Introduction to Programming using the Python Programming Language

Scott Hazelhurst

2012
Objectives

- learn algorithmic techniques and data structures; apply them in Python; write larger pieces of code
- understand how to use Python libraries
  - regular expressions;
  - operating system;
  - ...
- Python as a *scripting* language
Curriculum

- brief introduction to algorithm design principles
- data structures i: basic data structures
- algorithmic constructs
- data structures ii: collections
- modularity and functions
Introduction to Programming using the Python Programming Language

Objectives

Documentation

▶ http://www.python.org
▶ http://www.python.org/doc/
▶ also look for the python-doc package. usually installed, if not get it installed!!!
  ▶ usually placed in something like /usr/share/doc/python2.7.

Bookmark it!
Why Python?

- Modern programming language
- Clean syntax
- Good learning curve
- Reasonable semantics
- Powerful features, Lots of libraries
- Cross-platform

Of course weaknesses too:
- Python not the right tool for all problems.
PLOTS AND STATISTICAL METHODS

My study used the multiple boxplot display shown in Figure 1 for its main evaluation tool. Each of the lines represents one subset of data, with its name appearing to the left. Each small circle stands for an individual data value. The rest of the plot provides visual aids for the comparison of two or more such data subsets. The shaded box indicates the range of the middle half of the data, that is, from the first (25 percent) quartile to the third (75 percent) quartile. The “whiskers” to the left and right of the box indicate the data’s bottom and top 10 percent, respectively. The fat dot within the box is the median (50 percent) quartile. The “M” and the dashed line around it indicate the arithmetic mean and the mean’s plus-and-minus-one standard error.

For quantitatively describing the variability within one group of values, I use the bad-to-good ratio: Imagine the data split into an upper and lower half, with the bad-to-good ratio being the median of the upper half divided by the median of the lower half. In the boxplot, the median is the value at the right edge of the box divided by the value at the left edge. In contrast to a variability measure such as the standard deviation, the bad-to-good ratio is robust against outliers.

Most significant observations can easily be made directly from the plots. To be sure, however, I also performed statistical tests. Medians are compared using the one-sided Mann-Whitney U-test (also known as the Wilcoxon rank sum test). The result of each test is a p-value, that is, a probability that the observed differences between the samples are only accidental and that either no difference or a difference in the opposite direction between the underlying populations.

Lutz Prechelt
University of Karlsruhe

Often heated, debates about different programming languages remain inconclusive. The author takes a first step toward providing hard data about the relative effectiveness of C, C++, Java, Perl, Python, Rexx, and Tcl. When it comes to the pros and cons of various programming languages, programmers and computer scientists alike usually hold strong opinions. By comparing several languages, I seek to provide some objective information about C, C++, Java, Perl, Python, Rexx, and Tcl.

For the comparison, I used the same program, which implements the same set of requirements for each language. Doing so makes the comparison narrow but homogeneous. Further, for each language, I analyze several separate implementations by different programmers. Such a groupwise comparison offers two advantages. First, it smoothes out the differences among individual programmers, which could threaten the validity of any comparison based on a single implementation per language. Second, it allows us to assess and compare the variability of the program properties the different languages induce.

The comparison investigates several aspects of each language, including program length, programming effort, runtime efficiency, memory consumption, and reliability. I also consider the languages both individually and combined into groups. Scripting languages such as Perl, Python, Rexx, and Tcl tend more toward being interpreted than compiled, at least during the program development phase, and they typically do not require variable declarations. The more conventional programming languages—C, C++, and Java—are compiled rather than interpreted, and they require typed variable declarations. Since Java is often believed to be very inefficient, I also sometimes consider C and C++ as one group and Java as another.
Table 1. Programming-language comparison statistics.

<table>
<thead>
<tr>
<th>Language</th>
<th>Number of Programs</th>
<th>Execution Platform</th>
</tr>
</thead>
<tbody>
<tr>
<td>C++</td>
<td>5</td>
<td>GNU g++ 2.7.2</td>
</tr>
<tr>
<td>Java</td>
<td>24</td>
<td>Sun JDK 1.2.1/1.2.2</td>
</tr>
<tr>
<td>Perl</td>
<td>13</td>
<td>Perl 5.005_02</td>
</tr>
<tr>
<td>Python</td>
<td>13</td>
<td>Python 1.5.2</td>
</tr>
<tr>
<td>Rexx</td>
<td>4</td>
<td>Regina 0.08g</td>
</tr>
<tr>
<td>Tcl</td>
<td>10</td>
<td>Tcl 8.2.2</td>
</tr>
</tbody>
</table>

The programs come from two different programs were produced under more variable conditions, created by volunteers after the study's validity caveats, these quantitative results should be considered at least that is my personal impression.}

The Perl, Python, Rexx, and Tcl languages may attract only fairly competent programmers, hence the script programs may reflect higher average programmer capabilities than the nonscript programs. Furthermore, the nonscript programs may attract especially capable programmers—because the Perl language appears to be more capable than the others. Within the nonscript group, the Java programmers tended to be less experienced than the C and C++ programmers because Java was still an experimental language in 1997 and 1998. In the case, we only need the programs to be comparable on average, not in individual cases, we only need the programs to be comparable on average, not in individual cases.
There is no significant difference between the average Java and script runtimes. The results give an 80 percent confidence that a script will run at least 1.29 times as long—and a Java program at least 1.22 times as long—as a C or C++ program. The bad-to-good ratios are much smaller for scripts (4.1), than for Java (18) or even C and C++ (35).

Initialization phase only, z0 data set. I next focused on the time to read, preprocess, and store the dictionary. Figure 2 shows the corresponding runtimes. The results clearly show C and C++ to be faster in this phase than all other languages tested. The fastest script languages are again Perl and Python.

For the aggregate grouping, we find with an 80 percent confidence level that, compared to a C or C++ program, a Java program will run at least 1.3 times as long, while a script will run at least 5.5 times as long. Compared to a Java program, a script will run at least 3.2 times as long.

Search phase only. Finally, I subtracted the runtime for the loading phase (z0 data set) from the total runtime (z1000 data set) to obtain the runtime for the actual search phase only. Figure 3 shows the corresponding runtimes, which reveal the following:

• Very fast programs occur in all languages except for Rexx and Tcl, while very slow programs occur in all languages.
• Tcl’s median run time is longer than the run times for Python, Perl, and C, but shorter than the run time for Rexx.
• Python’s median runtime is shorter than the run times for Rexx and Tcl and tends to be shorter than the run time for Java ($p = 0.13$).
• Perl’s median run time is shorter than the medians for Rexx, Tcl, and Java.
• C++’s median differs significantly from any other language’s.

The group-aggregated comparison indicates no significant differences among any of the groups. However, the results give an 80 percent confidence that the scripts’ runtime variability is smaller than that of Java by a factor of at least 2.1, and smaller than that of C and C++ by a factor of at least 3.4.

Memory consumption Figure 4, which shows the total process size at the end-of-program execution for the z1000 input file, encourages several observations:

• Clearly, the most memory-efficient programs come from the C and C++ groups, while the least memory-efficient programs come from the Java group.
• Except for Tcl, few of the script languages consume more memory than the worst half of the C and C++ programs.
• Tcl scripts require more memory than other scripts.
• For Python and Perl, the relative variability in memory consumption tends to be much smaller than for C, and in particular, C++.
• A select few of the scripts consume large amounts of memory.
• On the average, for the group-aggregated view and with a confidence of 80 percent, the Java programs consume at least 32 Mbytes more memory (297 percent) than the C and C++ programs, and at least 20 Mbytes more memory (98 percent) than the script programs. The script programs consume at least 9 Mbytes more memory (85 percent) than the C and C++ programs.
The chart above shows the total working time for realizing the program, measured in hours. The data is presented for different programming languages: TCL, Rexx, Python, Perl, Java, C++, and C. The y-axis lists the programming languages, and the x-axis represents the total time in hours.

- TCL: The total median time is 3.1 hours, with a range from 0.6 to 5.2 hours.
- Rexx: The total median time is 10.0 hours, with a range from 1.1 to 26.7 hours.
- Python: The total median time is 5.0 hours, with a range from 1.2 to 13.3 hours.
- Perl: The total median time is 3.6 hours, with a range from 0.8 to 9.2 hours.
- Java: The total median time is 10.3 hours, with a range from 2.4 to 26.0 hours.
- C++: The total median time is 4.4 hours, with a range from 0.9 to 9.1 hours.
- C: The total median time is 5.2 hours, with a range from 0.9 to 9.3 hours.

The chart indicates a range of Java, all data points, except perhaps the top three TCL and topmost Perl results, are quite plausible. The bad-to-good ratios range from 1.5 for C up to 3.2 for Perl. Three Java work times of 40, 49, and 63 hours exceed the chart's bounds and thus are not shown.
Objectives

- I learned it last night! Everything is so simple!
- Hello World is just print "Hello, world!"

- I dunno... dynamic typing?
- Whitespace?
- Come join us! Programming is fun again! It's a whole new world up here!
- But how are you flying?

- I just typed import antigravity
- That's it?
- ... I also sampled everything in the medicine cabinet for comparison.
- But I think this is the Python.
Top programming languages – TIOBE February 2012

1. Java 17.1%
2. C 16.5%
3. C# 8.6%
4. C++ 9%
5. Objective-C 7.1%
6. PHP 5.6%
7. Visual Basic 4.3%
8. Python 3.1%
9. Perl 2.9%
10. JavaScript 2.5%

Highly used in bioinformatics
Python 2 vs 3

Long term future is Python 3
- But some libraries not available

Python 2
- Last version is 2.7 — now in maintenance

Python 3
- Python 3.2
- **Not** backwards compatible
Program design

Use a design methodology for any real program.

- requirements
- design
- implementation
- testing
- ... and repeat

Planning

plan ... plan ... plan ... plan ... plan ... plan ...
... plan ... plan ... plan ... plan ... plan ... plan ...
program = data structures + algorithms

Outline of process

▶ what data we need represent
▶ how it is manipulated
  ▶ what is the input?
  ▶ what is the output?
Data structure: first, understand the concept.

- understand the problem
**Data structure:** first, understand the concept.

- understand the problem
- decide what data must be represented
  - what is persistent data – exists beyond life of program
  - stored data
  - what is temporary / on-the-fly
Data structure: first, understand the concept.

- understand the problem
- decide what data must be represented
  - what is persistent data – exists beyond life of program
  - stored data
  - what is temporary / on-the-fly
- determine relationship of data
- strive for
  - simplicity, elegance, clarity, accuracy . . .
Data structure: first, understand the concept.

- understand the problem
- decide what data must be represented
  - what is persistent data – exists beyond life of program
  - stored data
  - what is temporary / on-the-fly
- determine relationship of data
- strive for
  - simplicity, elegance, clarity, accuracy . . . . no compromises, no tricks
Implementation
Second, decide how to implement this using the tools available and data structures provided by the language:

- variables (integer, floating point)

Strive for: as above, but efficiency (memory, time) may become critical issues so compromise often necessary
Implementation
Second, decide how to implement this using the tools available and data structures provided by the language:

- variables (integer, floating point)
- sequences
  - strings
  - tuples
  - lists

Strive for: as above, but efficiency (memory, time) may become critical issues so compromise often necessary
Implementation
Second, decide how to implement this using the tools available and data structures provided by the language:

▶ variables (integer, floating point)
▶ sequences
   ▶ strings
   ▶ tuples
   ▶ lists
▶ files
▶ classes
▶ callable types

Strive for: as above, but efficiency (memory, time) may become critical issues so compromise often necessary
Algorithm

Start with high-level design

- make sure you understand how to solve the problem
Algorithm

Start with high-level design

- make sure you understand how to solve the problem
- break the problem up
  - divide-and-conquer
  - understand the interface between the different components
Algorithm

Start with high-level design

- make sure you understand how to solve the problem
- break the problem up
  - divide-and-conquer
  - understand the interface between the different components
- sketch a solution (can you give written instructions for an 8 year old)
Algorithm

Start with high-level design

- make sure you understand how to solve the problem
- break the problem up
  - divide-and-conquer
  - understand the interface between the different components
- sketch a solution (can you give written instructions for an 8 year old)
- formalise solution
Algorithm

Start with high-level design

- make sure you understand how to solve the problem
- break the problem up
  - divide-and-conquer
  - understand the interface between the different components
- sketch a solution (can you give written instructions for an 8 year old)
- formalise solution
- think about alternatives, options
Algorithm

Start with high-level design

- make sure you understand how to solve the problem
- break the problem up
  - divide-and-conquer
  - understand the interface between the different components
- sketch a solution (can you give written instructions for an 8 year old)
- formalise solution
- think about alternatives, options
Refine into final program . . . worry about exact syntax etc

- what structure
  - what modules (new or old)
  - what functions/methods
  - how functions built
- sequence
- assignment, expressions
- iteration (for/while)
- selection (if)
- exceptions
Testing . . . getting it to work

Reality, even for small programs:

- program is read much more than written
- you’ll spend more time debugging it than writing it
Introduction to Programming using the Python Programming Language

Program Design

Testing . . . getting it to work

Reality, even for small programs:

► program is read much more than written
► you’ll spend more time debugging it than writing it

So:

► plan ahead
► comment your code, make sure well structured
► design good test cases
► learn to use tools like debuggers
Range of test cases depends on complexity of code

Test cases:
- trivial cases
- simple cases
- ‘normal’ cases
- difficult/extreme cases
- illegal cases

Test as you go along
- unit testing
- system testing
What is an algorithm

Algorithm is a set of instructions for accomplishing a task:

- a pattern for knitting a jersey
- a recipe for cooking a meal
- instructions for making tea, a kite, . . .
- directions for travelling from one place to another
- instructions for filling a bath
- instructions for performing a mathematical calculation
What is an algorithm

Algorithm is a set of instructions for accomplishing a task:

- a pattern for knitting a jersey
- a recipe for cooking a meal
- instructions for making tea, a kite, . . .
- directions for travelling from one place to another
- instructions for filling a bath
- instructions for performing a mathematical calculation
Key points about instructions:

▶ should be *unambiguous*

▶ written in an appropriate language

▶ require person/thing executing instruction to understand *context* and the language
Program

Program: an algorithm written in a special language capable of being executed by a computer.

- To solve a problem on computer, we specify the problem.
- Give high level solution
- Refine solution into program.

Many different programming languages with different strengths, weaknesses and purposes.
To understand and use a programming language must understand

▶ syntax: the form. The rules of grammar which must be obeyed.  
  rules are very strict.
▶ semantics: the meaning

Correct syntax does not imply correct meaning – we study Python as an example.
▶ The \textit{constructs} and \textit{structures} of Python have a precise semantics.
▶ Ultimately, any solution must be expressed using these constructs.
Data structures I

Data must be stored in memory:

- primary memory: RAM
- secondary memory: disk
Primary memory — “RAM”

- transient, fast: typical sizes 2GB-8GB (120GB huge).
- PLs have a convenient way to use and access.
Primary memory — “RAM”

- transient, fast: typical sizes 2GB-8GB (120GB huge).
- PLs have a convenient way to use and access.

Permanent memory — the file system

- typically hard disk, tapes etc.
- larger, cheaper, but much slower than primary memory (typical sizes above 250GB, easily up to TB size)
- not as convenient to access

Programs can access both types of memory
Variables

The key construct is a variable: a location in (primary) memory where data is stored. In Python:

- variable has a type
- must assign a variable a value before trying to access it!
- declaration is implicit
Different types for:

- numbers (distinguish between integers and floating point)
- strings
- truth values/booleans
- collections of different sorts
- files (for accessing external memory)

We can construct new types from primitive ones
Operations on variables

**Assignment** gives or updates a variable’s value.

```
varname = expression
```

Read from right-to-left:

- evaluate the expression on the right-hand side;
- store its value in the variable on the left-hand side
Examples

\[
i = 10
k = (i+20)*3
i = i+1
k = k*i
\]
value of a variable can be used in an expression;

- **syntax** and **semantics** of an expression depends on type of expression

- can print out a variable (or expression)
  ```python
  print(i)
  ```
Numeric types in Python

- `int` : integer type
  unlimited in size

- `float` : for floating point numbers (e.g. 3.5, 2.66626)
  NB: generally there is a loss of precision when representing floats and you must take care.

- `Complex`: for complex numbers. Use $j$ to represent square root of $-1$.
  e.g. $5.1 + 6.3j$
Operations on numbers

+ , −  unary plus, minus
**  power
*, /, //, %  multiplication, division, int div, remainder
+ , −  binary: addition, subtraction

Note the precedence.

▶ −5 − 3
▶ 5 + 3 * 4 ** 2
▶ (5 + 3 * 4) ** 2
Aside: Built-in Functions

A function is piece of code that:

- accomplishes a particular task;
- can be called by referring to its name;
- usually returns a value

Functions can be

- built-in functions
- from libraries
- written by us
Built-in functions on numbers

- `abs(x)`: return the absolute value of \( x \)
  For example: \( \text{diff} = \text{abs}(x-y) \)

- `float(x)`: convert to a floating point. It must be sensible to convert \( x \) to a float.

- `int(x)`: convert to an integer (if sensible)

- `max(...)`, `min(...)`: return largest and smallest
  \( x = \text{max}(a,b,c,d) \)

- `round`: round a float to the nearest int.
  \( \text{round}(5.3) \) is 5; \( \text{round}(5.7) \) is 6.

Many others: look at the documentation.
Note the difference between integer and floating point arithmetic

\begin{verbatim}
x = 10
y = 3

print x/y

print 1.0*x/y
\end{verbatim}

There is also a \texttt{math} module that provides many functions.
Strings

Strings allow us to manipulate general alphanumeric data.

city = "Johannesburg"
protein = "alanine"
digstr = "55763"
dignum = 55763
example = 'The book is "Bioinformatics" by Krane'
etc = "don’t hesitate to try other examples"
escapechars = "Examples are tab \t and new line 

Note the difference between digstr and dignum
oaddress = """University of the Witwatersrand  
1 Jan Smuts Avenue  
Johannesburg"""

pbag = "Private Bag 3\n2050 Wits\nSouth Africa"

notraw = "this is NOT an example \n of a \t raw string \u0026"
raw = r"this is an example \n of a \t raw string \u0026"

notraw
print(notraw)
raw
print(raw)
Operations on string

- Concatenation: +
  
  ```python
  fruit = "orange"
  beverage = fruit + " juice"
  ```

- Repetition: box=fruit*12
What is the difference between:

- `x=digstr*10, x=dignum*10`
- `x=digstr+"10", x=digstr+10, x=dignum+10`
Also many builtin functions:

- `len(x)`: find the length of the string
- Each character is represented internally by a number
  - `chr(x)` returns the character represented by the number `x`;
- `ord(x)`: returns the number (ordinal value) of the *single character* `x`.
- `split`: split a string
- `str(x)`: return a string (printable) version of `x`
- Note that `int` and `float` will convert *strings* made up of digits into the appropriate number
  - Good  
    
    ```
    x=int("23213"); x=float("232.13")
    ```
  - Bad  
    
    ```
    x=int("23acc")
    ```

There is a strings module with many functions.
Simple input from keyboard

There is a special built-in function called `rawinput` that takes input from the keyboard (standard input). It returns a string.

► When executed, the program pauses and waits for input from the keyboard.
► The user enters something at the keyboard terminated by pressing the return key,
► What the user types in is returned.
# program to convert Fahrenheit to Celsius

datain = rawinput("Enter temperature in C: ")
celsius = int(datain)
fahrenheit = celsius*9/5+32
print(fahrenheit)

# or could just do

print(int(rawinput())*9/5+32)
**Boolean/Truth values**

Used to make decisions.

There are two distinguished boolean values: `True` and `False`

- operators are: `and`, `or`, and `not`

Truth tables:

<table>
<thead>
<tr>
<th>and</th>
<th>F</th>
<th>T</th>
</tr>
</thead>
<tbody>
<tr>
<td>F</td>
<td>F</td>
<td>F</td>
</tr>
<tr>
<td>T</td>
<td>F</td>
<td>T</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>or</th>
<th>F</th>
<th>T</th>
</tr>
</thead>
<tbody>
<tr>
<td>F</td>
<td>F</td>
<td>T</td>
</tr>
<tr>
<td>T</td>
<td>T</td>
<td>T</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>not</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>F</td>
<td>T</td>
</tr>
<tr>
<td>T</td>
<td>F</td>
</tr>
</tbody>
</table>

We’ll see more general use of truth values later . . .
The relational operators are most useful: $>$, $\geq$, $<$, $\leq$, $==$, $!=$

- $x < y$
  Returns True if $x$ is less than $y$, False otherwise. Numerical or lexicographic ordering is used as appropriate.
- Similarly for the other operators.
Algorithmic constructs

Python programs are made up using the following constructs

- assignments;
- sequence;
- selection;
- repetition
- calls to functions
- blocks of code.
Statement Sequence

An algorithm is a *sequence* of steps if it consists of simple steps which are to be executed one after the other.

- Start at top
- Execute each line
- in order
- exactly once
Statement Sequence

An algorithm is a sequence of steps if it consists of simple steps which are to be executed one after the other.

- Start at top
- Execute each line
- in order
- exactly once
- unless we use a Python construct – we’ll see later – that says otherwise

General rule: top-to-bottom; in-order; once

- order really matters
Example

Read in dimensions of rectangle, compute the area.

```python
a = int(rawinput("Enter the length of the rectangle: "))
b = int(rawinput("Enter the width of the rectangle: "))
area = a * b
print "Area is ",area
```
Sequence thus means

- Steps are executed one at a time
- Each step is executed exactly once.
- The order in which the steps are executed is the same as that in which they are written down.
- Termination of the last step implies termination of the algorithm.

Working out the steps is a critical part of the process.
In Python, sequencing is done by putting

- each instruction on a line by itself
- **mandatory**: each line must start in the column as the preceding line

Unlike many other languages, it is not necessary (nor idiomatic) to end a line with a semi-colon.

- If you have two statements on the same line, separate them with a semi-colon
  
x=1; y=10
  
Not idiomatic
Selection

To solve most problems we need to make choices.

- In Python most commonly with *selection*

```python
if condition:
    sequence of statements
```

Example

temp = int(rawinput)
if temp > 30:
    aircond=aircond+1
hot = true
Example

Enter two numbers into the variables $a$ and $b$ so that the larger of the two numbers is in $a$, and the smaller in $b$.

```python
put vals in a and b
if a < b:
    Swap a and b
```

NB: not a program yet . . . how is swapping done.

► must avoid losing information . . .
Using a temporary variable

```
a = int(input("Enter a: "))
b = int(input("Enter b "))
if a<b:
    tmp = a
    a = b
    b = tmp
print a, b
```
Indentation

In Python, indentation (which column a line starts) conveys meaning — a mandatory part of the syntax. The sequence of statements executed when a condition of an if statement become true are

- introduced by a colon;
- consist of all subsequence lines that are further indented
- terminated by the first statement indented at same level
Parentheses and semicolons totally ruin the feng shui of the code.

Sally learns an important lesson about inviting the right people to her code reviews.

Communications of the ACM 52(10)
if x > y:
    y = y + 1
    t = x
    m = rawinput()
if a > b: a=a+1
if len(x) > 10:
    if cond and b<a:
        z=z+1
    t = t+1
z=z+1
There is also an *else* construct.

*Example:* Read in two numbers and put the smaller into a variable called `smaller`

```python
x = int(rawinput("Enter the first number: "))
y = int(rawinput("Enter the second number: "))
if x < y:
    smaller = x
else:
    smaller = y
print(smaller)
```
Bigger and smaller

```python
x = int(rawinput("Enter the first number: "))
y = int(rawinput("Enter the second number: "))
if x < y:
    smaller = x
    larger = y
else:
    smaller = y
    larger = x
print "The smaller number is", smaller, \
    and the larger number is", larger
```
Nested selection
e.g. *find the largest of 3 numbers*

- read the numbers into $x$, $y$ and $z$
- compare the value in $x$ with the value in $y$
  - if the value in $x$ is bigger than the value in $y$
    - compare $x$ and $z$
  - otherwise
    - compare $y$ with $z$. 
x = int(input("Enter the first number: "))
y = int(input("Enter the second number: "))
z = int(input("Enter the third number: "))
if x > y:
    if x > z:
        largest = x
    else:
        largest = z
else:
    if y > z:
        largest = y
    else:
        largest = z
print "The largest number is", largest
x = int(rawinput("Enter the first number: "))
y = int(rawinput("Enter the second number: "))
z = int(rawinput("Enter the third number: "))
largest = x;
if y > largest:
    largest = y
if z > largest
    largest = z
Elif

And the elif construct allows multiple choices

```python
if not aircond:
    message = "Mild"
elif temp > 40:
    message = "Very hot"
elif temp > 30:
    message = "Hot"
elif temp > 20:
    message = "Moderate"
else:
    message = "Cold"
```
Repetition I

Repeating sections of code is essential in most algorithms. General approach:

- Identify a set of operations that are performed repetitively (but on different data each time);
- Determine how many times the code must execute or when the stopping condition is.
- Identify the essence of the loop
  - usually a variable or set of variables that records the partial answers, summarises the results, and contains the final answer when the loop finishes.
Examples

- Read in 3 numbers, find the smallest
- Read in 30000 numbers, find the smallest
- Read in a list of student marks (total number of students may not be known before) find the average.
- Ditto: but find all the students who got more than 20% above the average.
- Read in a list of student marks – find the first one who failed.
Guidelines: in designing a loop, each loop must have an

- ▶ invariant
  This is the thing that makes the loop correctly. One perspective is to see the loop as transforming the data – after each iteration we get closer to the final answer.

- ▶ variant
  This is the thing that makes the loop terminate.
Read in a set of 10 numbers and sum them up

do 10 times
  read in a number
  add it to what we have seen so far

refine this to

sum = 0
do 10 times
  x = int(rawinput())
  sum = sum+x

Refine this into Python code.
There are two major ways of looping in Python:

- **while loops**
  Loop while some condition is true — often used when we don’t know at the beginning how many times the loop will execute.

- **for loop**
  Loop over a list of things — we’ll look at in more detail later.
Basic form of the while loop

```
while cond:
    sequence of statements
```
Basic form of the while loop

```
while cond:
    sequence of statements
```

**Usual structure**

initialisation code
while cond:
    repeated operations
    change variables in the condition
Loop conditions
Can depend on data directly, or program artifacts

- e.g. loop while temp>30
  depends on program data
Loop conditions

Can depend on data directly, or program artifacts

- e.g. loop while temp > 30
depends on program data

- may introduce loop indices to control number of iterations
e.g. loop \(n\) times – have a variable \(i\) to keep track of how many times the loop might iterate.
Example: Read in 10 numbers, find the sum.

```
sum = 0
i = 0
while i < 10:
    x = int(input())
    sum = sum + x
    i = i + 1
```
Example: Read in a list of numbers terminated by a 0, find the sum.

```python
sum = 0
x = int(input())
while x != 0:
    sum = sum + x
    x = int(input())
```
Read in 10 positive numbers, and print out the largest number

```python
largest = 0
i = 0
while i < 10:
    x = int(input())
    if x > largest:
        largest = x
    i += 1
```
Read in 10 positive numbers – print out the largest number. If any non-positive numbers, print error message and terminate.

```python
largest = i = 0
x = 1
while i < 10 and x>0:
    x = int(rawinput())
    if x > largest:
        largest = x
    i=i+1
if x <= 0:
    print("A non-positive number was entered")
else:
    print(largest)
```
Find the first positive integer $x$ such that $x^3$ is greater than $1000x$.

$x=1$

while $x*x*x <= 1000*x$:
    $x = x+1$
Read in six integers $a, b, c$ and $d, e, f$. Determine whether there are any integers between $-1000$ and $1000$ such that $ax^2 + bx + c = dx^2 + ex + f$

```
a = int(rawinput())
...
f = int(rawinput())
x = -1000
while x <= 1000 and a*x*x+b*x+c != d*x*x+e*x+f:
    x = x+1
```
Another approach:

```python
a = int(rawinput())
...
f = int(rawinput())
more = True
x = -1000
while more:
    x2 = x*x
    if a*x2+b*x+c == d*x2+e*x+f or x > 1000:
        more = False
    else:
        x = x+1
```
General guidelines:

- Every variable must have a purpose: it must represent something meaningful to you, and you must know what the value of the variable means.
- formalise the continuing condition may be easier to decide terminating condition and negate
- what variables control the condition?
- what initialisation is necessary? will the loop iterate at least once? (What do you want?)
- how are the control variables changed? What will make the loop terminate?
Data structures II: Sequences and files

Sometimes we can process data ‘on-the-fly’

▶ read it in one at a time, do some processing, read the next, never have to look at the same piece of data twice.
Data structures II: Sequences and files

Sometimes we can process data ‘on-the-fly’

- read it in one at a time, do some processing, read the next, never have to look at the same piece of data twice.

But often we need to store the data all in RAM

- algorithm requires data to be processed several times;
- more natural to do so
Requirements for storing data collections

- must be able to refer to the data as a collective;
- must be able to refer to individual items;

Would like flexibility in how to structure data.

- Sequence types are one way of storing collections.
Strings, again
Strings are a sequence type:
  - sequence of characters
Given a string variable:
  - can manipulate it as a whole;
  - can access individual characters;
  - can access slices.

```
juice = "apple juice"
print juice[0]
pie = juice[0:5]+" pie"
```

Can also iterate over – will see examples.
Strings are immutable.
Example

Read sequences from input, find the index of the longest sequence, determine how many start with ACCA and how many have CGGT somewhere

Will develop in steps

- find how long the longest sequence is
- find index of longest sequence
- solve the whole problem
len_max = 0
i = 0
seq = rawinput()
while len(seq) > 0:
    if len(seq) > len_max:
        len_max = len(seq)
    seq = rawinput()
i = i + 1
print "longest sequence is length ", len_max
big_seq = 0
len_max = 0
i = 0
seq = rawinput()
while len(seq) > 0:
    if len(seq) > len_max:
        big_seq = i; len_max=len(seq)
    seq = rawinput()
i=i+1
print "longest sequence is ",big_seq
big_seq = len_max = i = numACCA = numCGGT = 0
seq = rawinput()
while len(seq) > 0:
    if len(seq) > len_max:
        big_seq = i; len_max=len(seq)
    if seq[0:4] == "ACCA": numACCA=numACCA+1
    if "CGGT" in seq: numCGGT = numCGGT+1
seq = rawinput()
i=i+1
print "longest sequence is ",big_seq
print numACCA," sequences start with ACCA"
print numCGGT," sequences contains CGGT"
Lists

Lists enable you to have a collection of Python objects – represented as a list in square brackets.

- numlist = [1,2,3,4,5,6]
- evenlist = [0,2,4,6,8,10]
- stringlist =
  ["apple","pear","banana","vampire"]
- nested = [[0,2,3],[4,1,0],[3,2,1]]
- oddsnends =
  ["banana",2j+3,-4.32,[1,2,3],true]

Individual elements and sub-ranges accessed as slices (like strings).

Lists are mutable: e.g. can say nested[1]=25
Tuples
Represented as a list in parentheses (round brackets)

- date = (3,"February",2007)

Tuples vs Lists

- **Major difference**
  Tuples are immutable: date[1]="March" is not allowed. Lists are mutable: stringlist[2]=3 is allowed.
Repetition II: Iterating over sequences

Using the `for` loop, we can iterate over any sequences. General form.

```python
for var in list:
    statement - seq
```

For each value in the list, in turn

- give the variable the value
- execute the body of the code
for ch in dnaseq:
    if ch=="C": numCs = numCs+1

for seq in seqList:
    if seq[0:4]=="ACCA": numACCAs=numACCAs+1
    for ch in seq:
        if ch=="C": numCs = numCs+1
Read in a list of DNA sequences. Print out all those above average length.

```python
seq = rawinput()
sum = num = 0
seqlist = []
while len(seq) > 0:
    seqlist.append(seq)
    sum = sum + len(seq)
    seq = rawinput()
    num = num + 1
average = float(sum) / num
for seq in seqlist:
    if len(seq) > average: print(seq)
```
List functions and helpers

One of the most useful functions is `range` which can be used to produce lists:

- `range(n)` produces the list $[0, 1, \ldots, n-1]$
- `range(i, n)` produces the list $[i, i+1, \ldots, n-1]$
- `range(i, n, d)` produces the list $i, i+d$, up to but not including $n$. \

seqlist=[]
for i in range(10):
    seq = rawinput()
    seqlist.append(seq)
sum = 0
for i in range(10):
    sum = sum+len(seqlist[i])
Some useful list and tuple functions:

- concatenation: +
- len
- append
- sort, insert, del
- Assignment: e.g. m[5]=23
  But! List element must exist already. Can’t do
  ml = ["a","b"]
  ml[10]= "c"
List comprehension
Can also create lists using list comprehension.

evens = [2*x for x in range(20)]

odds = [2*x+1 for x in range(20)]

oddsnot3 = [x for x in odds if x%3 !=0]

pairs = [(x,2*x,4*x) for x in range(20) if x%3 == 0]
Input/Output

So far we have only done simple I/O to standard input and standard output (usually keyboard and screen).
Need to be able to

- do I/O to/from files on disk
  permanent storage, much larger data
- need finer control
  do pretty output
  fancier input
print 1,2,3
print (1,2,3)
print 1,2,3,

In Python 2.x can use print to print to a file by assigning sys.stdout
Aside: unpacking lists

Often lists are useful – sometimes you want to unpack them

```python
data = ["Physics", "Chemistry", "Bioinformatics"]
print(data, sep="*")
```

▶ 

```python
["Physics", "Chemistry", "Bioinformatics"]
```

But

```python
print(*data, sep="*")
```

▶ Physics*Chemistry*Bioinformatics
Files

file variables in Python let us manipulate external data objects such as files on disk.

- The file variable is the internal representation of the external data object such as a file on disk or a pipe.
- Before using, must associate the external and internal objects. Done with the open function.
Opening files
The `open` command instructs the system to prepare the external file for reading or writing and also gives us an internal reference to that file so we can manipulate it

- give the name of the external file
- give the *mode* of opening
- store the returned reference in a file

```python
myfile = open("/tmp/seqlist.fasta")

ddfile = open("results.out","w")

mm = open("/usr/local/Hseq.inp","r")
```
Reading from a file

To read a line from a file, use the `readline` function.

- `seq = myfile.readline()`

Or, can read all the lines in the file into a list

- `seqlist = myfile.readlines()`

The latter is often more convenient

- Need to understand application
- BUT: has performance problems if the file is very large
**Example:** Write a program that counts the number of lines in a file.

```python
count = 0
f = open("mydata")
line = f.readline()
while len(line) > 0:
    count = count+1
    line = f.readline()
print("There are ",count,"lines")
```

Usual template for most programming languages. Note the structure of the loop.
Python alternative: In Python could also do

```python
f = open("mydata")
lines = f.readlines()
print "There are ",len(lines),"lines"
```

Or also

```python
count = 0
f = open("mydata")
for line in f:
    count = count+1
print "There are ",count,"lines"
```
Example

Open a file, read in sequences. Print out the first sequence that starts with ACCAG and has CCATT in it. If no such sequence, stop.

Need a loop — what is the controlling condition:

- Stop if
  we get to the end of the file; or
  the sequence read in starts with ACCAG and has CCATT in it

- Continue if this is not the case
  Continue while there is still more to read in and it’s not
  the case that the sequence read in starts with ACCAG
  and has CCATT in it
f = open("data/set1.seq")
seq = f.readline()
while len(seq)>0 and
    not(seq[0:5]=="ACCAG" and "CCATT" in seq):
    seq = f.readline()
if seq[0:5]=="ACCAG" and "CCATT" in seq:
    print(seq)
else:
    print("No such sequence")
```python
f = open("data/set1.seq")
seq = f.readline()
found = False
while len(seq)>0 and not found:
    if seq[0:5]=="ACCAG" and "CCATT" in seq:
        found = True
    else:
        seq = f.readline()
if found:
    print(seq)
else:
    print("No such sequence")
```
What about using a for loop?

- for loops are usually used when the number of times the loop will execute depends only on the size of the list — is known at the start of the loop.

- can do out-of-the-ordinary termination and skipping
  - break: leave the loop immediately.
  - continue: skip the rest of this iteration of the loop; start the next one if there is.

- very useful, but should not be over-used

- can apply to a while loop too.
```python
f = open("data/set1.seq")
for seq in f:
    if seq[0:5] == "ACCAG" and "CCATT" in seq:
        break
    if seq[0:5] == "ACCAG" and "CCATT" in seq:
        print(seq)
else:
    print("No such sequence")
```
Output
To write to a file:
  ▶ open a file with the write mode.
  ▶ use write and writelines()

outf = open("/tmp/res.dat","w")
for i in range(0,10):
    outf.write(str(i)+" "+str(i*i)+str("\n"))
outf.close()
Aside: Manipulating strings – the % operator
Particularly useful for output.
General form:

- `formatstring%(list of expressions)`
- The format string gives the form in which we want the string to appear, with % flags for place-holders;
- The expressions are the values we want.
"Capital of %s is %s\n"%("Russia","Moscow")

"Rate at hour %d was %f"%(i,r[i])
univs = ["Wits","UWC","UP","Rhodes","UCT","UKZN"]
for u in univs:
    print("Name is %s; length name is %d\n")%(u,len(u))
Read for yourself the details. Key points are:

- `%s`: string substitution
- `%d`: integer substitution
  - `%4d`: integer substitution R/J in a field of width 4
  - `%−4d`: integer substitution L/J in a field of width 4
- `%f`: float
  - `%6.3f`: float sub R/J in a field of width 6, with 3 to the right of the decimal
outf.write(('Name: %s; Surname: %s\n')%(fname,sname))

x = ('%d %d-%d\n')%(areac,exch,num)

outf.write(('Seq: %d; \t ID: %5d; \t ratio: %5.3f\n')%(i,seqid, numc/len(seq)))
## Case study – building histograms

Produce average, median, histogram from a file of marks

<table>
<thead>
<tr>
<th>Input</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>76</td>
<td>..</td>
</tr>
<tr>
<td>81</td>
<td>40-49  *</td>
</tr>
<tr>
<td>75</td>
<td>50-59</td>
</tr>
<tr>
<td>69</td>
<td>60-69  ***</td>
</tr>
<tr>
<td>72</td>
<td>70-79  ****</td>
</tr>
<tr>
<td>63</td>
<td>80-89  **</td>
</tr>
<tr>
<td>45</td>
<td>..</td>
</tr>
<tr>
<td>88</td>
<td>Ave: 63.4</td>
</tr>
<tr>
<td>69</td>
<td>Median: 65.2</td>
</tr>
<tr>
<td>78</td>
<td></td>
</tr>
</tbody>
</table>
Assumptions

All programs must make assumptions. Assumptions have impact on what you do.

- Where is the data?
- Can we assume there are always marks in the file?
- Is the file always in the right format?
- Marks integers?
- Any marks above 100?
- Any marks below 0?
Design issues

How do we solve? What data needs to be stored?

▶ Do we need to store data, or do on fly?
Design issues

How do we solve? What data needs to be stored?

► Do we need to store data, or do on fly?
► How to compute average?
Design issues

How do we solve? What data needs to be stored?

- Do we need to store data, or do on fly?
- How to compute average?
- How to compute median?
Design issues

How do we solve? What data needs to be stored?

- Do we need to store data, or do on fly?
- How to compute average?
- How to compute median?
- How to compute histogram?
Design issues

How do we solve? What data needs to be stored?

- Do we need to store data, or do on fly?
- How to compute average?
- How to compute median?
- How to compute histogram?
- How to print out histogram?
Computing average

```python
f = open("marks.dat")  # NB: bad idea to hard
n=sum = 0
for line in f:
    sum = sum+int(line)
n=n+1
print("Ave=%4.1f\n"%(sum/n))
```
Computing median and average

```python
marks = []
for line in f:
    num = int(line)
    marks.append(num)
    sum = sum + num
    n = n + 1

marks.sort()
print("Ave: %.1f"%(sum/n))
print("Median: %.1f"%(marks[n//2]))
```
Computing histogram

Use an array of counters – one element for each decile

- Create, initialise to zero
Computing histogram

Use an array of counters – one element for each decile

- Create, initialise to zero
- `histo = [0]*10`
Computing histogram

Use an array of counters – one element for each decile

- Create, initialise to zero
- `histo = [0]*10`
- As you read in a mark, determine the decile, increment the counter
num = int(line)
marks.append(num)
if 0 <= num < 10:
    histo[0] = histo[0] + 1
elif 10 <= num < 20:
.. 
..  
elif 80 <= num < 90:
    histo[8] = histo[8] + 1
elif 90 <= num <= 100:
for line in f:
    num=int(line)
    decile = num//10
    if decile==10: decile=9
    histo[decile]=histo[decile]+1

for i in range(10):
    if i < 9:
        print "%2d-%3d "%(i*10,(i+1)*10-1),
    else:
        print "%2d-%3d "%(i*10,(i+1)*10),
print "*"*histo[i]
f = open("marks.dat")

n=sum = 0
marks = []
histo = [0]*11
for line in f:
    num=int(line)
    marks.append(num)
    decile = num//10
    histo[decile]=histo[decile]+1
    sum=sum+num
    n=n+1
for i in range(10):
    start = i*10
    last = start+9
    if i==9: last=100
    print "%2d-%-3d "%(start,last),"*"*histo[i]
print
marks.sort()
print "Ave:  \%4.1f\%(sum/n)"
print "Median: \%4.1f\%(marks[n/2])"
Modularity and sub-algorithms

Divide-and-conquer is a key design principle
Modularity and sub-algorithms

Divide-and-conquer is a key design principle

- For everything but the most trivial program, the design phase will break the problem into parts.

Programming languages provide useful constructs for this:

- promote good design and implementation techniques;
- promote code re-use
Functions

Functions (also known as methods, procedures, routines, sub-routines) are sections of named, separately callable code:

- the same piece of code may be called in several places;
- can structure your code to reflect the design

We have seen examples:

- `len(x)`
- `f.readlines()`

There are built-in functions, functions in libraries, libraries you write yourself.
Calling a function

To call a function, you give its name and its argument list.

- execution transfers to the code of the function;
- the values of the argument are passed to the function;
- the function then executes using the given values;
- when the function finishes executing, control returns to the caller.

- function may return a value. \( x = \text{len}(\text{datalist}) \)

Note:
- one function can call another function and so on.
In Python we see two ways of referring to functions that reflect different perspectives of data.

- **passive view**: the data is a passive thing on which we perform operations.
  \[ x = \text{len}(\text{datalist}) \]

- **active view**: the data is an active thing and the operations that are relevant to the data are parts of its definition.
  \[ x.\text{capitalize}(); \ x.\text{center}(20) \]
  often known as *methods*
Functions from Modules

A module is a collection of related functions (possibly implementing a class – will talk about later).

To use a module, must

- import the whole module
- use the functions in the module, qualifying the names; or
- import the functions required from the module; or
- use a function to create an object of the type that the module defines.
  invoke appropriate methods on that object.

Modules can also contain constants
import math
from time import clock

start = clock() # cpu usage not wall-clock
radius = 20.0
area = math.pi*radius**2

l = math.log(radius)
k = math.log(radius,2)

for i in range(1000):
    print k,l

finish = clock()
print "This took ",finish-start
Example using csv module

CSV (comma-separated-values) files
- Produced by most spreadsheet programs like Excel

David,78272782,25,38,57
Barack,8228728,82,73,91,75
Hosni,726276,5,3,12,16,10,9,17

We want to compute the average of each of the students.
import csv

classfile = open("classlist")
csvfile = csv.reader(classfile)

for student in csvfile:
    name = student[0]
    number = student[1]
    sum=n=0
    for mark in student[2:]:
        sum=sum+int(mark)
        n = n + 1
    print "Average of %s was %5.1f"%(name,float(sum)/n)
from datetime import date

# create a new date object
d = date.today()

print("Today is: ", d.day,"/",d.month,"/",d.year)
print(d.isoformat())

dayofweek = d.weekday()
if dayofweek not in [0,6]:
    print("Work harder!!!!")
Defining new functions

You can write your own functions:

```python
def functionname(list of args):
    code
```

Functions usually return a value: use the `return` keyword.

- When the `return` keyword is reached, execution terminates and control passes back to where called.
- A value can be returned to the caller
- Can have multiple returns – may make things complex
- If no `return` given, then a special value `None` returned.
def count_vowels(s):
    count=0
    for ch in s:
        if ch.upper() in ["A","E","I","O","U"]:
            count=count+1
    return count

print "Please enter string: "
myinp = raw_input()
c = count_vowels(myinp)
print "There are %d vowels\n"%c
def double(x):
    return 2*x

def print_row(i):
    outs = ("Number is \%5d, double is \%5d")%\\
    (i,double(i))
    print outs

for j in range(20): print_row(j)
def count_chars(seq,target):
    count=0
    for ch in seq:
        if ch==target: count=count+1
    return count

def count_chars_in_file(fname,t):
    count = 0
    f = open(fname)
    for seq in f:
        count=count+count_chars(seq,t)
    return count

print count_chars_in_file("see.dat","C")
Parameter/argument passing

Parameters given in two places:
- where the function is defined — *formal* arguments.
- where the function is called – *actual* arguments

When function called, actual, formal arguments bound:
- first actual argument bound to first formal, etc.
- values copied across
- after that no link between parameters (but see later... it’s not quite so simple)

Each time a function is called there is a fresh binding – there is no “memory”
your functions can call other functions;
they can call themselves – recursion (we’ll see this later);
functions can be nested
functions have their own name space
will have a look at this later in more detail
if a variable is assigned a value in a function it is local to that function
if a variable only has its value used, it is a global variable and must be given a value globally before the function is called.
Functions and good design

- let the structure of your program reflect the structure of your solution;
- can work top-down:
  know you need a function to accomplish a task
  build a stub, come back to the problem later
- can work bottom-up:
  build a function to accomplish a small task, test it separately
  then work upwards

Big-bang program design and implementation is **not** a good idea.
Some basic Python Modules

There are hundreds of Python module.

- Must know some well
- Have an overview of what’s out there
- Use the documentation
The *pickle* module

Useful for I/O of data to/from disk for
- want program data from one run of program to be used in another run of program
The *pickle* module

Useful for I/O of data to/from disk for

- want program data from one run of program to be used in another run of program
- problem addressed: easy to print out and subsequently read simple data, but not so easy for complex data (e.g. dictionaries, storing lists of tuples).
The *pickle* module

Useful for I/O of data to/from disk for

- want program data from one run of program to be used in another run of program
- problem addressed: easy to print out and subsequently read simple data, but not so easy for complex data (e.g. dictionaries, storing lists of tuples).
- data not intended to be human readable on disk
The *pickle* module

Useful for I/O of data to/from disk for

- want program data from one run of program to be used in another run of program
- problem addressed: easy to print out and subsequently read simple data, but not so easy for complex data (e.g. dictionaries, storing lists of tuples).
- data not intended to be human readable on disk
- don’t need to interface with non-Python programs
Dumping data

```python
f = open("prog.dat","w")
x = [1,2,("a","b")]
y = {}
pickle.dump(x,f)
pickle.dump(y,f)
```

Reading data

```python
f = open("prog.dat")
x=pickle.load(f)
y=pickle.load(f)
```
The sys module

Allows system specific information to be retrieved. Most very technical. Some of the useful ones are

- `sys.exit(arg)`: Leave the current program immediately.
  - `arg` can be numeric: 0 is good; 1–127 is bad.
  - Can also be a string which may be printed out.
- `sys.path`: A list of places Python looks to import code
- `sys.platform`
Sometimes you want to pass your program: can use list sys.argv
Suppose this is the file see.py

import sys

print(sys.argv)

and it is run by

python see.py a b c d e

Then sys.argv is the list ['see.py', 'a', 'b', 'c', 'd']
Our histogram example, could now be written as

```python
import sys
f = open(sys.argv[1])
```

And then when the program is run, we give it the file name containing the marks.
sys.stdin, sys.stdout, sys.stderr
Say where default I/O is done
  ▶ Can use, change

if file_output:
    f=open(fname,"w")
else:
    f=sys.stdout

f.write(-answer)
Regular expressions: re

Support for regular expressions is common in scripting languages:

- generalised form of string matching;
- useful for finding text, processing and replacing supplements methods on strings

Example:

- using strings it’s easy to find particular text, but not patterns, e.g.:
  capitalise every letter after a full stop followed by a space
  replace \begin{xxxx} with <xxxx>
  find all numbers to the right of an equals sign.
Specifying Regular Expressions

- Most characters stand for themselves: F Fred 6312
Specifying Regular Expressions

- Most characters stand for themselves: F Fred 6312
- `\ | ( ) [ { ~ $ * + ? .` are metacharacters (have special meaning)
  - escape with a backslash for the chars:
    - `\(Fred\)` stands for the string `(Fred)`
Specifying Regular Expressions

- Most characters stand for themselves: F Fred 6312
- \ | ( ) [ { ^ \* + ? . are metacharacters (have special meaning)
  - escape with a backslash for the chars:
    \(Fred\) stands for the string (Fred)
- To group things together, use parentheses. Groups can be referred to.
To specify alternatives, use |. `green|red apples` stands for `green` or `red apples`.
To specify alternatives, use `|`. 
.green|red apples stands for *green* or *red apples*  
(green|red) apples stands for *green apples* or *red apples*
To specify alternatives, use |. 

green|red apples stands for *green* or *red apples*

(green|red) apples stands for *green apples* or *red apples*

A list of characters in square brackets matches any of the characters.

- [YyNn] matches any of an upper or lower case “y” or “n”.
- [A-Za-z0-9] is all the alphanumeric characters
\n new line; \t tab; \s a whitespace;
- \n new line; \t tab; \s a whitespace;
- \d digit; \D non-digit;
- \n new line; \t tab; \s a whitespace;
- \d digit; \D non-digit;
- \w a word charater, \W a non-word character
Some basic Python modules

Regular expressions: re

- \n new line; \t tab; \s a whitespace;
- \d digit; \D non-digit;
- \w a word character, \W a non-word character
- . anything but a \n
- \n new line; \t tab; \s a whitespace;
- \d digit; \D non-digit;
- \w a word character, \W a non-word character
- . anything but a \n
- m\{3\} stands for *mmm*
  (map){2,3} stands for *mapmap* or *mapmapmap*
- m* stands for 0 or more m’s
Some basic Python modules

Regular expressions: `re`

- \n new line; \t tab; \s a whitespace;
- \d digit; \D non-digit;
- \w a word character, \W a non-word character
- . anything but a \n
- m\{3\} stands for `mmm`
  (map)\{2,3\} stands for `mapmap` or `mapmapmap`
- m* stands for 0 or more m’s
- m+ stands for 1 or more m’s
\n new line; \t tab; \s a whitespace;
\d digit; \D non-digit;
\w a word character, \W a non-word character
. anything but a \n
m{3} stands for mmm
(map){2,3} stands for mapmap or mapmapmap
m* stands for 0 or more m’s
m+ stands for 1 or more m’s
\k (where k is a number, matches the n-th group)
Many others. Rules long and complex in full generality, but easy to use basic REs
Using regular expressions

The `re` module provides extensive regular expression support

`search(pattern,string)`

Look for the pattern in the string.

```python
mtch = re.search(r"\(\w+) \1",data):

Returns a MatchObject or None
```
split(pattern, string)

Break a string into a list of strings

>>> re.split("\s\[;\-\]\s","a-b - c ; d")
[’a-b’, ’c’, ’d’]

findall(pattern, string)

return a list of matches

>>> re.findall("(\d+)\","(1) now (x) app (13) next")
[’(1)’, ’(13)’]
sub(pattern, replacement, data)
Substitute some pattern for another.

re.sub("(.+), (.+), (.+)",
       r"Number=\1\nName: \3 \2", data)
The matchobject is a very useful data type:

```python
mtch = re.search("caller (\w+); num (\w+); charge:(\d+),\n data)
```

- if the search does not succeed, you get back `None`
- otherwise, `mtch` can be used to extract out parts of the match

```python
mtch.group(0)  # entire string matched
mtch.group(1)  # the first group (caller)
mtch.group(2)  # the next
```
The *os* module

provides interaction with the operating system, including file and process management. Some of the useful features are

- `access(path, mode)` access to the path/file?
- `chdir(path)`, `getcwd()`: change, get the current working directory
The **os** module

provides interaction with the operating system, including file and process management. Some of the useful features are

- `access(path, mode)` access to the path/file?
- `chdir(path), getcwd()` change, get the current working directory
- `listdir(path)` list directory
The os module

provides interaction with the operating system, including file and process management. Some of the useful features are

- `access(path, mode)` access to the path/file?
- `chdir(path), getcwd()` change, get the current working directory
- `listdir(path)` list directory
- `walk(dirname)` generator of hierarchical directory structure
The os module

provides interaction with the operating system, including file and process management. Some of the useful features are

▶ `access(path, mode)` access to the path/file?
▶ `chdir(path), getcwd()` change, get the current working directory
▶ `listdir(path)`: list directory
▶ `walk(dirname)`: generator of hierarchical directory structure
▶ change modes, groups; make, delete directories, links; rename files; get file information (`stat`)
The os module

provides interaction with the operating system, including file and process management. Some of the useful features are

- `access(path, mode)`: access to the path/file?
- `chdir(path), getcwd()`: change, get the current working directory
- `listdir(path)`: list directory
- `walk(dirname)`: generator of hierarchical directory structure
- change modes, groups; make, delete directories, links; rename files; get file information (`stat`)
times returns the time the process has taken up to now (use the time module to get info/perform operations on time)
Some basic Python modules

- **os module**
  - `times` returns the time the process has taken up to now (use the `time` module to get info/perform operations on time)
  - `system`: run a system command

- `popen`: run another process and open up a pipe
  - Allows execution of the command, and reading the results.
Some basic Python modules

- `os` module
  - `times` returns the time the process has taken up to now (use the `time` module to get info/perform operations on time)
  - `system`: run a system command
    ```python
    os.system("cp f1.dat /tmp/f2.dat")
    ```
  - `popen`: run another process and open up a pipe
Some basic Python modules

- The `os` module

  - `time`: returns the time the process has taken up to now (use the `time` module to get info/perform operations on time)

  ```python
  os.system("cp f1.dat /tmp/f2.dat")
  ```

  - `popen`: run another process and open up a pipe

    Allows execution of the command, *and* reading the results.

- Many others
More Python

1. dictionaries
2. more on functions
3. sets
Dictionaries

Known in other languages as hash tables, or associative maps. generalised form of a list:
  ▶ the index can be any immutable type

```
os.environ['PATH']
os.environ['HOME']
```

```
winner['Wimbledon'] = 'Williams'
```
table = {}
table["sat"] = "saturday"
table["sun"] = "sunday"
table["mon"] = 12

table[3] = "no day"
table[(1,2)] = "tuple1"
table[(1,4)] = [1,2,3,4]

print(table[3])
>> 'no day'
print(table["sat"])  
>> 'saturday'
print(table[1,2])    
>> 'tuple1'
print(table[(2,3)])  
>> keyerror: (2, 3)
    resp = {"y": "yes", "y": "yes", "n": "no", 3:[4,7,"africa"]}
print(resp["n"])
'no'
There are a number of useful methods available for tables.

```python
>>> len(table)
6
>>> table.keys()
[3, 'sun', 'mon', (1, 2), (1, 4), 'sat']
>>> table.values()
['no day', 'sunday', 12, 'tuple1', [1, 2, 3, 4], 'saturday']
>>> table.items()
[(3, 'no day'), ('sun', 'sunday'), ('mon', 12), ((1, 2), 'tuple1'), ((1, 4), [1, 2, 3, 4]), ('sat', 'saturday')]
>>> table.has_key("mon")
false
>>> table.has_key("mon")
true
```
Example use of dicts

We have a list of email addresses we have written to. Summarise usage.

*Input*

vc@wits.ac.za
complaints@wits.ac.za
vc@wits.ac.za
complaints@wits.ac.za
vc@wits.ac.za
complaints@wits.ac.za
my.friend@gmail.com
complaints@wits.ac.za

*Output*

complaints@wits.ac.za 3
my.friend@gmail.com 1
vc@wits.ac.za 2
summary = {}
f = open(sys.argv[1])
for email in f:
    email = email.rstrip("\n");
    if summary.has_key(email):
        summary[email] = summary[email] + 1
    else:
        summary[email] = 1
for k in sorted(summary.keys()):
    print "%-30s %s"%(k, summary[k])
Sets

Can use lists for sets, but the set type and frozenset types are preferred.

```python
>>> a = set([1, 2, 3, 4, 5, "apple"])
set([1, 2, 3, 4, 5, 'apple'])
>>> len(a)
6
>>> a.union([5,"cherry"])
set([1, 2, 3, 4, 5, 'cherry', 'apple'])
>>> b = set([4,5,"apple","cherry"])
>>> a.intersection(b)
set([4, 5, 'apple'])
```

Usual types of set operations available.
Functions in Python

While not first class citizens in python, functions can be manipulated in a number of ways:

Higher order functions
functions that take other functions as arguments.

Examples:
- map: apply a function to a list
- filter: use a function to select items from a list
- reduce: use a function to combine a list
map
Applies a function to list of items

```python
x = map(sqrt,[10,3,40,5])
```
filter

use a boolean function to select items
Applies the function to each item in the list and returns a new list:

▶ if true result, put in returned list
▶ if false result, left out of returned list

```python
def passmark(x):
    return x >= 50
```

```python
numpassed = len(filter(passmark, marks))
```
functools.reduce
combine a list with given binary function

- reduce(func, list): apply the function to the first and second items in the list to get a result. then repeatedly use computed result and elements in the list

def concat(x, y): return x+y

reduce(concat, ["apple", "pie", "is", "good"])
reduce(func, list, init): as above but use init as the starting point

reduce(concat, ["apple", "pie", "is", "good"], "do you think ")

*do you think applepieisgood*
Lambda forms: anonymous functions

- higher order functions require functions as argument, but often function wanted
  - hasn’t been defined already
  - has very specific purpose so will only be used in one place
  - don’t want overhead of defining function first
Lambda forms: anonymous functions

- higher order functions require functions as argument, but often function wanted
  - hasn’t been defined already
  - has very specific purpose so will only be used in one place
  - don’t want overhead of defining function first
- solution: anonymous function – can define it where we need it
Lambda forms: anonymous functions

- higher order functions require functions as argument, but often function wanted
  - hasn’t been defined already
  - has very specific purpose so will only be used in one place
  - don’t want overhead of defining function first
- solution: anonymous function – can define it where we need it
- general forms: lambda args: expression
Lambda forms: anonymous functions

- Higher order functions require functions as argument, but often function wanted
  - hasn’t been defined already
  - has very specific purpose so will only be used in one place
  - don’t want overhead of defining function first
- Solution: anonymous function – can define it where we need it
- General forms: lambda args: expression
More python

Functions in Python

► lambda x, y: x+y
\[ \text{lambda } x, y: x+y \]
\[ \text{reduce}(\text{lambda } x, y: x+y, \text{list}) \]
▶ lambda x, y: x+y
▶ reduce(lambda x, y: x+y, list)
▶ filter(lambda mark: mark>=50, marklist)
- lambda x, y: x+y
- reduce(lambda x, y: x+y, list)
- filter(lambda mark: mark>=50, marklist)
- map usage: map(lambda x: 9*x/5+32, temps)
- `lambda x, y: x+y`
- `reduce(lambda x, y: x+y, list)`
- `filter(lambda mark: mark>=50, marklist)`
- `map usage: map(lambda x: 9*x/5+32, temps)`
- `filter usage`
▶ lambda x, y: x+y
▶ reduce(lambda x, y: x+y, list)
▶ filter(lambda mark: mark>=50, marklist)
▶ map usage: map(lambda x: 9*x/5+32, temps)
▶ filter usage

marklist = [(' 0233663 ', 58), (' 0241713 ', 82), (' 0321878 ', 42), ...]
numfirsts = len(filter(lambda (number, mark): mark>=75, marklist))
lambda x, y: x+y
reduce(lambda x, y: x+y, list)
filter(lambda mark: mark>=50, marklist)
map usage: map(lambda x: 9*x/5+32, temps)
filter usage

marklist =
["0233663", 58], ["0241713", 82], ["0321878", 42]...
numfirsts = len(filter(lambda (number, mark): mark >= 75, marklist))
Functions as variables

```python
if peaktime
    rate = math.sqrt
else
    rate = math.log
charge = rate(minutes)
```

```python
def find_max(list, checkfn):
    best = list[0] \n    for x in list[1:]
        if checkfn(best, x) best = x

find_max(list, lambda x, y: x[0] < y[0])
find_max(list, lambda x, y: x[1] < y[1])
```
Also supports the table-driven approach – elegant, easy to change

def calfun(): ...
def inputproc(): ...
def calc():......

actiontable = {calibrate: calfun, read: inputproc}
allowable = actiontable.keys()
request = lower(input("enter choice: "))
if request in allowable
    actiontable[request]()
Note the difference between

\[ x = f.\text{readline}() \]
\[ y = f.\text{readline} \]
Note the difference between

\[ x = f.\text{readline}() \]
\[ y = f.\text{readline} \]

- \( x \) will be a string – a value read in from a file

\[ a = y() \]
\[ b = y() \]
Note the difference between

\[ x = f\text{.readline}() \]
\[ y = f\text{.readline} \]

- \( x \) will be a string – a value read in from a file
- \( y \) will be a \textit{function}

\[ a = y() \]
\[ b = y() \]
Note the difference between

```python
x = f.readline()
y = f.readline
```

- `x` will be a string – a value read in from a file
- `y` will be a *function*
- which you can call, e.g.

```python
a=y()
b=y()
```
Case study: UPGMA phylogenetic tree

Phylogenetic tree:
- tree that describes the relationship of a set of objects (DNA sequences)
- *leaves* of the tree are a known set of objects;
- *internal nodes* are (usually) unknown, putative ancestor objects;
- structure of tree reflects the evolutionary history.
Case study: UPGMA phylogenetic tree

Phylogenetic tree:
- tree that describes the relationship of a set of objects (dna sequences)
- leaves of the tree are a known set of objects;
- internal nodes are (usually) unknown, putative ancestor objects;
- structure of tree reflects the evolutionary history.

Problem: given a set of DNA sequences, infer the evolutionary history

- input: set of sequences
- output: phylogenetic tree
UPGMA: unweighted pair group method using arithmetic means
  ▶ popular phylogenetic technique
  ▶ fast, but questions about quality
NB: case study on algorithm design, not phylogenetics
UPGMA: unweighted pair group method using arithmetic means

- popular phylogenetic technique
- fast, but questions about quality
  NB: case study on algorithm design, not phylogenetics
- based upon *distance* between sequences
UPGMA: unweighted pair group method using arithmetic means

- popular phylogenetic technique
- fast, but questions about quality
  NB: case study on algorithm design, not phylogenetics
- based upon distance between sequences
  - sequence distance implies evolutionary distance
UPGMA: unweighted pair group method using arithmetic means

- popular phylogenetic technique
- fast, but questions about quality
  NB: case study on algorithm design, not phylogenetics
- based upon distance between sequences
  - sequence distance implies evolutionary distance
  - use distance to build tree
- different ways to compute
Hierarchically cluster
Hierarchically cluster

- Case study: UPGMA phylogenetic tree
Hierarchically cluster

0 1

2 3

4

5

6 1 2 3 4 5 7

build tree: leaves are data points, internal nodes ancestors

Case study: UPGMA phylogenetic tree
Hierarchically cluster

[Diagram showing a hierarchical clustering of data points with a phylogenetic tree structure]
Hierarchically cluster
Hierarchically cluster

build tree: leaves are data points, internal nodes ancestors
Basic algorithm

read $n$ sequences $s_0, \ldots, s_{n-1}$
compute distances between all sequences
$k = n$
repeat
    choose unprocessed $s_i, s_j$ with smallest distance
    add new ‘sequence’ $s_k$ as parent of $s_i, s_j$
    compute distance of $s_k$ to all other sequences
    $k = k + 1$
Distance between two nodes is the average of the distance between leaves in the sub-tree rooted by the nodes.

\[ d(x, y) = \sum_{x \in x, y \in y} d(x, y) / |x||y| \]
Introduction to Programming using the Python Programming Language

Case study: UPGMA phylogenetic tree
Case study: UPGMA phylogenetic tree

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>4</td>
<td>20</td>
<td>24</td>
<td>20</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
<td>30</td>
<td>26</td>
<td>48</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td>8</td>
<td>18</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td>14</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Graph:

- Nodes 0, 1, 2, 3, 4 are connected to node 5.
Case study: UPGMA phylogenetic tree

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>4</td>
<td>20</td>
<td>24</td>
<td>20</td>
<td>2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>30</td>
<td>26</td>
<td>48</td>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>8</td>
<td>18</td>
<td>25</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>14</td>
<td>25</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>34</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Diagram showing the UPGMA phylogenetic tree.
Introduction to Programming using the Python Programming Language

Case study: UPGMA phylogenetic tree

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td>* 0</td>
<td></td>
<td>4</td>
<td>20</td>
<td>24</td>
<td>20</td>
<td>2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>* 1</td>
<td></td>
<td></td>
<td>30</td>
<td>26</td>
<td>48</td>
<td>2</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>8</td>
<td>18</td>
<td>25</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>14</td>
<td>25</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>8</td>
<td>18</td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>25</td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>34</td>
</tr>
</tbody>
</table>

```
      5
     /\  \
    0 1 2

    6
   /\  \
  2 3 4
```

---

* 2

---

2

---

22

---

4

---

20

---

48

---

25

---

34

---

25

---

4

---

28

---

4

---

8

---

18

---

25

---

34

---

28

---

8

---

* 2

---

2

---

25

---

6

---

16

---

8

---

26

---

48

---

20

---

24

---

20
### Introduction to Programming using the Python Programming Language

**Case study: UPGMA phylogenetic tree**

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td>*0</td>
<td>4</td>
<td>20</td>
<td>24</td>
<td>20</td>
<td>2</td>
<td>22</td>
<td></td>
<td></td>
</tr>
<tr>
<td>*1</td>
<td></td>
<td>30</td>
<td>26</td>
<td>48</td>
<td>2</td>
<td>28</td>
<td></td>
<td></td>
</tr>
<tr>
<td>*2</td>
<td></td>
<td></td>
<td>8</td>
<td>18</td>
<td>25</td>
<td>4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>*3</td>
<td></td>
<td></td>
<td></td>
<td>14</td>
<td>25</td>
<td>4</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Diagram:**

```
0 -- 5  
  |    
1 -- 6  
  |    
2 -- 7  
  |    
3 -- 4  
```
Introduction to Programming using the Python Programming Language

Case study: UPGMA phylogenetic tree

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>4</td>
<td>20</td>
<td>24</td>
<td>20</td>
<td>2</td>
<td>22</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>30</td>
<td>26</td>
<td>48</td>
<td>2</td>
<td>28</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>8</td>
<td>18</td>
<td>25</td>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>14</td>
<td>25</td>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>34</td>
<td>16</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>25</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Case study: UPGMA phylogenetic tree

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td>*0</td>
<td>4</td>
<td>20</td>
<td>24</td>
<td>20</td>
<td>2</td>
<td>22</td>
<td></td>
<td></td>
</tr>
<tr>
<td>*1</td>
<td>30</td>
<td>26</td>
<td>48</td>
<td>2</td>
<td>28</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>*2</td>
<td>8</td>
<td>18</td>
<td>25</td>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>*3</td>
<td>14</td>
<td>25</td>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>*4</td>
<td></td>
<td></td>
<td>34</td>
<td>16</td>
<td>8</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>*5</td>
<td></td>
<td></td>
<td></td>
<td>25</td>
<td>28</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>*6</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>8</td>
<td></td>
<td></td>
</tr>
<tr>
<td>*7</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Case study: UPGMA phylogenetic tree
Data structures

conceptually: what data do we need to represent? what is the relationship between data?
Data structures

*conceptually*: what data do we need to represent? what is the relationship between data?

- sequences
- distances between sequences
Data structures

*conceptually:* what data do we need to represent? what is the relationship between data?

- sequences
- distances between sequences
- tree
Data structures

*conceptually:* what data do we need to represent? what is the relationship between data?

- sequences
- distances between sequences
- tree
- descendants
Data structures

conceptually: what data do we need to represent? what is the relationship between data?

- sequences
- distances between sequences
- tree
- descendants
now how we do implement this . . . may need to make compromises

- sequences
now how we do implement this . . . may need to make compromises

▶ sequences
  list of strings . . . easy
now how we do implement this . . . may need to make compromises

▶ sequences
  list of strings . . . easy
▶ distance between sequences
now how we do implement this . . . may need to make compromises

- sequences
  list of strings . . . easy

- distance between sequences
  list of lists of integers
  \[
  \begin{bmatrix}
  0 & 10 & 15 \\
  10 & 0 & 20 \\
  15 & 20 & 0
  \end{bmatrix}
  \]
  can access individual elements:
  \[d[2][0] = 5\]
  the only issue: entries must exist! need to set up matrix at beginning
trees are a common data structure and there are many ways to represent them.
trees are a common data structure and there are many ways to represent them.
formally a tree is a
  ▶ leaf; or
trees are a common data structure and there are many ways to represent them. Formally, a tree is a

- leaf; or
- interior node with one or more children trees
trees are a common data structure and there are many ways to represent them.
formally a tree is a
- leaf; or
- interior node with one or more children trees

some data structures:
trees are a common data structure and there are many ways to represent them. formally a tree is a

- leaf; or
- interior node with one or more children trees

some data structures:

- *parent* array: for element we keep the index of its parent in the tree
trees are a common data structure and there are many ways to represent them. Formally a tree is a

- leaf; or
- interior node with one or more children trees

Some data structures:

- *parent* array: for element we keep the index of its parent in the tree
- *children* array: for each element we keep a list or tuple of its children
trees are a common data structure and there are many ways to represent them. formally a tree is a

- leaf; or
- interior node with one or more children trees

some data structures:

- parent array: for element we keep the index of its parent in the tree
- children array: for each element we keep a list or tuple of its children
- dynamic structures: using python’s oo features – we’ll see that later
trees are a common data structure and there are many ways to represent them. Formally a tree is a

- leaf; or
- interior node with one or more children trees

Some data structures:

- parent array: for element we keep the index of its parent in the tree
- children array: for each element we keep a list or tuple of its children
- dynamic structures: using Python’s oo features – we’ll see that later
leaf-descendants of each node:

- keep a list for each element – contains the leaf descendants
leaf-descendants of each node:

▶ keep a list for each element – contains the leaf descendants
▶ this is redundant
leaf-descendants of each node:
  ▶ keep a list for each element – contains the leaf descendants
  ▶ this is redundant

has a sequence been processed:
  ▶ could keep a list of processed or unprocessed, or
leaf-descendants of each node:
  ▶ keep a list for each element – contains the leaf descendants
  ▶ this is redundant

has a sequence been processed:
  ▶ could keep a list of processed or unprocessed, or
  ▶ keep a boolean list: for each element we record whether it has been processed
leaf-descendants of each node:

- keep a list for each element – contains the leaf descendants
- this is redundant

has a sequence been processed:

- could keep a list of processed or unprocessed, or
- keep a boolean list: for each element we record whether it has been processed

we choose latter
leaf-descendants of each node:

- keep a list for each element – contains the leaf descendants
- this is redundant

has a sequence been processed:

- could keep a list of processed or unprocessed, or
- keep a boolean list: for each element we record whether it has been processed

we choose latter

note the choices to be made — need to make choices, trade off space, time, ease of programming
Outline of program

- start with very abstract view
- focus on critical issues
  think about what is needed
- focus on interface
Outline of program

- start with very abstract view
- focus on critical issues
  think about what is needed
- focus on interface

what is the input:
Outline of program

- start with very abstract view
- focus on critical issues
  think about what is needed
  focus on interface

what is the input:
- name of the file of sequences — pass as command line
  parameter to the program,

what is the output?
- a nice picture of the tree.
High-level program

```python
read(sequences)
dist = compute_distances(sequences)
tree = make_tree(sequences, dist)
display(tree)
```
plan:

▸ implement the read sequences and test
plan:

- implement the read sequences and test
- implement the make tree algorithm
plan:

- implement the read sequences and test
- implement the make tree algorithm build a *stub* for the compute distance
  - difficult to test two things at once – where is the error
  - difficult to make test cases with raw data. easier with distance matrix.
- implement the compute distance & test
- draw pretty pictures
Some revision

We know how to open a file, read from the file and close the file.
Some revision

We know how to open a file, read from the file and close the file.

▶ file not just something on disk – can be a more abstract object
▶ e.g. unix command: wc u393.tex
Some revision

We know how to open a file, read from the file and close the file.

▶ file not just something on disk – can be a more abstract object
▶ e.g. unix command: `wc u393.tex`
  
    `cat u393.tex | wc`
Some revision

We know how to open a file, read from the file and close the file.

- file not just something on disk – can be a more abstract object
- e.g. unix command: \texttt{wc u393.tex}
  \begin{verbatim}
  cat u393.tex | wc
  \end{verbatim}
  or \texttt{ls -1 | wc}
- a \textit{pipe} connects the output of one program to the input of another.
Some revision

We know how to open a file, read from the file and close the file.

▸ file not just something on disk – can be a more abstract object
▸ e.g. unix command: wc u393.tex
  cat u393.tex | wc
  or ls -1 | wc
▸ a pipe connects the output of one program to the input of another.
▸ in python, we can
  ▸ run a command
Some revision

We know how to open a file, read from the file and close the file.

- file not just something on disk – can be a more abstract object
- e.g. unix command: `wc u393.tex`
  ```
  cat u393.tex | wc
  ```
  or `ls -1 | wc`
- a *pipe* connects the output of one program to the input of another.
- in python, we can
  - run a command
  - open a pipe to that command
Some revision

We know how to open a file, read from the file and close the file.

- file not just something on disk – can be a more abstract object
- e.g. unix command: wc u393.tex
  cat u393.tex | wc
  or ls -1 | wc
- a *pipe* connects the output of one program to the input of another.
- in python, we can
  - run a command
  - open a pipe to that command
  - treat the pipe as a file
- we’ll use the simplest case – reading from a pipe.
f = os.popen(cmdname)

- run the command given.
- the output of the command goes to a ‘virtual file’, which we refer to as f.
- this is treated in our program as a file.

for example:

```
dir = popen("ls -1")
count=0
for fname in dir: count=count+1
dir.close()
```
- `os.system(expression)`
  ask the shell to execute the given command.
  return value is the system success code.
os.system(expression)
ask the shell to execute the given command.
return value is the system success code.

os.environ
python dictionary – can access/manipulate system environment variables.
Implementation

**reading in sequences:** use the following assumptions

- read from a file — know the file name
- don’t know upfront how many sequences there’ll be
- each sequence will be on a line by itself
- at the end, when we try to read, we’ll get an empty string
Implementation

**reading in sequences:** use the following assumptions

- read from a file — know the file name
- don’t know upfront how many sequences there’ll be
- each sequence will be on a line by itself
- at the end, when we try to read, we’ll get an empty string

think:

- what about alternatives/if the assumptions don’t hold
- what possible error conditions?
Assumptions lead to the use of a while loop.

def read_sequences(fname):
    f = open(fname)
    seq = []
    curr = f.readline()
    for line in f:
        if line[0]=">":
            seq.append(curr)
            curr=line
        else:
            curr=curr+line
    seq.append(curr)
    f.close()
    return seq

after writing, can test!
Making the tree

- do initialisation
  - descendants
  - processed
- build tree
  - find smallest entry in distance matrix
  - update tree
  - update distance matrix
def make_tree(n, dist):
    descendants = [[] for i in range(2*n-1)]
    for x in range(n): descendants[x] = [x]
    children = [(-1, -1)] * (2*n-1)
    toprocess = [true]*(2*n-1)
    for x in range(n, 2*n-1):
        (i, j) = get_min(dist, processed, x)
        toprocess[i] = false
        toprocess[j] = false
        children[x] = (i, j)
    descendants[x] = descendants[i] + descendants[j]
    for k in range(0, x)
        if toprocess[k]:
            dist[k][x] = dist[x][k] = \
                compute_new_dist(k, x, dist, descendants)
    return children
Finding the minimum...

▶ m = min (list)
Finding the minimum...

- \( m = \text{min} \ (\text{list}) \)
- or ...

\[
\text{minelt} = \text{list}[0]
\]
for \( x \) in \( \text{list}[1:] \)
\[
\text{if } x < \text{minelt}: \quad \text{minelt} = x
\]
- or ...

\[
\text{minelt} = \text{sys.maxsize}
\]
for \( x \) in \( \text{list} \)
\[
\text{if } x < \text{minelt}: \quad \text{minelt} = x
\]
Must find *indices* of where the min element is, only look at unprocessed rows, cols
Must find *indices* of where the min element is, only look at unprocessed rows, cols

```python
def get_min(dist, processed, x):
    the_i = the_j = 0
    min = 10000000
    for i in range(x-1):
        if toprocess[i]:
            for j in range(i+1, x):
                if toprocess[j]:
                    if dist[i][j]<min:
                        min = dist[i][j]
                        the_i = i
                        the_j = j
    return (the_i, the_j)
```
Computing a new distance

```python
def compute_new_dist(x, y, dist, descendants):
    sum = 0
    for a in descendants[x]:
        for b in descendants[y]:
            sum = sum + dist[a][b]
    return sum / (len(descendants[x]) * len(descendants[y]))
```

Only works if $x, y$ not in ancestor/descendant relationship

▸ but not worth checking for since we never use the distance if $x, y$ are both in the same branch of the tree
**Stub Computing distances**

Instead of reading the sequences, we read in distance matrix. All that the compute distance function does is extract out the data.

```python
def compute_distance(sequence):
    n = len(sequence)
    m = 2*n - 1
    dist = [[0 for i in range(m)] for j in range(m)]
    for i in range(n):
        snums = map(int, sequence[i].split())
        for j in range(n):
            dist[i][j] = dist[j][i] = snums[j]
    return dist
```

Now can test ...
Exercise

- Copy the program `upgma0.py`.
- Make up test data for the program.
- Run the code on the test data.
  - NB: there are 2 deliberate bugs in the code
Computing real distances

Several ways to compute biological distance. We shall use the $d^2$ measure.

- Based upon word frequencies. For all combinations of words of length $k$, count the words in both sequences. Sum the square of the differences.
Computing real distances

Several ways to compute biological distance. We shall use the $d^2$ measure.

- Based upon word frequencies.
  For all combinations of words of length $k$, count the words in both sequences. Sum the square of the differences.
  
- $d^2(x, y) = \sum_{|w|=6} (d_w(x) - d_w(y))^2$
Computing real distances

Several ways to compute biological distance. We shall use the $d^2$ measure.

▸ Based upon word frequencies.
   For all combinations of words of length $k$, count the words in both sequences. Sum the square of the differences.

▸ $d^2(x, y) = \sum_{|w|=6} (d_w(x) - d_w(y))^2$
   (Actually, a little more complicated)
Computing real distances

Several ways to compute biological distance. We shall use the $d^2$ measure.

- Based upon word frequencies. For all combinations of words of length $k$, count the words in both sequences. Sum the square of the differences.
  \[ d^2(x, y) = \sum_{|w|=6} (d_w(x) - d_w(y))^2 \]
  (Actually, a little more complicated)

We shall not implement $d^2$ ourselves, but rather use a program that already does the computation.
**wcd**: open source program that clusters sequences.

- Can be used to give $d^2$ score with the right options.
- Correct form: `wcd args -p filename i j`

Example:
```
wcd -l 500 -p hiv-db.out 10 30
```
Computes the $d^2$ score between sequences 10 and 30 of the file `hiv-db.out`

- Compares each region of length 500 of sequence 10, and each region of length 500 of sequence 30 – compute $d^2$ score.
- Choose smallest score of all pairs.
- Result: two numbers separated by space printed to standard output
  - First number is the $d^2$ score of the two sequences.
  - Second number is the $d^2$ score of the first sequence and the RC of the second.
**wcd**: open source program that clusters sequences.

- Can be used to give $d^2$ score with the right options.
- Correct form: `wcd args -p filename i j`
  
  e.g. `wcd -l 500 -p hiv-db.out 10 30`

  Computes the $d^2$ score between sequences 10 and 30 of the file *hiv-db.out*
**wcd**: open source program that clusters sequences.

- Can be used to give $d^2$ score with the right options.
- Correct form: `wcd args -p filename i j`
  e.g. `wcd -l 500 -p hiv-db.out 10 30`
  Computes the $d^2$ score between sequences 10 and 30 of the file `hiv-db.out`
  - Compares each region of length 500 of sequence 10, and each region of length 500 of sequence 30 – compute $d^2$ score.
  - Choose smallest score of all pairs.
**wcd**: open source program that clusters sequences.

- Can be used to give $d^2$ score with the right options.
- Correct form: `wcd args -p filename i j`
  e.g. `wcd -l 500 -p hiv-db.out 10 30`
  Computes the $d^2$ score between sequences 10 and 30 of the file `hiv-db.out`
    - Compares each region of length 500 of sequence 10, and each region of length 500 of sequence 30 – compute $d^2$ score.
    - Choose smallest score of all pairs.

- Result: two numbers separated by space printed to standard output
  First number is the $d^2$ score of the two sequences.
  Second number is the $d^2$ score of the first sequence and the RC of the second.
def compute_distance(sequence):
    n = len(sequence); m = 2*n - 1
    dist = [[0 for i in range(m)] for j in range(m)]
    fname = "\tmp\seqs" + environ["USER"]
    instruction = "wcd -l 1000 -p " + fname + " 0"
    for i in range(n - 1):
        for j in range(i+1, n):
            seqf = open(fname, "w")
            seqf.write("> \n" + sequence[i] + "> \n" + sequence[j] + "\n")
            seqf.close()
            wfile = popen(instruction)
            ans = wfile.readline(); res = ans.split()
            dist[i][j] = dist[j][i] = int(res[0])
            wfile.close()

    return dist
What are the performance characteristics?
What are the alternatives?
What are the performance characteristics?
What are the alternatives?
Why do we use the environment variable?
What happens if the output file exists already?
  if we have write permissions?
  if we don’t?
Drawing the tree . . .

(Actually about how to do tree traversals)
Drawing the tree . . .

(Actually about how to do tree traversals)
Drawing trees is a little tricky . . . so let’s use an external package that does drawing:

- convert our tree representation into *Newick* format
  (used by a number of phylogenetic packages)
Drawing the tree . . .

(Actually about how to do tree traversals)
Drawing trees is a little tricky . . . so let's use an external package that does drawing:

- convert our tree representation into Newick format
  (used by a number of phylogenetic packages)
- save the representation to an external file
Drawing the tree . . .

(Actually about how to do tree traversals)
Drawing trees is a little tricky . . . so let's use an external package that does drawing:

- convert our tree representation into *Newick* format (used by a number of phylogenetic packages)
- save the representation to an external file
- run an external tree drawing program — in our case a program called *tv*. 
Drawing the tree . . .

(Actually about how to do tree traversals)
Drawing trees is a little tricky . . . so let's use an external package that does drawing:

- convert our tree representation into Newick format (used by a number of phylogenetic packages)
- save the representation to an external file
- run an external tree drawing program — in our case a program called tv.

The conversion is an example of tree traversal — process the elements in the tree.
Recursion as a programming technique

Traversal can be done
- in-order
- pre-order
- post-order
Recursion as a programming technique

Traversal can be done

- in-order
- pre-order
- post-order

General format of in-order traversal

```python
def inorderTraversal(current)
inorderTraversal(left)
process(current)
inorderTraversal(right)
```
Example — find the leaf-descendants of a node

def leaf_descendants(node):
    if node is leaf:
        return [ID of node]
    else:
        leftdesc = leaf_descendants(left child of node)
        rightdesc = leaf_descendants(right child of node)
        return leftdesc + rightdesc

Typical example of recursive algorithm:
  ▶ but not completely circular: must be a base case!
def leaf_descendants(tree, curr):
    (left, right) = tree[curr]
    if left == -1 and right == -1):
        return [curr]
    else:
        return leaf_descendants(tree, left) +
        leaf_descendants(tree, right)
The Newick format

Present the tree as a hierarchically nested list:

- (0,1)
- ((0,1),(2,3))
- (0, (1, (2, 3)))
- ((0,1), (2, (((3,4),(5,6)),(7,8)),9))

Also put *branch lengths*:

- (0:10, 1:10)
- (0:10, (1:7,2:7):3)
- (0:10, (1:7,2:7):3)
Algorithm to convert child list into Newick list

Branch length:

- \( \text{height}[x] \):
  - for a leaf: 0
  - for an internal node with children \( u, v \),

\[
\frac{\text{dist}(u, v)}{2}
\]

- Change our \text{make\_tree} so that we compute \text{height}

Conversion: pre-order traversal. At each step

- Convert our left sub-tree to Newick format
- Print distance to left sub-tree
- Convert our right sub-tree to Newick format
- Print distance to right sub-tree
Doing the conversion:

```python
def convert(curr, children, height):
    (left, right) = children[curr]
    if left == -1:
        ans = str(curr)
    else:
        left_tree = convert(left, children, height)
        right_tree = convert(right, children, height)
        left_blen = height[curr] - height[left]
        right_blen = height[curr] - height[right]
        ans = "\n        "("+left_tree+":"+str(left_blen)+",
        ","+right_tree+":"+str(right_blen)+")"
    return ans
```
The Whole Caboodle

```python
sequences = read_sequences(sys.argv[1])
dist = compute_distance(sequences)
(children, height) = make_tree(len(sequences), dist)
tree_n = convert(children.length-1, children, height)
fname = "/tmp/"+environ["USER"]+"tre"
outfile = open(fname,"w")
outfile.write(tree_n+";")
outfile.close();
system("tv "+fname)
```
Object orientation

OO Design philosophy:

► Model the ‘world’ as a set of objects
Object orientation

OO Design philosophy:
- Model the ‘world’ as a set of objects
- More precisely as classes of objects
Object orientation

OO Design philosophy:

- Model the ‘world’ as a set of objects
- More precisely as classes of objects
  - objects are instances of classes
  - classes describe what objects are, what they can do
Object orientation

OO Design philosophy:

▶ Model the ‘world’ as a set of objects
▶ More precisely as classes of objects
  ▶ objects are instances of classes
  ▶ classes describe what objects are, what they can do
▶ Class should define for each object its members
  ▶ what type of data is stored (data members)
  ▶ what actions can be performed on the object (functions members or methods
▶ Classes can be related through class hierarchy
Benefit 1 of OO: Modularity and abstraction

Supports the divide-and-conquer approach.
Benefit 1 of OO: Modularity and abstraction

Supports the divide-and-conquer approach.

- Given a problem, decide what the key classes are.
Benefit 1 of OO: Modularity and abstraction

Supports the divide-and-conquer approach.

- Given a problem, decide what the key classes are.
- For each class, decide its data members and methods.
Benefit 1 of OO: Modularity and abstraction

Supports the divide-and-conquer approach.

▶ Given a problem, decide what the key classes are.
▶ For each class, decide its data members and methods
▶ Principle (not always followed)
  ▶ set of method declarations are the *public interface*

Describe the *what*. All that a user of the class needs to know.
Benefit 1 of OO: Modularity and abstraction

Supports the divide-and-conquer approach.

- Given a problem, decide what the key classes are.
- For each class, decide its data members and methods.
- Principle (not always followed)
  - set of method declarations are the public interface
  Describe the what. All that a user of the class needs to know.
  - The how is the data members and the implementation of the methods.
Benefit 1 of OO: Modularity and abstraction

Supports the divide-and-conquer approach.

- Given a problem, decide what the key classes are.
- For each class, decide its data members and methods
- Principle (not always followed)
  - set of method declarations are the *public interface*
    Describe the *what*. All that a user of the class needs to know.
  - The *how* is the data members and the implementation of the methods

Principle of abstraction.
Benefit 2 of OO: Inheritance

Classes can be organised in class hierarchies (super-class/sub-class)

- Super-class is more abstract – carries information, behaviour common to sub-classes
Benefit 2 of OO: Inheritance

Classes can be organised in class hierarchies (super-class/sub-class)

- Super-class is more abstract – carries information, behaviour common to sub-classes

Example: Animal Class

- Mammal class
  - Primate class
  - Cat class
- Fish class
  - big fish class
  - goldfish class
Sub-classes *inherits* all data/behaviour from its super-classes.

- can add new members/behaviour
Sub-classes *inherits* all data/behaviour from its super-classes.

- can add new members/behaviour
- can redefine members too

if you access a member of an object, python starts searching the class hierarchy from the bottom
Sub-classes *inherits* all data/behaviour from its super-classes.

- can add new members/behaviour
- can redefine members too
  if you access a member of an object, python starts searches the class hierarchy from the bottom

**Benefits of inheritance**

- code re-use
- promotes abstraction
- mainly seen when have large programs, or code re-use
Scope and Existence

General issue in Python (and any programming language):

- **Scope**
  Where an identifier can be seen.
  Usually syntactic issue — determined at compile time.
  Should always be the same
  — not in python.
Scope and Existence

General issue in Python (and any programming language):

▶ Scope
Where an identifier can be seen.
Usually syntactic issue — determined at compile time.
Should always be the same
– not in python.

▶ Existence
When a variable/object exists.
Determined at run-time – dynamic, often different for every run
Scope & name spaces...

Very simplified rule:

- Variable can be accessed if it’s in a current namespace.
- Name space is a collection of variables.
- Name spaces can be determined statically
  Nested: e.g. functions inside functions
- Scope: textual region where a namespace exists.
- The contents of a namespace is determined dynamically.
  Not common in programming languages.

Also be aware of the Python reference model: variables for complex types do not contain the variable itself, but contain references to the variable.
d = 10
x = 20

def fun1(a, b):
    a = a+x
    c = 20; m = 30

def fun2(x,a):
    c = 30
    x = 10

x = x + 30
d = 2*x+d
m = 100; n = 200
d = 10
x = 20

def fun1(a, b):
    a = a+x

def fun2(b):
    s = a+3; t = s+b
    return t
c = 20+fun2(a+10)
m = 30

def fun2(x,a):
    c = 30
    x = 10

x = x + 30; d = 2*x+d
Object-oriented language supports object-oriented design.
Object-oriented language supports object-oriented design. In Python:

- use class name like a function to *instantiate* or create an object
  
  \[d = \text{Animal}()\]

  New animal object is created. The variable name is \(d\)
Object-oriented language supports object-oriented design. In Python:

- use class name like a function to *instantiate* or create an object
  
  d = Animal()
  
  New animal object is created. The variable name is `d`

- use dot notation to refer to members
  
  d.setName("Samson")
  
  d.height = 200
  
  x=d.getAge()
  
  x=changeMass(delta)
Principle of information hiding

Good idea not to access data members directly – access through methods.
Class implementation

In the class:

- methods are declared — first parameter of each method is the actual object
Class implementation

In the class:

- methods are declared — first parameter of each method is the actual object
- there can be other statements
  - create data items (but not necessary in Python)
  - initialising code
  - other execution (but rare)
class Sequence:
    def setSeq(self, newseq):
        self.seq = newseq

    def setID(self, id):
        self.id = id

    def showSeq(self):
        print "">"+self.id+"\n"+self.seq

    def getID(self):
        return self.id
Class is a generic description for all objects

- NB: parameter self refers to actual object being acted upon

- The use of the word self is convention

- NB: Different to C++, Java where this is passed implicitly. In Python, self must be explicitly include.
How used:

```python
s1 = Sequence()
s1.setSeq("ACCATTG")
s1.setID("A766671")
s1.showSeq()
```

Can think of the dot-notation as a short-hand. The following two are equivalent:

- `s1.showSeq()`
- `Sequence.showSeq(s1)`
Initialiser method: `__init__` is a special method that will be called automatically when an object is instantiated.

- can give it parameters

```python
def __init__(self, id, newseq):
    self.data = newseq
    self.id = id
```

Called thus:

```python
s1 = Sequence("A23321", "CCTTGGTTA")
```
Can have some data members which belong to the *class* rather than the *object*

- static data members
Can have some data members which belong to the *class* rather than the *object*

- static data members

Example: suppose we want to keep track of how many sequences we have created.

```python
class Sequence:
    numseqs = 0

    def __init__(self, id, newseq):
        self.data = newseq
        self.id = id
        self.serialnum = Sequence.numseqs
        Sequence.numseqs = Sequence.numseqs + 1
```

Referred to through the class name
Declaring sub-classes

Indicate which sub-classes are inherited from when declared.

class NucleotideSeq(Sequence):

    def count_base(self, base):
        ....

    def setID(self, id):
        self.id = "N"+id

Objects of class NucleotideSeq inherit from Sequence. All members are inherited, but –

- new method: count_base
- redefined method setID
Python has dynamic dispatch: assume we have a ProteinSeq class in which setID is defined:

```python
s1 = Sequence("1","GGJHKPOHHH")
s2 = ProteinSeq("2","RARRTTT")
s3 = NucleotideSeq("3","ACTG")
seqs = [s1,s2,s3]
```

```python
for s in seqs:
    s.setID(s.getID()+"db_NBN")
```

- samegetID is called each time – defined in Sequence
- different setID is called each time, depending on class
Uses of classes for non OO purposes

Classes can be used, esp in simple programs for aggregating data

▶ cf. records/structs in Pascal, C

If a number of variables logically belong to each other, or are associated with each other, can bundle them up into one object:

▶ conceptually cleaner
▶ shows relationship of components
▶ Pragmatically easier (e.g. in passing of parameters)

Example:

▶ list of students, each with name, number, mark, etc.
▶ have one array, rather than 5
▶ each item is an object that records the information
class TreeNode:
    pass

for i in range(n):
    newelt = TreeNode()
    newelt.children = (-1, -1)
    newelt.height = 0
    newelt.processed = False
    tree.append(newelt)

- pass is just a ‘do-nothing’
- usually at least have an initialiser
But why not use some OO:

class TreeNode:
    def __init__(self, left, right):
        self.children = (left, right)
        self.height = 0
        self.processed = False

    def setHeight(self, height):
        self.height = height

for i in range(n):
    newelt = TreeNode(-1, -1)
    tree.append(newelt)
    newelt = TreeNode(0, 1)
    tree.append(newelt)
Exceptions

An exception is an event or operation that disrupts the normal flow of the program.

- Commonly associated with something going wrong

Examples:

- Dividing by zero: \( x=4/0 \)
  Gives a ZeroDivisionError

- Opening a file for reading that doesn’t exist
  \( f=open("utopia") \)
  IOError

Exceptions raised when detected (hardware or software).

- it must be caught, or the program terminates
Example use 1:

- Section of code which does lots of I/O.
- If problem, want to detect, print error message, terminate
- To check after each line, makes the code klunky, hides logic;
- Protect the whole section with try
try:
    op1
    op2
    op3
except IOError as err:
    print("I/O error: %s\n", err)
Example use 2:

- Method A calls method B calls method C calls method D.
- C detects a problem: What do do?
Example use 2:

- Method A calls method B calls method C calls method D.
- C detects a problem: What do do?
- Exit program? may be too extreme
Example use 2:

- Method A calls method B calls method C calls method D.
- C detects a problem: What do do?
- Exit program? may be too extreme
- Write error handling code in D, C, B, A – lot’s of work
Example use 2:
- Method A calls method B calls method C calls method D.
- C detects a problem: What do do?
- Exit program? may be too extreme
- Write error handling code in D, C, B, A – lot’s of work
- Use exceptions
def read_sequence(): ....

def read_file(): ....

def read_directory(): ...

def process_data():
    for dir in dirs:
        try:
            read_directory(dir)
        except IOError as err:
            print("I/O error: %s\n", err)
Creating your own exceptions

class IllegalData(Exception):
    def __init__(self, val):
        self.val = val

try:
    do_ops
    if (problem): raise IllegalData(v)
    ....
except IllegalData, val:
    ....
Case study: DNA layout

Small case study to show

▶ use of OO
▶ use of exceptions
▶ use of new data structures

Problem: Given, a set of short DNA sequences that come from a long sequence:

▶ find the layout of the sequences
Given the fragments with an overlap of 3

A00  cacgcatgt
A01  cccgacggg
A02  tgtttggtg
A03  aaatcgccc
A04  gtgacacccc
A05  ccctatccac
Given the fragments with an overlap of 3

A00  cacgcaggt
A01  cccgacggg
A02  tgtttgggtg
A03  aaatcgccc
A04  gtgacacccc
A05  ccctatccac

The originating sequence is

\[
\text{aaatcgccc tatccac gcatgt ttgggtg acaccc gacggg}
\]
Real world much more complex

- longer overlapping areas
- approximate match, not exact match, varying length
- errors!
- but simple problem gives a flavour
Real world much more complex

- longer overlapping areas
- approximate match, not exact match, varying length
- errors!
- but simple problem gives a flavour

For an overlap of 3, there are 64 possible overlaps:

- call them connections
Some background — graph theory

Graphs are very useful structures — good abstractions for many real world problems:

- lots of bioinformatic problems cast as graph problems
Some background — graph theory

Graphs are very useful structures — good abstractions for many real world problems:

- lots of bioinformatic problems cast as graph problems

Definitions:

- A graph is a set of vertices (nodes) and edges
- Graphically represented by blobs and lines
Some background — graph theory

Graphs are very useful structures — good abstractions for many real world problems:

▶ lots of bioinformatic problems cast as graph problems

Definitions:

▶ A graph is a set of vertices (nodes) and edges
▶ Graphically represented by blobs and lines
▶ Edges can be directed or undirected. Use lines / lines with arrows to represent
Some background — graph theory

Graphs are very useful structures — good abstractions for many real world problems:

▶ lots of bioinformatic problems cast as graph problems

Definitions:

▶ A graph is a set of vertices (nodes) and edges
▶ Graphically represented by blobs and lines
▶ Edges can be directed or undirected. Use lines / lines with arrows to represent
▶ degree of a vertex number of edges going from a vertex
▶ in-degree of a vertex is the number of edges coming into a vertex
Path between $x$ and $y$ is a sequence of adjacent edges going from $x$ to $y$
Path between \( x \) and \( y \) is a sequence of adjacent edges going from \( x \) to \( y \)

Computer representation of graph:
Path between $x$ and $y$ is a sequence of adjacent edges going from $x$ to $y$

Computer representation of graph:

- list of vertices (number them from 0 . . . )
- edges – many ways – we use an adjacency list:
  - for each vertex, have a list of the edges that leave the vertex
  - maybe other auxiliary information

Graph is a complex object — ideal for an OO representation.
DNA layout as a graph problem

Build a graph as follows:

- The *connections* are vertices.
  - e.g. AAA, AAC, AAT, AAG, ACA, ...
DNA layout as a graph problem

Build a graph as follows:

- **The connections are vertices.**
  - e.g. AAA, AAC, AAT, AAG, ACA, ...
- For each DNA fragment, create a directed edge
  - starts at the left-hand connection
  - finishes at the right-hand connection

  e.g. for ATATTGTACCCTAGAAGCG: edge (ATA, GCG)
DNA layout as a graph problem

Build a graph as follows:

- The *connections* are vertices.
  - e.g. AAA, AAC, AAT, AAG, ACA, ...  
- For each DNA fragment, create a directed edge
  - starts at the left-hand connection
  - finishes at the right-hand connection

  e.g. for ATATTGTACCTAGAAGCG: edge (ATA, GCG)
DNA layout as a graph problem
Case Study: DNA Layout

DNA layout as a graph problem
Introduction to Programming using the Python Programming Language

Case Study: DNA Layout

DNA layout as a graph problem
Introduction to Programming using the Python Programming Language

Case Study: DNA Layout

DNA layout as a graph problem
Introduction to Programming using the Python Programming Language

Case Study: DNA Layout

DNA layout as a graph problem

AAC
GTA

GGG

GTG

AAA

GAG

CCC

GCG

CAC

GTT

GTA

GGG

AAATCGCCC
TGTGTTTGGTG
CCCGACGGG
CCCTATCAC

GAT
DNA layout as a graph problem

AAATCGCC
TGTTTGGGTG
CCCGACGGG
CCCTATCAC
GTGACACCC

GAG
GCG
GGG
GTG
AAA
CCC
CAC
TGT
GTA
GTG
GTT
GAT
Solution to the problem is an Euler path:

- Path in the graph that goes through each edge *exactly* once.
- Exists if
  - all vertices have same in-degree as out-degree; in which case any vertex could be the start; or
Solution to the problem is an Euler path:

- Path in the graph that goes through each edge \textit{exactly} once.
- Exists if
  - all vertices have same in-degree as out-degree; in which case any vertex could be the start; or
  - if one vertex has in-degree 1 greater than out-degree and one vertex has out-degree 1 greater than in-degree
  Start is vertex with a vertex with out-degree 1 greater in-degree
Euler algorithm:

▶ If you follow an edge mark it.
Euler algorithm:

- If you follow an edge mark it.
- Pick start vertex.
Euler algorithm:

- If you follow an edge mark it.
- Pick start vertex.
- Repeat: follow any unmarked edge out and go to the next vertex.
  Mark the edge as processed.
- Stop when you can go no further.
- If the path created contains all the edges in the graph, you can stop.
- Otherwise, repeat until all edges processed
  - there must be a vertex in the path with an unprocessed outgoing edge
  - find a path from that point
  - insert into the path found already
Python solution

- Create a sequence class
- Create a graph class (general) — this can be used in future applications.
- Create a SeqGraph class for representing sequence graphs
- Put everything together.
class FileFinished(Exception):
    pass

class Sequence:
    def __init__(self, accid, seq, quality):
        self.accid = accid
        self.data = seq
        self.quality = quality
    def getAccID(self):
        return self.accid
    def getSeq(self):
        return self.data
class Graph:

    def __init__(self, n, directed=False):
        self.numedges = 0
        self.numv = n
        self.adj = {}
        self.indeg = {}
        self.directed = directed

    def vertexOf(self, edge):
        # return the dest vertex of the edge
        # default is we just have the vertex
        return edge
def numEdgesOf(self):
    return self.numEdges

def addEdge(self, i, edge):
    j = self.vertexOf(edge)
    if not self.adj.has_key(i): self.adj[i] = []
    if not self.indeg.has_key(j): self.indeg[j] = 0
    self.adj[i].append(edge)
    self.indeg[j] = self.indeg[j] + 1
    self.numEdges = self.numEdges + 1

def isEdge(self, i, j):
    return self.degree(i) > 0 and (j in map(vertexOf, self.adj[i]))
def degree(self, i):
    if not self.adj.has_key(i):
        return 0
    else:
        return len(self.adj[i])

def indegreeOf(self, i):
    if not self.indeg.has_key(i):
        return 0
    else:
        return self.indeg[i]

def vertexList(self):
    return self.adj.keys()
Now we derive the SeqGraph sub-class

class SeqGraph(Graph):

    def __init__(self,n):
        Graph.__init__(self,n,True)
        self.vertexInfo = []

    def addEdge(self,i,j,seq):
        Graph.addEdge(self,i,(j,seq))

    def vertexOf(self,edge):
        (j,info) = edge
        return j
def seq0f(self,edge):
    (j,info) = edge
    return info

def findStart(self):
    for v in self.vertexList():
        if self.indegree0f(v) < self.degree(v):
            return v
    return 0
def subPath(self, start):
    path = []
    curr = start
    # provided there’s somewhere to go
    while (self.degree(curr) > 0):
        # go to first elt in adj list
        (next, edge) = self.adj[curr][0]
        # delete from adj list
        del self.adj[curr][0]
        path.append(edge)
        curr = next
    return path
def eulerPath(self):
    start = self.findStart()
    path = self.subPath(start)
    while len(path) < self.numEdgesOf():
        i = 0
        for edge in path:
            v = edge.getSeq()[0:3]
            if self.degree(v) > 0:
                newpath = self.subPath(v)
                path = path[:i]+newpath+path[i:]
                break
            i=i+1
        return map(Sequence.getAccID,path)
Main program

sgraph = SeqGraph(64)
# Now read in the graph vertices
inpfile = open(argv[1])
try:
    while (True):
        s = readSeq(inpfile)
        the_seq = s.getSeq()
        first = the_seq[0:3]
        last = the_seq[-3:]
        sgraph.addEdge(first, last, s)
except FileFinished:
    pass