009/working-with-python-subprocess.html

FACULTAT DE CIÈNCIES, TECNOLOGIA I ENGINYERIES
UVIC | UVIC-UCC

Working with streams and subprocesses

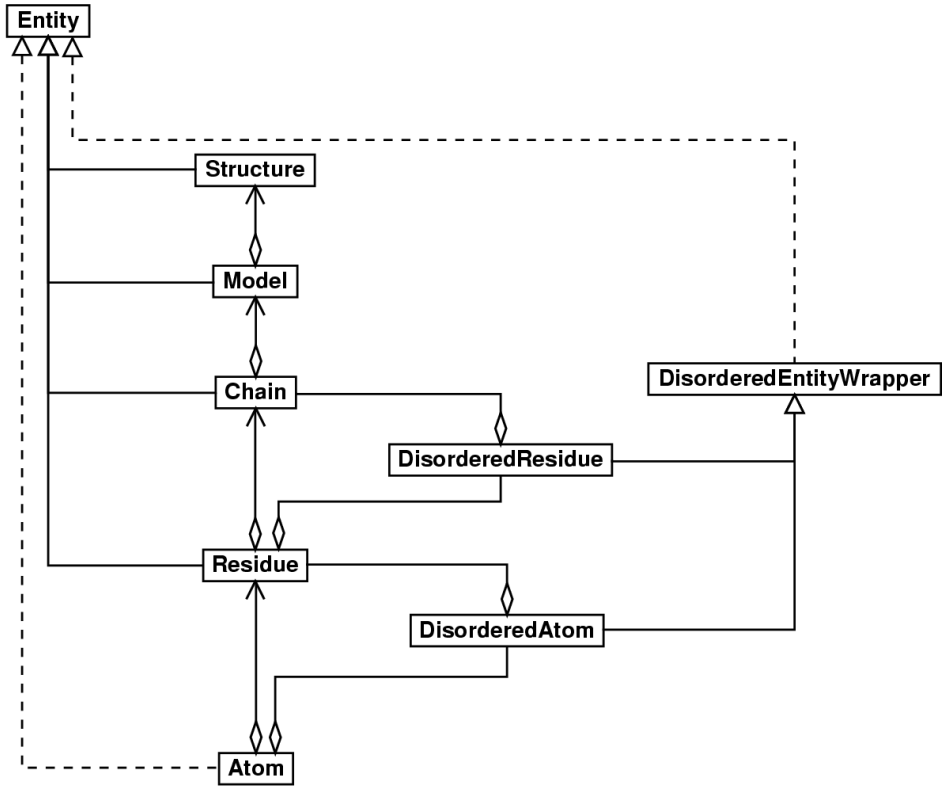
```
import sys
while 1:
    try:
        input = sys.stdin.readline()
        if input:
            sys.stdout.write('Echo to stdout: %s'%inpu
            sys.stderr.write('Echo to stderr: %s'%inpu
    except KeyboardError:
        sys.exit()
```

```
>>> subprocess.Popen('echo $PWD', shell=True)
/home/james/Desktop
```

```
>>> subprocess.Popen("""
... cat << EOF > new.txt
... Hello World!
... EOF
```

Dealing with PDB files

http://www.biopython.org/DIST/docs/tutorial/Tutorial.html#htoc133



PDB parsing example

```
>>> from Bio.PDB.PDBParser import PDBParser
>>> parser=PDBParser()
>>> structure=parser.get_structure("test","1WQ1.pdb")
>>> structure.get_list()
[<Model id=0>]
>>> model=structure[0]
>>> model.get_list()
[<Chain id=R>, <Chain id=G>]
>>> chain=model["R"]
>>> chain.get_list()
[<Residue MET het=  resseq=1 icode= >, <Residue THR het=  resseq=2 icode=
resseq=3 icode= >, <Residue TYR het=  resseq=4 icode= >, <Residue LYS het=
resseq=5 icode= >
```

Retrieving a PDB file

```
>>> from Bio.PDB import PDBList
>>> pdbl=PDBList()
>>> pdbl.retrieve_pdb_file('5P21')
retrieving ftp://ftp.wwpdb.org/pub/pdb/data/structures
'/Users/jordivilla/merda/p2/pdb5p21.ent'
```

http://www.biopython.org/DIST/docs/cookbook/biopdb_faq.pdf

or:

```
import urllib
def fetch_pdb(id):
  url = 'http://www.rcsb.org/pdb/files/%s.pdb' % id
  return urllib.urlopen(url).read()
```

Plotting with Python

Matplotlib is the reference tool for plotting 2D data in Python. iPython has a "pylab" mode specific for interacting with matplotlib.
http://wiki.python.org/moin/NumericAndScientific/Plotting
https://matplotlib.org/stable/tutorials/index.html

```
>>> from pylab import randn, hist
>>> x = randn(10000)
>>> hist(x, 100)
```

The pylab mode offers interaction similar to Matlab.
http://matplotlib.sourceforge.net/ Check also
http://gnuplot-py.sourceforge.net/

pyplot

https://www.tutorialspoint.com/matplotlib/matplotlib_pylab_module.htm

```
import matplotlib.pyplot as plt
plt.plot([1,2,3])
plt.ylabel('some numbers')
plt.show()
```

```
import matplotlib.pyplot as plt
plt.plot([1,2,3,4], [1,4,9,16], 'ro')
plt.axis([0, 6, 0, 20])
```

RPy

http://rpy.sourceforge.net/

```
>>> from rpy import *
>>>
>>> degrees = 4
>>> grid = r.seq(0, 10, length=100)
>>> values = [r.dchisq(x, degrees) for x in grid]
>>> r.par(ann=0)
>>> r.plot(grid, values, type='lines')
```

working with numpy arrays

```
import numpy as np
import matplotlib.pyplot as plt

# evenly sampled time at 200ms intervals
t = np.arange(0., 5., 0.2)

# red dashes, blue squares and green triangles
plt.plot(t, t, 'r--', t, t**2, 'bs', t, t**3, 'g^')
```

Even before talking on CGI

```
import urllib

fwcURL = "http://cbbl.imim.es"

try:
    print ("Going to Web for data")
    fwcall = urllib.urlopen(fwcURL).read()
    print ("Successful")
    print ("Will now print all of the data to screen")
    print ("fwcall = ", fwcall)
except:
    print ("Could not obtain data from Web")
```

Even before talking on CGI

```
>>> import urllib2
>>> for line in urllib2.urlopen('http://tycho.usno.navy.mil
...     if 'EST' in line or 'EDT' in line:  # look for East
...         print (line)

<BR>Nov. 25, 09:43:32 PM EST

>>> import smtplib
>>> server = smtplib.SMTP('localhost')
>>> server.sendmail('soothsayer@example.org', 'jcaesar@exam
... """To: jcaesar@example.org
... From: soothsayer@example.org
...
... Beware the Ides of March.
... """)
>>> server.quit()
```

```
#!/usr/bin/env python

import cgi
print ("Content-Type: text/html\n")

print ("""
<HTML>
<HEAD>
<TITLE>Hello World</TITLE>
</HEAD>
<BODY>
<H1>Greetings</H1>
</BODY>
</HTML>
""")
```

Interface design

1. Encapsulation
2. Generalization
3. Interface design
4. Refactoring

Extending/embedding Python

Python provides bindings to other languages that allow for powerful large project building. Check http://docs.python.org/extending/index.html for general information.

Glossary I

problem solving	The process of formulating a problem, finding a solution, and expressing the solution.
high-level language	A programming language like Python that is designed to be easy for humans to read and write.
low-level language	A programming language that is designed to be easy for a computer to execute; also called "machine language" or "assembly language"
portability	A property of a program that can run on more than one kind of computer.
interpret	To execute a program in a high-level language by translating it one line at a time.
compile	To translate a program written in a high-level language into a low-level language all at once, in preparation for later execution.
source code	A program in a high-level language before being compiled.
ob ject code	The output of the compiler after it translates the program.
executable	Another name for ob ject code that is ready to be executed.
prompt	Characters displayed by the interpreter to indicate that it is ready to take input from the user.
script	A program stored in a file (usually one that will be interpreted).
program	A set of instructions that specifies a computation.
algorithm	A general process for solving a category of problems.
bug	An error in a program.
debugging	The process of finding and removing any of the three kinds of programming errors.
syntax	The structure of a program.
syntax error	An error in a program that makes it impossible to parse (and therefore impossible to interpret).
exception	An error that is detected while the program is running.
semantics	The meaning of a program.
semantic error	An error in a program that makes it do something other than what the programmer intended.

Glossary II

natural language	Any one of the languages that people speak that evolved naturally.
formal language	Any one of the languages that people have designed for specific purposes, such as representing mathematical ideas or computer programs; all programming languages are formal languages.
token	One of the basic elements of the syntactic structure of a program, analogous to a word in a natural language.
parse	To examine a program and analyze the syntactic structure.
print statement	An instruction that causes the Python interpreter to display a value on the screen.
instance	A member of a set.
loop	A part of a program that can execute repeatedly.
encapsulation	The process of transforming a sequence of statements into a function definition.
generalization	The process of replacing something unnecessarily specific (like a number) with something appropriately general (like a variable or parameter).
interface	A description of how to use a function, including the name and descriptions of the arguments and return value.
development plan	A process for writing programs.
docstring	A string that appears in a function definition to document the function's interface.

This document's history

1. 2007 : Original version by Michael A. Johnston
2. 2008 : modifications and examples added by JVF
3. 2010-: LaTeX2e version and extensions by JVF

Sources

- Style guide for Python code
  http://www.python.org/dev/peps/pep-0008/
- Library: http://docs.python.org/library/
- http://www.thinkpython.com
- http://diveintopython.org/toc/index.html
- http://docs.python.org/tutorial/introduction.html
- http://openbookproject.net/thinkcs/python/english2e/
- http://www.awaretek.com/tutorials.html
- http://code.google.com/edu/languages/google-python-class/
- http://www.sthurlow.com/python/