Introduction to Python

Prof. James H. Thomas

Use python interpreter for quick syntax tests.

Write your program with a syntax-highlighting text editor.

Save your program in a known location and using ".py" extension.

Use the command window (or terminal session) to run your program (make sure you are in the same directory as your program).

Getting started on the Mac

- Start a terminal session
- Type “python”
- This should start the Python interpreter (often called “IDLE”)
- Use the Python interpreter to test simple things.

```
> python
Python 2.6.4 (something something)
details something something
Type "help", "copyright", "credits" or "license"
for more information.
>>> print “Hello, world!”
Hello, world!
```

Run your program

- In your terminal, Ctrl-D out of the python interpreter (or start a new terminal).
- Type "pwd" to find your present working directory.
- Open TextWrangler.
- Create a file with your program text.
- Be sure that you end the line with a carriage return.
- Save the file as "prog.py" in your present working directory.
- In your terminal, type "python prog.py"

```
> python hello.py
hello, world!
```

Common beginner's mistakes

If your terminal prompt has three '>' characters you are in the Python interpreter:

```
>>> print 7
7
>>>
```

To run a program, be sure you have a normal terminal prompt (will vary by system), will usually end with a '$' or a single '>' character:

```
> python myprog.py arg1 arg2
(program output)
```

When you write your program (in a text editor), be sure to save it before trying out the new version! Python reads the <u>saved</u> file to run your program.

Summary of Command Line Basics

Run a program by typing at a terminal session command line prompt (which may be `>` or `$` or something else depending on your computer; it also may or may not have some text before the prompt).

If you type '`python`' at the prompt you will enter the Python IDLE interpreter where you can try things out (ctrl-D to exit).

If you type '`python myprog.py`' at the prompt, it will run the program '`myprog.py`' if it is present in the present working directory.

'`python myprog.py arg1 arg2`' (etc) will provide command line arguments to the program. Arguments are separated by spaces.

Each argument is a string object and they are accessed using `sys.argv[0]`, `sys.argv[1]`, etc., where the program file name is the zeroth argument.

Write your program with a text editor and be sure to save it in the present working directory before running it.

Objects and types

- An object refers to any entity in a python program.
- Every object has an associated type, which determines the properties of the object.
- Python defines six types of built-in objects:

Number	10 or 2.71828
String	“hello”
List	[1, 17, 44] or [“pickle”, “apple”, “scallop”]
Tuple	(4, 5) or (“homework”, “exam”)
Dictionary	{“food” : “something you eat”, “lobster” : “an edible arthropod”}
File	more later…

- It is also possible to define your own types, comprised of combinations of the six base types.

Literals and variables

- A variable is simply a name for an object.
- For example, we can assign the name “pi” to the Number object 3.14159, as follows:

```
>>> pi = 3.14159
>>> print pi
3.14159
```

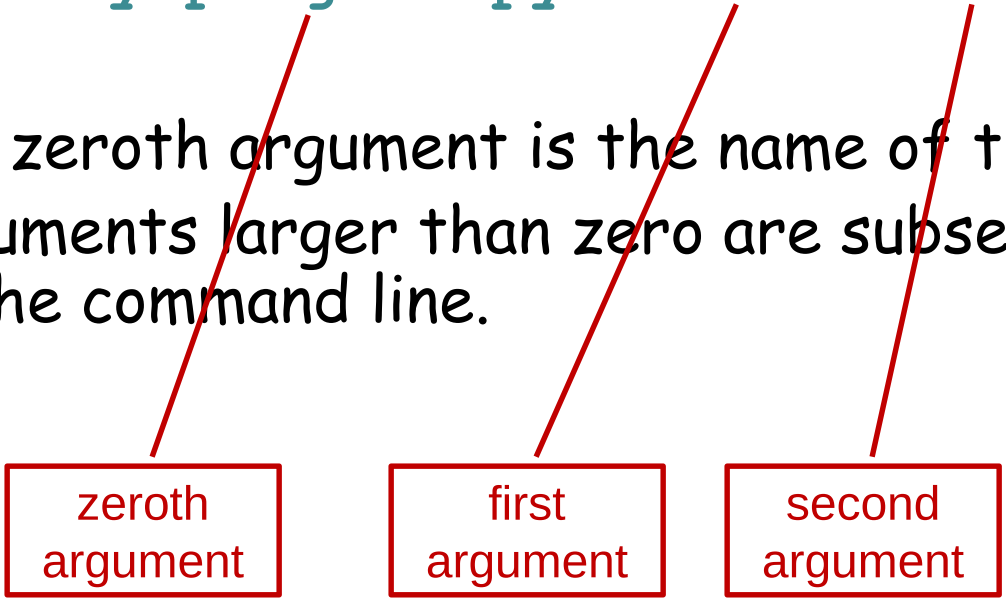
- When we write out the object directly, it is a literal, as opposed to when we refer to it by its variable name.

The command line

- The command line is the text you enter after the word "python" when you run a program.

```
python my-program.py GATTCTAC 5
```

- The zeroth argument is the name of the program file.
- Arguments larger than zero are subsequent elements of the command line.



Reading command line arguments

Access in your program like this:

```
import sys
print sys.argv[0]
print sys.argv[1]

> python my-program.py 17
my-program.py
17
```

zeroth argument

first argument

There can be any number of arguments, accessed by sequential numbers (sys.argv[2] etc).

Assigning variables

In order to retain program access to a value, you have to assign it to a variable name.

```
import sys
sys.argv[0]
```

this says "give me access to all the stuff in the sys module"

this doesn't do anything – it says "get the string that is stored at index 0 in the list sys.argv and do nothing with it"

```
import sys
print sys.argv[0]
```

this says "get the string that is stored at index 0 in the list sys.argv and print it" (but it doesn't do anything else)

```
import sys
s = sys.argv[0]
```

this says "get the string that is stored at index 0 in the list sys.argv and assign it to the variable s"

Numbers

- Python defines various types of numbers:
  - Integer (1234)
  - Floating point number (12.34)
  - Octal and hexadecimal number (0177, 0x9gff)
  - Complex number (3.0+4.1j)
- You will likely only need the first two.

Conversions

```
>>> 6/2
3
>>> 3/4
0
>>> 3.0/4.0
0.75
>>> 3/4.0
0.75
>>> 3*4
12
>>> 3*4.0
12.0
```

truncated rather than rounded

- The result of a mathematical operation on two numbers of the <u>same</u> type is a number of that type.
- The result of an operation on two numbers of <u>different</u> types is a number of the more complex type.

integer → float

Formatting numbers

- The `%` operator formats a number.
- The syntax is `<format> % <number>`

```
>>> "%f" % 3
'3.000000'
>>> "%.2f" % 3
'3.00'
>>> "%5.2f" % 3
' 3.00'
```

Formatting codes

- %d = integer (d as in digit)
- %f = float value - decimal (floating point) number
- %e = scientific notation
- %g = easily readable notation (i.e., use decimal notation unless there are too many zeroes, then switch to scientific notation)

More complex formats

%[flags][width][.precision][code]

Left justify ("-")
Include numeric sign ("+")
Fill in with zeroes ("0")

Total width of output

Number of digits after decimal

d, f, e, g

Examples

```
>>> x = 7718
>>> "%d" % x
'7718'
>>> "%-6d" % x
'7718  '
>>> "%06d" % x
'007718'
>>> x = 1.23456789
>>> "%d" % x
'1'
>>> "%f" % x
'1.234568'
>>> "%e" % x
'1.234568e+00'
>>> "%g" % x
'1.23457'
>>> "%g" % (x * 10000000)
'1.23457e+07'
```

Read as "use the preceding code to format the following number"

Don't worry if this all looks like Greek – you can figure out how to do these when you need them in your programs.

Ιτ συρε λοοκσ λικε Γρεεκ το με.

string basics

Basic string operations:

```
S = "AATTGG"               # assignment - or use single quotes ' '
S1 + S2                    # concatenate two strings
S*3                        # repeat string S 3 times
S[i]                       # get character at position 'i'
S[x:y]                     # get a substring from x to y (not including y)
len(S)                     # get length of string
int(S)                     # turn a string into an integer
float(S)                   # turn a string into a floating point decimal number
len(S[x:y])                # the length of s[x:y] is always y - x
```

Methods:

```
S.upper()                  # convert S to all upper case, return the new string
S.lower()                  # convert S to all lower case, return the new string
S.count(substring)         # return number of times substring appears in S
S.replace(old,new)         # replace all appearances of old with new, return the new string
S.find(substring)          # return index of first appearance of substring in S
S.find(substring, index)          # same as previous but starts search at index in S
S.startswith(substring)           # return True or False
S.endswith(substring)             # return True of False
```

Printing:

```
print var1,var2,var3              # print multiple variables with space between each
print "text",var1,"text"          # print a combination of explicit text and variables
```

list basics

Basic list operations:

```
L = ['dna','rna','protein']     # list assignment
L2 = [1,2,'dogma',L]            # list can hold different object types
L2[2] = 'central'               # change an element (mutable)
L2[0:2] = 'ACGT'                # replace a slice
del L[0:1] = 'nucs'             # delete a slice
L2 + L                          # concatenate
L2*3                            # repeat list
L[x:y]                          # get a slice of a list
len(L)                          # length of list
''.join(L)                # convert a list to a string (a string function that acts on lists)
S.split(x)                      # convert string to list- x delimited
list(S)                         # convert string to list - explode
list(T)                         # converts a tuple to list
```

Methods:

```
L.append(x)                     # add to the end
L.extend(x)                     # append each element from x to list
L.count(x)                      # count the occurrences of x
L.index(x)                      # get element position of first occurrence of x
L.insert(i,x)                   # insert element x at position i
L.remove(x)                     # delete first occurrence of x
L.pop(i)                        # extract (and delete) element at position i
L.reverse()                     # reverse list in place
L.sort()                        # sort list in place
```

`dict` basics

```
D = {'dna':'T','rna':'U'}   # dictionary literal assignment
D = {}                      # make an empty dictionary
D.keys()                    # get the keys as a list
D.values()                  # get the values as a list
D['dna']                    # get a value based on key
D['dna'] = 'T'              # set a key:value pair
del D['dna']                # delete a key:value pair
D.pop('dna')                # remove key:value (and return value)
'dna' in D                  # True if key 'dna' is found in D, else False
```

The keys must be immutable objects (e.g. string, int, tuple).

The values can be anything (including a list or another dictionary).

The order of elements in the list returned by **`D.keys()`** or **`D.values()`** is arbitrary (effectively random).

File reading and writing

The `open()` command returns a file object:

```
<file_object> = open(<filename>, <access type>)
```

Access types:	
	`'r'` = read
	`'w'` = write
	`'a'` = append

`myFile = open("data.txt", "r")` – open for reading
`myFile = open("new_data.txt", "w")` – open for writing

`myString = myFile.read()` – read the entire text as a string

`myStringList = myFile.readlines()` – read all the lines as a list of strings

`myString = myFile.readline()` – read the next line as a string

`myFile.write(“foo”)` – write a string (does not append a newline)

`myFile.close()` – always close a file after done

if - elif - else

```
if <test1>:
    <block1>
elif <test2>:
    <block2>
elif <test3>:
    <block3>
else:
    <block4>
```

- Only one of the blocks is ever executed.
- A block is all code with the same indentation.

Comparison operators

- Boolean: `and, or, not`
- Numeric: `< , > , ==, !=, >=, <=`
- String: `in, not in`

`<`	`is less than`
`>`	`is greater than`
`==`	`is equal to`
`!=`	`is NOT equal to`
`<=`	`is less than or equal to`
`>=`	`is greater than or equal to`

`for` loops

`for <target> in <object>`	object can be a list, a string, a tuple
`for letter in "Constinople"`	
`for myString in myList`	(where myList is a list of strings)
`continue`	skip the rest of the loop and start at the top again
`break`	quit the loop immediately

As usual, all the commands with the same indentation are run as a code block.

`for integer in range(12)`	range simply returns a list of integers

```
range([start,] stop [,step])
```

Loops can be nested or have other complex code blocks inside them.

Examples of `for` loops

```
for base in sequence:
    <do something with each base>

for sequence in database:
    <do something with each sequence>

for base in ["a","c","g","t"]:
    <do something with each base>

for index in range(5,200):
    <do something with each index.
```

`while` loops

Similar to a `for` loop

```
while (conditional test):
         <statement1>
         <statement2>
         . . .
         <last statement>
```

While something is `True` keep running the loop, exit as soon as the test is `False.`

Any expression that evaluates True/False can be used for the conditional test.

Examples of **while** loops

```
while (error > 0.05):
    <do something that will reduce error>


while (score > 0):
    <traceback through a DP matrix, each
    time setting the current score>
```

Sorting

The `list` method `sort` implements an efficient version of the merge sort algorithm (actually it uses both merge sort and insertion sort in a hybrid usually call timsort).

The sort is "stable" meaning that equal elements come out of the sort in the same order they started.

The stable property means you can apply multiple sorts sequentially and everything will behave the way you want.

You can supply your own comparison method as an argument to sort.

```
myList = [0,2,7,-5,4,-2,3]
myList.sort()
# myList will now be [-5,-2,0,2,3,4,7]

# switching the -1 and 1 below will cause this to
# act the same as the default sort
def comp(A,B):
   if A > B:
      return -1
   if A < B:
      return 1
   return 0

myList.sort(comp)
# myList will now be [7,4,3,2,0,-2,-5]
```

The comparator function provided to sort can be arbitrarily complex and it can be designed to compare any two entities.

Here is a comparator function that will sort by length:

```
def lencomp(A,B):
   if len(A) > len(B):
      return 1
   if len(A) < len(B):
      return -1
   return 0

myList = ["a","tttt","cc","ggg","ttaa"]
myList.sort(lencomp)
# myList is now ["a","cc","ggg", "tttt","ttaa"]
```

Note that `tttt` and `ttaa` remain in their original order even though they are the same length (the stable property).

Sequential sorts taking advantage of the stable property:

```
def lencomp(A,B):
   if len(A) > len(B):
      return 1
   if len(A) < len(B):
      return -1
   return 0

myList = ["a","tttt","cc","ggg","aatt"]
myList.sort()
# myList is now ["a","aatt","cc","ggg","tttt"]

myList.sort(lencomp)
# myList is now ["a","cc","ggg","aatt","tttt"]
```

Code efficiently

Time efficiency

Rough order of speed for common operations:

> reading/writing files - very slow
> going through a list serially for matching elements - slow
> accessing a list (or string) element by index - fast
> accessing a dictionary value by key - fast

File reading - sometimes you only need to look through a file until you find something. In this case, read lines until you have what you need then close the file.

Dictionaries can be used in various clever ways to save time.

Do simple profiling to see what part of your code is slowest (for example, invoke `time.time()` twice, once before and once after a code block).

Future - beginning Python is kept simple by hiding a lot of complex things from you - dig in deeper to understand what takes more time (and memory).

Beware of string slicing - it is beguilingly simple and very time (and often memory) wasteful. For example, these two code snippets do the same thing, but the second one is much faster, especially if `longStr` is very long and has lots of matches:

```
query = "foo"
pos = longStr.find(query)
while pos != -1:
    print pos
    longStr = longStr[pos+1:]  # creates a new string
    pos = longStr.find(query)
```

BAD

```
query = "foo"
pos = longStr.find(query)
while pos != -1:
    print pos
    pos = longStr.find(query, pos+1) # starts search inside the same string
```

GOOD

By the way, `longStr.find(query)` is identical to `longStr.find(query, 0)`. When you leave out the search index, python provides 0 as the default.

Think about what is happening under the hood

For example, here is what is happening under the hood in the bad code snippet from the previous slide:

1) find and report the position of the match
2) allocate a memory block to hold the new (sliced) string
3) read the string slice from the old memory block into the new memory block
4) free the memory associated with the old memory block
5) repeat

Steps 2 through 4 are time consuming and totally unneeded.

Beware of string slicing clocked example

string of length 700,000:

```
count = 0
ix = 0
while ix < len(longStr):
    if longStr[ix] == 'T':
        count += 1
    ix += 1
```

runtime 0.25 sec

```
count = 0
while len(longStr) > 0:
    if longStr[0] == 'T':
        count += 1
    longStr = longStr[1:]
```

runtime 73.7 sec

Use `xrange()` rather than `range()`

`xrange()` is functionally nearly identical to `range()`, but rather than actually creating a list that you loop (iterate) over, it provides a single incremented value.

We teach `range()` because it elegantly shows how you can iterate over a list, but usually you don't need the list.

$\texttt{for ix in }\underbrace{\texttt{range(0, len(longStr))}}_{\text{creates a list, with elements 0, 1, 2, 3, ...}}\texttt{:}$

$\texttt{for ix in }\underbrace{\texttt{xrange(0, len(longStr))}}_{\text{creates a single integer value that is increased by 1 each time through the loop}}\texttt{:}$

This hardly matters when `longStr` isn't all that long, but if it is, e.g. genome sized, it matters a <u>lot</u>, both in memory and in time.

Memory efficiency

File reading - often you don't need to save the entire contents of a file into memory. Consider whether you can discard some of the information while you are reading the file.

If your program generates many copies of the same long `string`, consider making a `dict` or `list` entry with the `string` as the value (you only need to store it once). You will access the `string` via the `list` index or `dict` key.

If you are working with a long string and you want to access many segments of it, do NOT save each segment as a string - use string indexing to get each segment as you need it.

Future - instead of using Python lists, consider using classes in the Python `array` module to store long sequences, etc.

Timing code blocks

You may want to know what part of your program is working slowly in order to focus on improving it. This is very easy.

The `time()` method in the `time` module returns a float number that gives the number of seconds elapsed since "the epoch" (an arbitrary reference moment in the past) at the exact moment the `time()` method is executed (it accesses your computer clock to do this). To use it, just flank the code you want to time with two calls to `time.time()` and print to the screen (or save to a log file, or whatever). If you want to time the entire program, put them at the start and end (duh!). Typically `time.time()` is accurate to about 1 msec.

```
import time

<various code>

startTime = time.time()
<code you want to clock here>
print time.time() - startTime, 'seconds elapsed'

<various other code>
```

Classes

Writing and using your own classes

Writing and using your own classes is very easy.

Use when you want:

- a data structure not conveniently fitting existing Python types
- a data structure with your own implemented methods

```
class <class_name>:
    def __init__(self, [optional constructor args]):
        <constructor code>
    def myClassMethod(self, [optional args]):
        <method code>
```

Notes - python defines several special method declarations related to classes. They all start and end and double underscores. `__init__` is a "constructor" method used to initialize an instance of a class. The `__str__` method on the next slide is another example. The "self" reference appears first in the arguments for every class method - it gives Python a reference to the specific object being used.

Example of a fully implemented (though simple) class to hold protein or nucleotide sequences

```
class Sequence:
    def __init__(self, name, seq):
        self.name = name
        self.sequence = seq.replace('\n','')
    def toFastaString(self):
        lines = []
        for i in xrange(0, len(self.sequence), 100):
            lines.append(self.sequence[i:i+lineLen])
        return '>' + self.name + '\n' + '\n'.join(lines)
    def __str__(self):
        return self.toFastaString()
    def __len__(self):
        return len(self.sequence)
```

stores the name and seq data into class data variables.

Note: object-associated data (self.name and self.sequence in this case) are called many things, including "attributes", "fields", "data fields", "properties", or "data members". They all mean essentially the same thing. Python authors usually use "attributes".

The constructor removes newlines in the sequence as a precaution, since they are often read in a form that has embedded newlines. The `toFastaString` method inserts new lines in the sequence to make easily readable output. The `__str__` method makes built-in Python `print` and `str` functions do something sensible with a `Sequence` object. The `__len__` method makes Python `len` work sensibly. It is often a good idea to implement `__str__` and `__len__` for a new class if you plan to use it much.

Using the Sequence class

(assumes you have already defined the sequence class as shown on the previous page)

```
>>> sname = 'mySeq'
>>> seqres = 'AGGCTATACTAGGCTA'
>>> seq1 = Sequence(sname, seqres)
>>> seq1.toFastaString()
>mySeq
AGGCTATACTAGGCTA
>>> len(seq1)
16
>>> print seq.name
mySeq
>>> print seq.sequence
AGGCTATACTAGGCTA
```

creates a Sequence object

the '>' on this line is the start of the fasta name, not a prompt

because we defined the `__len__` method, the Python built in len function does the sensible thing

you can reference any data associated with your object (object attributes) using the familiar dot notation

Notes on classes

The only real difference from creating a built-in Python object is that you use the class name to construct it, rather than the special symbols that Python interprets as a particular class type. e.g.:

```
myList = ['Hello', 'World']
```

what Python is really doing is creating a List object using an `__init__` method; the [ ] is just a shorthand

The "self" reference that appears first in the definition of every class method is NOT used when you call the method. It is there so that when the Python method is called it has a reference to the specific object to which the method is to be applied.

There is a whole series of special class methods that all start and end with double underscore (e.g. `__len__`). These can be very handy. Another example is the `__add__` method, which defines what the operator `+` does when you try to “add” two of your objects.

Keep in mind namespace and variable passing when constructing an object or using an object method. E.g. in our definition of `Sequence`, the pointer associated with `name` in the constructor is copied to `self.name`. Since they both point to an immutable string, changing `name` (outside the object) or `self.name` (inside the object) has no effect on the other. However, if name pointed to a list or any other mutable object you COULD change the contents in either place.

Extending Classes

You can "extend" any Python class, including ones you have written or ones that are provided with Python.

Extension formalism - much like biological classification

```
class Eukaryote

          class Animal (extends Eukaryote)
                    define class method movementRate()

                    class Insecta (extends Animal)
                              define class method numberOfWings()

                              class Drosophila (extends Insecta)
                                        define class method preferredFruit()
```

What methods are available for an object of type `Drosophila`?

`Drosophila` is an `Insecta` so it has all the `Insecta` data structures and methods.
`Drosophila` is also an `Animal` so it has all the `Animal` data structures and methods (and Eukaryote, though we didn't define any).

Writing a new Class by extension

Writing a class:

```
class Date:
    def __init__(self, day, month, year):
        <assign arguments to class variables>
    def etc. # define a whole bunch of Date methods
```

Extending an existing class:

class to extend

```
class HotDate(Date):
   def __init__(self, day, month, year, toothbrush):
       super(day, month, year)
       self.bringToothbrush = toothbrush
```

super - call the constructor for Date

All of the data types and methods written for `Date` are also available for a `HotDate` object (plus others specific for `HotDate`).

Class hierarchy

super is Eukaryote

```
class Eukaryote

     class Animal (extends Eukaryote)
          add class method movementRate()
```

super is Animal

```
          class Insecta (extends Animal)
               add class method numberOfWings()

               class Drosophila (extends Insecta)
                    add class method preferredFruit()
```

super is Insecta

The next class up the hierarchy is the <u>superclass</u> (there can only be one).
Each class down one level in the hierarchy (there can be more than one)
is a <u>subclass</u>.

Exception handling

Using Python exceptions

When an error occurs in your program, you don't necessarily have to live with those nasty default error messages. If you anticipate a possible error, you can do something more sensible (usually user feedback).

You do this by embedding the code that might cause an error in a `try` clause, following by an `except` clause.

The `try` key word tells Python that it should try to execute a code block, and the `except` key word tells Python what to do if it fails.

If `except` is followed by a specific type of exception, only that type of exception will be caught.

If `except` is followed by nothing, ANY exception will be caught.

You can put `try-except` clauses anywhere.

Python provides several kinds of exceptions (each of which is of course a class!). Some common exception classes:

```
ZeroDivisionError # when you try to divide by zero
NameError # when a variable name can't be found
MemoryError # when program runs out of memory
ValueError # when int() or float() can't parse a value
IndexError # when a list or string index is out of range
KeyError # when a dictionary key isn't found
ImportError # when a module import fails
SyntaxError # when the code syntax is uninterpretable
```

(note - each of these is actually an extension of the base Exception class - any code shared by all of them was written once for the Exception class)

Example - enforcing format in the Date class

```
class Date:
   def __init__(self, day, month, year):
      try:
         self.day = int(day)
      except ValueError:
         print 'Date constructor: day must be an int value'
      try:
         self.month = int(month)
      except ValueError:
         print 'Date constructor: month must be an int value'
      try:
         self.year = int(year)
      except ValueError:
         print 'Date constructor: year must be an int value'
```

only catches this type of exception

FYI, if there are <u>other types</u> of exceptions (not a `ValueError`), they will be reported by the default Python exception handler, with familiar output, e.g.:

```
Traceback (most recent call last):
  File <pathname>, line X, in <module>
    <code line>
<default exception report>
```

Create your own Exception class

(not normally necessary, but FYI)

```
import exceptions

class DayFormatException(exceptions.Exception):
    def __str__(self):
        print 'Day must be parseable as an int value'
```

DayFormat extends the Python defined Exception class (which is the superclass of other Exception classes as well)

Remember that the __str__ function is what print calls when you try to print an object.

Using your own Exceptions

```
class Date:
    def __init__(self, day, month, year):
        try:
            self.day = int(day)
        except:
            raise DayFormatException
```

`raise` is a new reserved key word - it raises an exception.
The DayFormatException will get returned to whenever the constructor was called - there it can be "caught" if it in turn was embedded in a try-catch clause like this:

```
try:
    myDate = Date("Ides", "March", "IXIV")
except:
    <do something>
```

catch the exception raised by the Date constructor

Exceptions - when to use

- Any software that will be given to someone else, especially if they don't know Python.

- Private software that is complex enough to warrant.

- Just as with code comments, exceptions are a useful way of reminding yourself of what the program expects.

- They have NO computational cost (if no exception is thrown, nothing at all is computed).

Better command line arguments

(how I learned to stop worrying and love the command line)

avoid writing programs that are invoked like this:

```
python myprog.py 12 7 1.2 True file1 file2
```

(it might seem obvious while you are writing the program, but come back to it a month later and you will have to read the code to figure out what the hell all those arguments are for)

Improved command line arguments

The method we covered in class for command line arguments is very clunky when you are writing programs that have many arguments.

No doubt you have used programs (like grep etc) that have a "-" or "--" argument system (e.g. `grep -v -x mystring myfile`).

Python provides a rather handy system to do a similar thing, although at first it will seem rather weird. (There is even a new system for doing this that I have not bothered with and I will show the old system since I know it and it still works just fine.)

This Python system also gives you a simple form of text help for free (typically invoked using `myProgram -h`, which prints some instructions to the screen).

The system is embodied in the `OptionParser` class in the `optparse` module.

It is easiest to see an example and use it as a template:

```
from optparse import OptionParser

oparser = OptionParser() # create an instance to populate with options
oparser.add_option("--maxdist", type="int", dest="maxDist", default="3000")
oparser.add_option("--score_min", type="float", dest="scoreMin", default="40")
oparser.add_option("--align", action="store_true", dest="align", default=False)
oparser.add_option("--in", type="string", dest="fileName", default=None)
# when all the options have been added, put them into a tuple for easy use
(opt, args) = oparser.parse_args(sys.argv)
# here are some typical uses of the options - notice their name was set by dest=
if opt.align:
    <do some alignment thing>

reportAlignments(opt.scoreMin)   # report some alignments with appropriate scores

etc.
```

Each **add_option** call sets the command line option, the type expected, the destination variable name to use inside your program, and a default value. For booleans the "type" is replaced by an action to take.

When you use the options at the command line, all except the boolean type are paired with a value:

```
>myprog.py --align --score_min 20 --in mySeqFile.fasta
```

The boolean options (like `--align`) are "flags" - typically they have a default of `False` and are set to `True` if you include them at the command line. The other options always are followed by their value.

The options can be given at the command line in any order. These are identical:

```
>myprog.py --align --score_min 20 --in mySeqFile.fasta

>myprog.py --score_min 20 --align --in mySeqFile.fasta
```

If the user gives a `-h` or `--help` option at the command line some help text is provided without your writing any new code. Try it out - it is kind of cool.

If you want to add specific help for the user, just add to the end of your `add_option()` code, something like this:

```
oparser.add_option("--in", type="string",
dest="fileName", default=None, help="input fasta file")
```

You can also use single hyphen options but only if they are one letter long. (don't ask why, it has to do with the tangled history of Unix)

stdin, stdout, stderr, and log files

Making output to stdout and stderr and log files

Many programs make their output to what are called "standard out" and "standard error" (the print command goes to standard out, the default python error messages go to standard error). By default, both are written to the screen, but you can redirect each of them (see below).

These two streams are available directly to you via the `sys` module: `sys.stdout` and `sys.stderr`.

You can write to either one with file output-like statements, e.g.

```
sys.stdout.write("blah blah\n")
sys.stderr.write("read 6 sequences, analysis complete\n")
```

When you use a program with these outputs, you can direct each stream into files as follows (`stdout` to fileA and `stderr` to fileB):

```
python myprog.py > fileA.txt 2> fileB.txt
```

Users of command line programs often expect to find these two sorts of outputs, which roughly correspond to the main intended output of the program (`stdout`) and error or progress messages (`stderr`).

If you have more complex output, or want to have an error output separate from a progress report or other sorts of metadata, create a log file and write the metadata to that file. e.g.

```
from optparse import OptionParser
oparser = OptionParser()
oparser.add_option("--log", type="string", dest="logFile", default="out.log")
(opt, args) = oparser.parse_args(sys.argv)

# ancillary information, such as progress report, parameters used for the
# analysis, files used in the analysis, time and date stamp etc. will be
# written to the log file, which has a default name but can be set by the user.
```

These three code patterns (using `stdout`, `stderr`, and a log file) are in common use and are very handy. I use them frequently.

A log file is especially useful to record program parameters associated with a specific program run, but that you don't want cluttering up the main output.

For example, suppose you write a sequence aligner that makes its main output in the form of fasta format alignment. You might want to have a log file record things like the input file name, the score matrix, the gap penalties, and the date and time run, and `stderr` to show program progress.

The log file is like a lab notebook.

Standard input

Standard in is analogous to standard out, except of course it is input to a program. `sys.stdin` is a file-like stream that you can read inside your program as if it were a file:

```
for line in sys.stdin:
    # do something with each line
```

Under some circumstances it is ideal to allow a user to provide input EITHER from a file OR from stdin. e.g.

```
if fileName != None:
    inf = open(fileName)
else:
    inf = sys.stdin
# after this the code is identical for either data source
```

Why bother with stdin?

Sometimes, writing to and reading from files is a slow step in program execution.

Though you can treat `sys.stdin` as if it were a file, in fact it is sitting in RAM (actually when you read a file you access the data from RAM too, it is just read from the disk first and put in RAM).

Suppose I have a tree-building program (gotten from someone else) that can take a sequence alignment from `stdin` or a file. I wrote a program (or got a program) that can write a sequence alignment either to `stdout` or to a file. These are equivalent:

```
python align_prog.py infile > outfile
tree_prog outfile > outtree
```

or

```
python align_prog.py infile | tree_prog > outtree
```

The difference is that the first version has to save the alignment to a file, then read it again. The second version never writes a sequence alignment file. That `|` symbol is read "pipe" and connects the two programs via stdout and stdin (if you use unix much it is used the same way).

By the way, the rather confusing-looking command below says to read the contents of `infile` and feed them to `tree_prog` on `stdin`.

```
tree_prog < infile > outtree
```

I find this a bit confusing and I avoid it, but it is in common use.