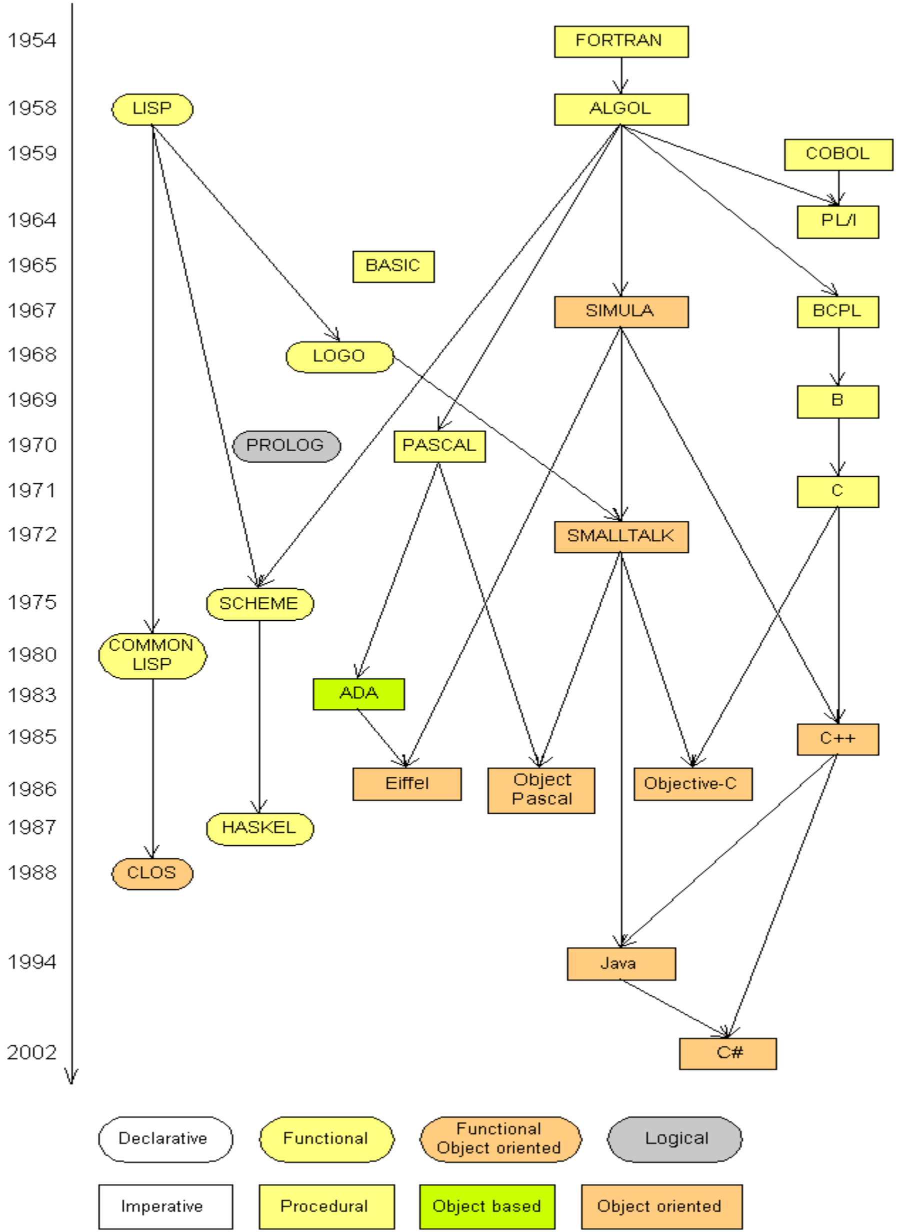
# Introduction to Computer Forensics and Security

# Introduction to Python

# Programming Languages …

- **Some influential ones:**
  - FORTRAN
    - science / engineering
  - COBOL
    - business data
  - LISP
    - logic and AI
  - BASIC
    - a simple language



# Programming Basics

- **code** or **source code**: The sequence of instructions in a program.

- **syntax**: The set of legal structures and commands that can be used in a particular programming language.

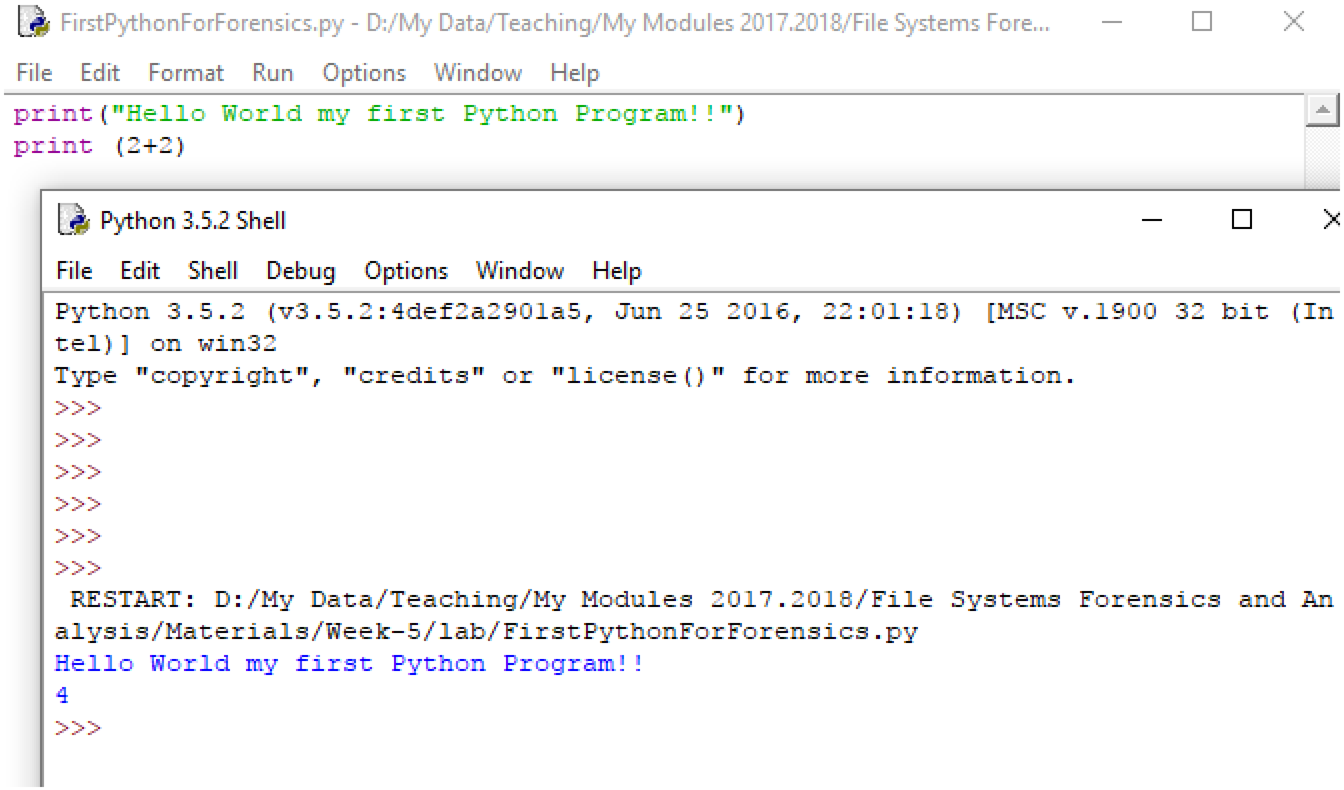
- **output**: The messages printed to the user by a program.

- **console**: The text box onto which output is printed.
  - Some source code editors pop up the console as an external window, and others contain their own console window.

# Python features

| | |
|---|---|
| no compiling or linking | rapid development cycle |
| no type declarations | simpler, shorter, more flexible |
| automatic memory management | garbage collection |
| high-level data types and operations | fast development |
| object-oriented programming | code structuring and reuse, C++ |
| embedding and extending in C | mixed language systems |
| classes, modules, exceptions | "programming-in-the-large" support |

# My First Python Program!


FirstPythonForForensics.py - D:/My Data/Teaching/My Modules 2017.2018/File Systems Fore...
File Edit Format Run Options Window Help

```python
print("Hello World my first Python Program!!")
print (2+2)
```

Python 3.5.2 Shell
File Edit Shell Debug Options Window Help

```
Python 3.5.2 (v3.5.2:4def2a2901a5, Jun 25 2016, 22:01:18) [MSC v.1900 32 bit (In
tel)] on win32
Type "copyright", "credits" or "license()" for more information.
>>>
>>>
>>>
>>>
>>>
>>>
 RESTART: D:/My Data/Teaching/My Modules 2017.2018/File Systems Forensics and An
alysis/Materials/Week-5/lab/FirstPythonForForensics.py
Hello World my first Python Program!!
4
>>>
```

# Whitespace

- Whitespace is meaningful in Python: especially indentation and placement of newlines.
  - Use a newline to end a line of code.
    (Not a semicolon like in C++ or Java.)
    (Use \ when must go to next line prematurely.)
  - No braces { } to mark blocks of code in Python… Use consistent indentation instead. The first line with a new indentation is considered outside of the block.
  - Often a colon appears at the start of a new block.
    (We'll see this later for function and class definitions.)

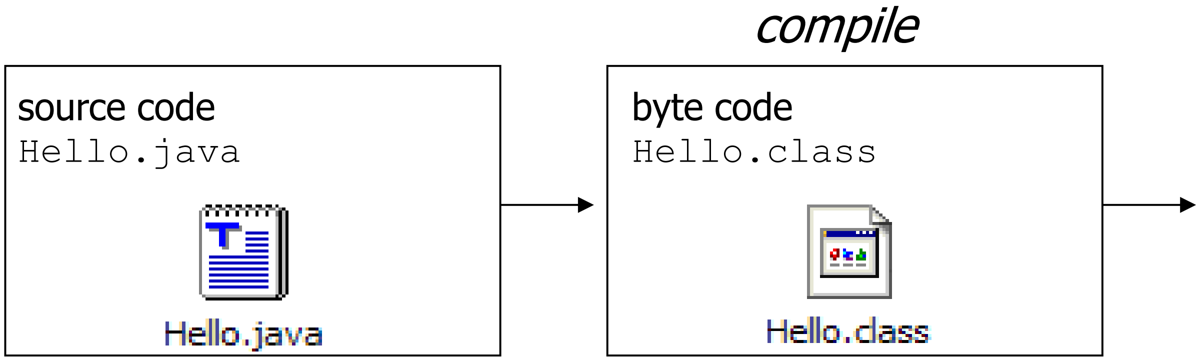
# Comments

- Start comments with # the rest of line is ignored.
- Can include a “documentation string” as the first line of any new function or class that you define.
- The development environment, debugger, and other tools use it: it’s good style to include one.

```
def my_function(x, y):
  “““This is the docstring. This
  function does blah blah blah.”””
  # The code would go here...
```

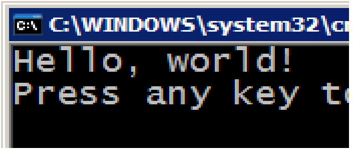
# Compiling and interpreting

- Many languages require you to *compile* (translate) your program into a form that the machine understands.



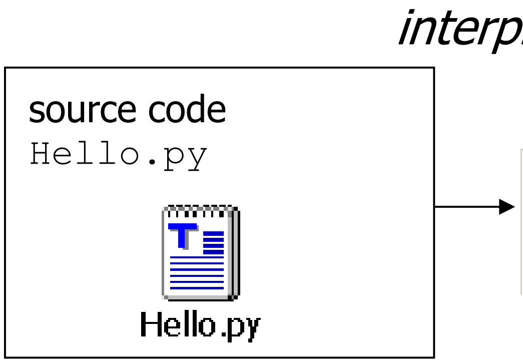
*execute*

output

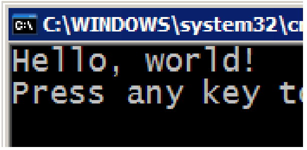


```
Hello, world!
Press any key t
```

- Python is instead directly *interpreted* into machine instructions.



output



```
Hello, world!
Press any key t
```

# Expressions

- **expression**: A data value or set of operations to compute a value.

- Arithmetic operators we will use:

| | |
|---|---|
| `+ - * /` | addition, subtraction/negation, multiplication, division |
| `%` | modulus, a.k.a. remainder |
| `**` | exponentiation |

- **precedence**: Order in which operations are computed.
  - `* / % **` have a higher precedence than `+ -`

    `1 + 3 * 4` is `13`

  - Parentheses can be used to force a certain order of evaluation.

    `(1 + 3) * 4` is `16`

# Real numbers

- Python can also manipulate real numbers.
  - Examples: `6.022` `-15.9997` `42.0`

- The operators `+ - * / % ** ( )` all work for real numbers.
  - The `/` produces an exact answer: `15.0 / 2.0` is **`7.5`**
  - The same rules of precedence also apply to real numbers: Evaluate `( )` before `* / %` before `+ -`

- When integers and reals are mixed, the result is a real number.
  - Example: `1 / 2.0` is `0.5`

# Variables

- **variable**: A named piece of memory that can store a value.
  - Usage:
    - Compute an expression's result,
    - store that result into a variable,
    - and use that variable later in the program.

- **assignment statement**: Stores a value into a variable.
  - Syntax:

    ***name*** = ***value***

  - Examples:

    ```
    x = 5
    gpa = 3.14
    ```

  - A variable that has been given a value can be used in expressions.

    `x + 4` is `9`

# print

- `print` (“”) Produces text output on the console.
- Syntax:

    `print("`***Message***`“)`

    `print(`***Expression)***

    - Prints the given text message or expression value on the console, and moves the cursor down to the next line.

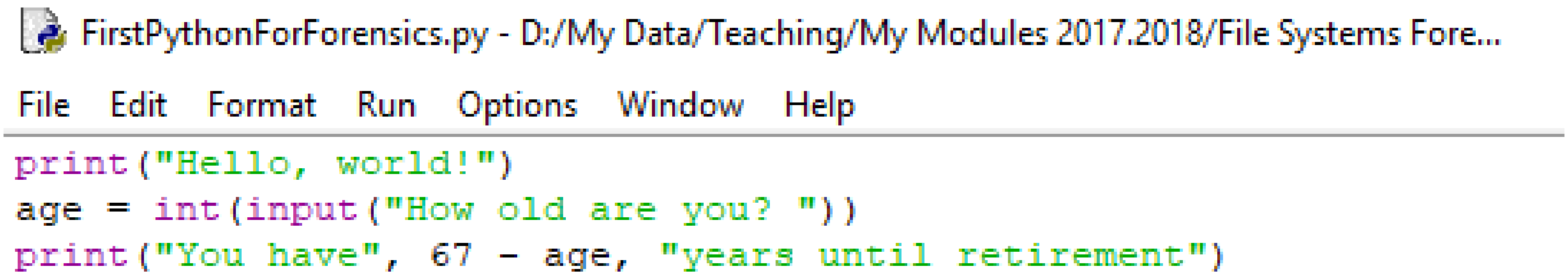
- Examples:

```
print("Hello, world!“)
age = 45
print("You have", 65 - age, "years until retirement“)
```

Output:

```
Hello, world!
You have 20 years until retirement
```

# Reading from the user


FirstPythonForForensics.py - D:/My Data/Teaching/My Modules 2017.2018/File Systems Fore...
File
Edit
Format
Run
Options
Window
Help

```
print("Hello, world!")
age = int(input("How old are you? "))
print("You have", 67 - age, "years until retirement")
```

# The `for` loop

- **`for` loop**: Repeats a set of statements over a group of values.
  - Syntax:

    `for` ***variableName*** `in` ***groupOfValues*****:**

        ***statements***

    - We indent the statements to be repeated with tabs or spaces.
    - ***variableName*** gives a name to each value, so you can refer to it in the ***statements***.
    - ***groupOfValues*** can be a range of integers, specified with the `range` function.

  - Example:

```
for x in range(1, 6):
    print x, "squared is", x * x
```

Output:

```
1 squared is 1
2 squared is 4
3 squared is 9
4 squared is 16
5 squared is 25
```

# `range`

- The `range` function specifies a range of integers:
  - `range(`***start***`,` ***stop***`)` - the integers between ***start*** (inclusive) and ***stop*** (exclusive)
- It can also accept a third value specifying the change between values.
  - `range(`***start***`,` ***stop***`,` ***step***`)` - the integers between ***start*** (inclusive) and ***stop*** (exclusive) by ***step***
- Example:

```
for x in range(5, 0, -1):
    print x
```

# range

- The `range` function specifies a range of integers:
  - `range(`***start***`,` ***stop***`)` - the integers between ***start*** (inclusive) and ***stop*** (exclusive)

- It can also accept a third value specifying the change between values.
  - `range(`***start***`,` ***stop***`,` ***step***`)` - the integers between ***start*** (inclusive) and ***stop*** (exclusive) by ***step***

- Example:

```
for x in range(5, 0, -1):
    print(x)
print "Blastoff!"
```

Output:

```
5
4
3
2
1
Blastoff!
```

# Cumulative loops

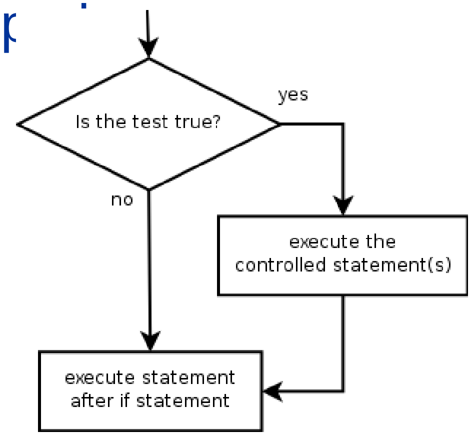
- Some loops incrementally compute a value that is initialized outside the loop. This is sometimes called a *cumulative sum*.

```
sum = 0
for i in range(1, 11):
    sum = sum + (i * i)
print("sum of first 10 squares is", sum)

Output:
sum of first 10 squares is 385
```

# if

- **if statement**: Executes a group of statements only if a certain condition is true. Otherwise, the statements are skipp


Is the test true?
yes
no
execute the controlled statement(s)
execute statement after if statement

  - Syntax:

```
if condition:
    statements
```

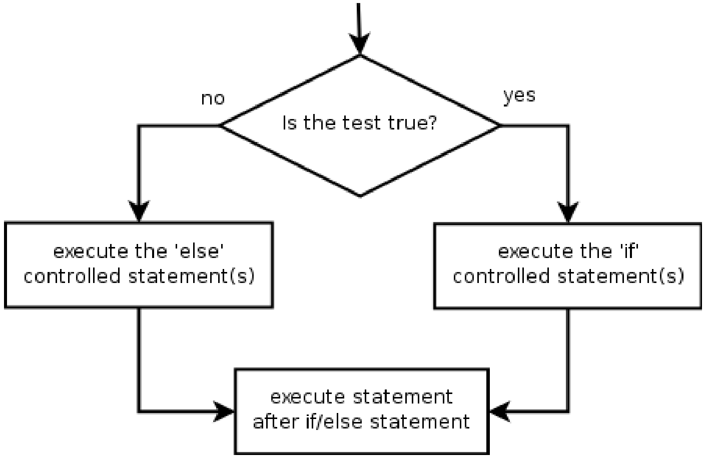
- Example:

```
gpa = 3.4
if gpa > 2.0:
    print("Your application is accepted.”)
```

# if/else

- **if/else statement**: Executes one block of statements if a certain condition is True, and a second block of statements if it is False.
  - Syntax:

```
if condition:
        statements
else:
        statements
```



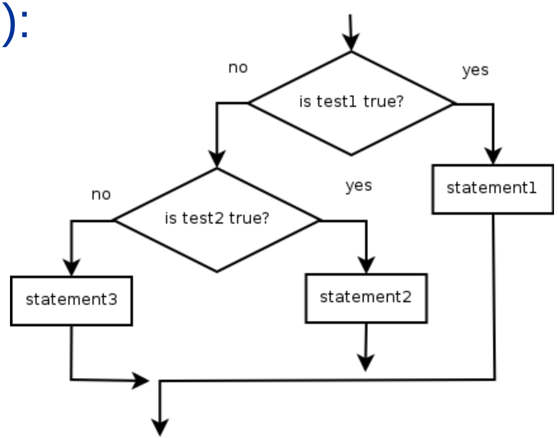
Is the test true?
no
yes
execute the 'else' controlled statement(s)
execute the 'if' controlled statement(s)
execute statement after if/else statement

- Example:

```
gpa = 1.4
if gpa > 2.0:
    print "Welcome to Mars University!"
else:
    print "Your application is denied."
```

- Multiple conditions can be chained with `elif` ("else if"):

```
if condition:
        statements
elif condition:
        statements
else:
        statements
```

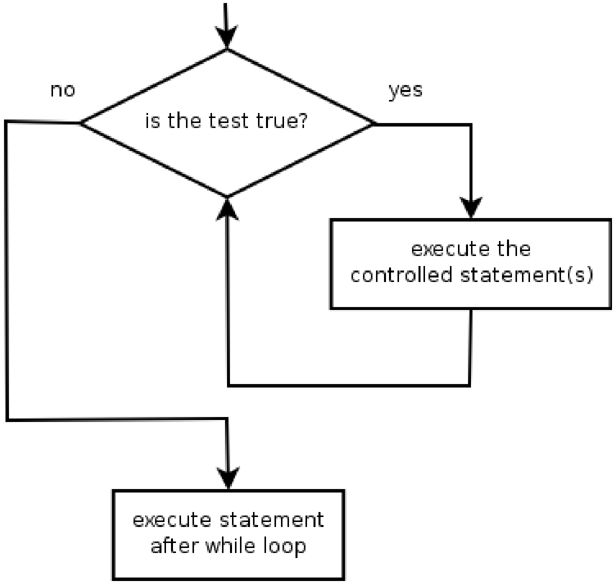


# while

- **`while` loop**: Executes a group of statements as long as a condition is True.
  - good for *indefinite loops* (repeat an unknown number of times)

- Syntax:

```
while condition:
      statements
```


no
is the test true?
yes
execute the controlled statement(s)
execute statement after while loop

- Example:

```
number = 1
while number < 200:
    print(number)
    number = number * 2
```

  - Output:

```
1 2 4 8 16 32 64 128
```

# Logic

- Many logical expressions use *relational operators*:

| Operator | Meaning | Example | Result |
|---|---|---|---|
| == | equals | 1 + 1 == 2 | True |
| != | does not equal | 3.2 != 2.5 | True |
| < | less than | 10 < 5 | False |
| > | greater than | 10 > 5 | True |
| <= | less than or equal to | 126 <= 100 | False |
| >= | greater than or equal to | 5.0 >= 5.0 | True |

Logical expressions can be combined with *logical operators*:

| Operator | Example | Result |
|---|---|---|
| and | 9 != 6 and 2 < 3 | True |
| or | 2 == 3 or -1 < 5 | True |
| not | not 7 > 0 | False |

# Strings

- **string**: A sequence of text characters in a program.
  - Strings start and end with quotation mark `"` or apostrophe `'` characters.
  - Examples:

```
"hello"
"This is a string"
"This, too, is a string.   It can be very long!"
```

- A string may not span across multiple lines or contain a " character.

```
"This is not
a legal String."
```

```
"This is not a "legal" String either."
```

- A string can represent characters by preceding them with a backslash.
  - `\t` tab character
  - `\n` new line character
  - `\"` quotation mark character
  - `\\` backslash character
  - Example: `"Hello\tthere\nHow are you?"`

# String Indexes

- Characters in a string are numbered with *indexes* starting at 0:
  - Example:

```
name = “Steve !?"
```

| index | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|---|
| character | S | t | e | v | e | | ! | ? |

- Accessing an individual character of a string:

  ***variableName*** [ ***index*** ]

  - Example:

```
print(name, "starts with", name[0])
```

Output:

```
Steve starts with S
```

# String properties

- `len(`***string***`)` - number of characters in a string (including spaces)
- `str.lower(`***string***`)` - lowercase version of a string
- `str.upper(`***string***`)` - uppercase version of a string

- Example:

```
name = “Steve Jobs"
length = len(name)
big_name = str.upper(name)
print(big_name, "has", length, "characters“)
```

Output:

```
STEVE JOBS has 10 characters
```

# Lists

- lists can be heterogeneous
  - `a = ['spam', 'eggs', 100, 1234, 2*2]`
- Lists can be indexed and sliced:
  - `a[0]` → spam
  - `a[:2]` → ['spam', 'eggs']
- Lists can be manipulated
  - `a[2] = a[2] + 23`
  - `a[0:2] = [1,12]`
  - `a[0:0] = []`
  - `len(a)` → 5

# List methods

- `append(`*x*`)`
- `insert(`*i*`,`*x*`)`
- `remove(`*x*`)`
- `index(`*x*`)`
  - return the index for value *x*
- `count(x)`
  - how many times x appears in list
- `sort()`
  - sort items in place
- `reverse()`
  - reverse list

# del – removing list items

- remove by index, not value
- remove slices from list (rather than by assigning an empty list)

```
>>> a = [-1,1,66.6,333,333,1234.5]
>>> del a[0]
>>> a
[1,66.6,333,333,1234.5]
>>> del a[2:4]
>>> a
[1,66.6,1234.5]
```

# Tuples and sequences

- lists, strings, **tuples**: examples of *sequence* type
- tuple = values separated by commas

```
>>> t = 123, 543, 'bar'
>>> t[0]
123
>>> t
(123, 543, 'bar')
```

# Tuples

- Tuples may be nested

```
>>> u = t, (1,2)
>>> u
((123, 542, 'bar'), (1,2))
```

- kind of like structs, but no element names:
  - (x,y) coordinates
  - database records
- like strings, immutable → can't assign to individual items

# Tuples

- Empty tuples: ()

```
>>> empty = ()
>>> len(empty)
0
```

- one item → trailing comma

```
>>> singleton = 'foo',
```

# Tuples

- sequence unpacking → distribute elements across variables

```
>>> t = 123, 543, 'bar'
>>> x, y, z = t
>>> x
123
```

- packing always creates tuple
- unpacking works for any sequence

# Dictionaries

- indexed by keys
- keys are any immutable type: e.g., tuples
- but not lists (mutable!)
- uses 'key: value' notation

```
>>> tel = {'hgs' : 7042, 'lennox': 7018}
>>> tel['cs'] = 7000
>>> tel
```

# Dictionaries

- no particular order
- delete elements with del

```
>>> del tel['foo']
```

- keys() method → unsorted list of keys

```
>>> tel.keys()
['cs', 'lennox', 'hgs']
```

- use has_key() to check for existence

```
>>> tel.has_key('foo')
0
```

# Defining functions

```
def numlist(n):
  """Print a numbers up to n."""
b = 1
  while b < n:
    print(b)
    b++

>>> numlist(20)
```

- First line is *docstring*
- first look for variables in local, then global
- need global to assign global variables

# Modules

- collection of functions and variables, typically in scripts
- definitions can be imported
- file name is module name + .py
- e.g., create module `fibo.py`

```
def fib(n): # write Fib. series up to n
   ...
def fib2(n): # return Fib. series up to n
```

# Modules

- import module:

```
import fibo
```

- Use modules via "name space":

```
>>> fibo.fib(1000)
>>> fibo.__name__
'fibo'
```

- can give it a local name:

```
>>> fib = fibo.fib
>>> fib(500)
```

# Modules

- function definition + executable statements
- executed only when module is imported
- modules have private symbol tables
- avoids name clash for global variables
- accessible as *module.globalname*
- can import into name space:

  ```
  >>> from fibo import fib, fib2
  >>> fib(500)
  ```

- can import all names defined by module:

  ```
  >>> from fibo import *
  ```

# Python Interfaces

- IDLE – a cross-platform Python development environment
- PythonWin – a Windows only interface to Python
- Python Shell – running 'python' from the Command Line opens this interactive shell
- For the exercises, we'll use IDLE, but you can try them all and pick a favorite

# Questions?

m.owda@mmu.ac.uk