MAS212 Scientific Computing and Simulation

Dr. Sam Dolan

School of Mathematics and Statistics,
University of Sheffield

Autumn 2019

http://sam-dolan.staff.shef.ac.uk/mas212/

G18 Hicks Building
s.dolan@sheffield.ac.uk
These slides are intended as a guide to some basics of the Python 3 language.

Many other tutorials are available on the web.

See also course webpage; and MAS115.

These slides show output from the standard Python interpreter, e.g.

```
>>> 3 + 4
7
```

I recommend that you try entering the code snippets for yourself in (e.g.) an `ipython` terminal within Spyder (lower right panel).

The best way to learn programming is through self-study.
Comments in Python

- **Comments** in Python are preceded by `#` symbol

- Comments are intended for humans, and they are ignored by the interpreter

- Example:

```python
>>> 3 + 4  # Python will ignore this comment
7
```

- Comments can be used to help explain to yourself, or others, what your code is doing . . . or at least what you hope it is doing!
Arithmetic

Add, subtract, multiply and divide:

```python
>>> 2 + 3
5
>>> 2 - 3
-1
>>> 2 * 3
6
>>> 2 / 3
0.6666666666666666
>>> 2 / 3.0
0.6666666666666666
```

NB: In Python 2.7, dividing an int by an int returns an int (after rounding down).

In Python 3.x this is not the case. Use `2 // 3` if you want only the integer part.
Arithmetic

- Raise to a power with **. Find the remainder with %.

```
>>> 5**2
25
>>> 2**0.5
1.4142135623730951
>>> 11 % 3
2
>>> 26 % 7
5
```

- Recall that raising to power of 0.5 is same as taking the square root.
Data Types

- **bool**: a Boolean, either True or False

- **Numerical types**: int, float, complex

- **Container types**: list, tuple, str, set, dict

- **Other types**: type, function, NoneType, ...

- **Specialized types**: [link](https://docs.python.org/3/library/datatypes.html)

- **Find out the data type with type function**:

  ```python
  >>> type(3)
  int
  ```
Data Types: Numerical

Examples:

```python
>>> type(True)
bool

>>> type(3)
int

>>> type(3.0)
float

>>> type(3.0 + 2.0j)
complex
```
Examples of `str`, `list`, `tuple`, `set`, `dict` types:

```python
>>> type("hello")
str
>>> type([1,2,5])
list
>>> type((1,2,5))
tuple
>>> type({1,2,5})
set
>>> c = {"level": 2, "code": "MAS212", "lecturer": "Dolan"}
>>> type(c)
dict
>>> c["lecturer"]
'Dolan'
```
Variables

- Data types are assigned to variables using `=`
  ```python
  >>> a = 3
  >>> b = 4
  >>> a + b
  7
  ```

- Check you understand the following:
  ```python
  >>> a = [1,2]
  >>> b = a
  >>> a.append(3)
  >>> b
  [1, 2, 3]
  ```

- Lists are *mutable*: they can be changed. Tuples and strings are *immutable*. 
Testing for equality and identity

- We test for equality with a double-equals `==`

  ```python
  >>> 0 == 1
  False
  >>> 1 == 1
  True
  ```

- Two lists are equal if their corresponding elements are equal:

  ```python
  >>> a = [1,2]
  >>> b = [1,2]
  >>> c = [2,1]
  >>> a == b
  True
  >>> a == c
  False
  ```
- Though $a$ and $b$ are equal, they are *not the same list*:

```python
>>> a = [1,2]; b = [1,2]
>>> a == b
True
>>> a is b
False
```

- Each object in memory (e.g. each list) has a unique ID:

```python
>>> id(a), id(b)
(139763716892936, 139763717044344)
```

- The `is` keyword compares the IDs, to see if two objects are actually the same object.

```python
>>> a is b
False
```
Testing inequalities

Further tests:

!= ‘not equal’
> ‘greater than’
>= ‘greater than or equal to’
< ‘less than’
<= ‘less than or equal to’

Examples:

```python
>>> a = 1.0; b = 1.0; c = 1.1;
>>> b > a
False
>>> c > a
True
>>> b >= a
True
>>> b != a
False
```
Lists

- Lists are great!
- Making a new list:

```python
>>> [0, 1, 4, 9, 16]  # you can specify elements explicitly
[0, 1, 4, 9, 16]
>>> list(range(5))  # or use a function to generate a list
[0, 1, 2, 3, 4]
>>> [i**2 for i in range(5)]  # or make a new list from another
[0, 1, 4, 9, 16]
```

- You can build a list up from scratch:

```python
>>> a = []
>>> a.append("horse")
>>> a.insert(0, "giraffe")
>>> a
['giraffe', 'horse']
```
Concatenate (i.e. join) two or more lists with +

```python
>>> a = ['horse', 'giraffe']
>>> b = ['kangaroo', 'hippo']
>>> a + b
['horse', 'giraffe', 'kangaroo', 'hippo']
```

Sort a list (e.g. alphabetically):

```python
>>> c = sorted(a+b)
>>> c
['giraffe', 'hippo', 'horse', 'kangaroo']
```

Reverse the list:

```python
>>> c.reverse()  # in place
>>> c
['kangaroo', 'horse', 'hippo', 'giraffe']
```
List comprehension

- We’ve seen how to make (e.g.) a list of square numbers:

```
>>> a = [i**2 for i in range(15)]
>>> a
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81, 100, 121, 144, 169, 196]
```

- Let’s break this down a bit:

  - `range(15)`: iterates over a list of integers 0...14
  - `for i in`: assigns `i` to each value in the list, in turn.
  - `i**2`: raises `i` to the power of 2 (squares it).

- Do you understand the following snippet?

```
>>> [i % 5 for i in a]
[0, 1, 4, 4, 1, 0, 1, 4, 4, 1, 0, 1, 4, 4, 1]
```
We can add an if condition when forming a list. For example,

```python
>>> [i**2 for i in range(15) if i**2 % 5 < 2]
[0, 1, 16, 25, 36, 81, 100, 121, 196]
```

We could use this (e.g.) to find the intersection of two lists:

```python
>>> a = [5,3,8,11]
>>> b = [8,1,6,3]
>>> [i for i in a if i in b]
[3, 8]
```

There's a better way:

```python
>>> set(a) & set(b)
set([8, 3])
```

In a set (unlike a list), duplicates are eliminated, and ordering is not significant.
List indexing

### Individual elements:

```python
>>> a = [i**2 for i in range(10)]
>>> a[0]  # the first element
0
>>> a[1]  # the second element
1
>>> a[-1]  # the last element
81
>>> a[-2]  # the second-to-last (penultimate) element
64
```
List slicing

- We can take slices of lists to get new lists

Examples:

```python
>>> a = [i**2 for i in range(10)]
>>> a[2:5]  # i.e. [a[2], a[3], a[4]]
[4, 9, 16]
>>> a[::2]  # every other element
[0, 4, 16, 36, 64]
>>> a[::-1]  # reversed list
[81, 64, 49, 36, 25, 16, 9, 4, 1, 0]
```

- The syntax here is `[first_index:last_index:step]`
- Omitted indices take default values: `[0:length_of_list:1]`
List slicing

List slicing can be used to modify part of a list:

```python
>>> a = list(range(10))
>>> a[1:3] = ['giraffe', 'iguana']
>>> a
[0, 'giraffe', 'iguana', 3, 4, 5, 6, 7, 8, 9]
```
Swaps

- Suppose we have two variables `a`, `b` and we wish to exchange their values, `a ↔ b`.
- It could be done like this:

  ```python
  >>> temp = a  # store in a temporary variable
  >>> a = b
  >>> b = temp
  ```

- In Python there’s a simpler way:

  ```python
  >>> a, b = b, a
  ```

- You can swap elements in lists in a similar way, e.g.

  ```python
  >>> a = list(range(5))
  >>> a[3], a[1] = a[1], a[3]
  >>> a
  [0, 3, 2, 1, 4]
  ```
Strings

- A ‘string’ of characters that can be indexed like a list.

Examples:

```python
>>> s = "This is a string"
>>> len(s)  # How many characters?
16
>>> s[0:5]  # Get first five characters
'This '
>>> s[::2]  # Get every other character
'Ti sasrn'
>>> s[::-1]  # Reverse the string
'gnirts a si sihT'
>>> 'i' in s  # Is there an 'i' in the string?
True
>>> 'e' in s  # Is there an 'e' in the string?
False
```

- Strings are **immutable**: e.g. `s[0] = 't'` will not work
Strings

Strings can be converted to lists and back again:

```python
>>> s = "A string"
>>> list(s)  # a list of characters
['A', '', 's', 't', 'r', 'i', 'n', 'g']
>>> " ".join(list(s))  # double spaced
'A s t r i n g'
```

Strings can be manipulated with `split` and `join`:

```python
>>> s = "This is a sentence"
>>> s.split(" ")  # get a list of words
['This', 'is', 'a', 'sentence']
>>> "---".join(s.split(" ")))  # Re-join words with three hyphens
'This---is---a---sentence'
```
Strings

- Changing the case:

```python
>>> s = "A String"
>>> s.upper()
'A STRING'
>>> s.lower()
'a string'
>>> s.capitalize()
'A string'
```
Strings

Find-and-replace is easy:

```python
>>> s = "A hexagon has six sides"
>>> s.replace("hexagon", "cube")
'A cube has six sides'
```
String formatting

- Often, you will want to format data in a particular way.
- Data types can be converted to strings quickly using `repr`.

```python
>>> p = 3.1415926
>>> print("Pi is roughly " + repr(p))
Pi is roughly 3.1415926
```

- For more control, you can use the `format` method:

```python
>>> for i in range(2, 5):
...     print("Square root of {} is {}".format(i, i**0.5))
...
Square root of 2 is 1.41421356237
Square root of 3 is 1.73205080757
Square root of 4 is 2.0
```
String formatting

- For (e.g.) three decimal places:

```python
>>> for i in range(2, 5):
...     print("Square root of %i is %.3f" % (i, i**0.5))
... 
Square root of 2 is 1.414
Square root of 3 is 1.732
Square root of 4 is 2.000
```

- Syntax: combine a string with a tuple using %.
- Here %i means ‘integer’ and %.3f means ‘float with three digits after decimal point’.
String formatting

- Format codes are common to many languages (Fortran, C, matlab, etc.). Here are the key letters:
  - `d` signed decimal integer
  - `i` integer
  - `u` unsigned decimal integer
  - `f` floating point real number
  - `E` exponential notation (uppercase ‘E’)
  - `e` exponential notation (lowercase ‘e’)
  - `g` the shorter of %f and %e
  - `s` string conversion
  - `c` character

- Examples:

```python
>>> print("%f %.3f %e %3.3e" % (p,p,p,p))
3.141593 3.142 3.141593e+00 3.142e+00
>>> print("%02i %05i" % (3, 3))
03 00003
```
**chr and ord**

- Single characters can be converted to Unicode integers using `ord` ...
- ... and vice versa with `chr`

```python
>>> ord('A')
65
>>> ord('!')
33
>>> chr(65)
'A'
>>> chr(33)
'!
```

**Challenge:** Can you use `ord` and `chr` to encrypt some text with a Caesar shift?
for loops

- We have already met for loops, in which a variable iterates over a list.

- Example:

```python
>>> for i in range(1, 6):
...     print("The square of \%d is \%d" % (i, i**2))
...
The square of 1 is 1
The square of 2 is 4
The square of 3 is 9
The square of 4 is 16
The square of 5 is 25
```

- The *body* of the for loop is indented by one tab-space.
while loops

The same result can be achieved with a while loop. Example:

```python
>>> i = 1
>>> while i < 6:
...     print("The square of %d is %d" % (i, i**2))
...     i += 1  # increase i by 1
...
The square of 1 is 1
The square of 2 is 4
The square of 3 is 9
The square of 4 is 16
The square of 5 is 25
```

Danger: If you forget to increment the counter variable (i), then the loop will not end! (This may crash your computer and you may lose your work!)
Breaking out of a loop

- To exit a loop early, use the `break` keyword. Example:

```python
>>> for i in range(10):
...     if i**2 >= 20:
...         break
...     print(i)
...
0
1
2
3
4
```

- Note that this is not a good way of finding numbers whose square is less than 20. Somewhat better would be:

```python
>>> [i for i in range(10) if i**2 < 20]
[0, 1, 2, 3, 4]
```
Control flow: if–else–elif

- if statements divert the flow of the program, depending on whether a condition is satisfied.
- elif is shorthand for ‘else if’.
- Example (try changing the value of a):

```python
>>> a = "horse"
>>> if a == "pig":
...     print("Oink!")
... elif a == "horse":
...     print("Neigh!")
... else:
...     print("-----")
...
Neigh!
```
Functions

- A function is like a ‘black box’ that takes one or more inputs (parameters) and produces one output.
- (Since the output may be a container type (e.g. list), it can actually produce several outputs).
- New functions are defined with the `def` and `return` keywords. Example:

```python
>>> def square(i):
...     return i**2
... 
>>> square(7)    # try the function
49
>>> square(7.0)
49.0
```

- Try passing a list or str data type to square – what happens?
- More info: https://docs.python.org/release/1.5.1pl/tut/functions.html
Functions

- Functions can have *optional* named parameters. Example:

```python
>>> def raisepower(a, power=2):
...     # Note that 'power' is assigned a default value of 2
...     return a**power
...
>>> raisepower(3)
9
>>> raisepower(2, power=3)
8
>>> raisepower(2, 0.5)
1.4142135623730951
```

- The function may be called *without* specifying optional parameters, or,
- optional parameters may be set by name, or in order.
Docstrings

- At the start of a function, you may write a (multiline) **docstring** to explain what the function does:

```python
>>> def raisepower(a, power=2):
...     """This is a docstring.
...     This function raises the first parameter to
...     the power of the second parameter."""
...     return a**power
...  
>>> help(raisepower)
```

- Now the docstring should appear in the ‘help’ for the function.
- In ipython, there is enhanced help. Try entering `?raisepower`. 
An example function: Fibonacci sequence

- Let’s try a function to compute the Fibonacci sequence from the recurrence relation \( f_{k+1} = f_k + f_{k-1} \):

```python
>>> def fibonacci(n=10):
...     r"""Computes a list of the first n Fibonacci numbers."""
...     l = [0, 1]
...     for i in range(n-1):
...         l.append(l[-1] + l[-2])
...     return l
...
```

- Example output

```python
>>> fibonacci(10)
[0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55]
```
An example function: Fibonacci sequence

The ratio of successive terms should tend towards the Golden Ratio \((\sqrt{5} + 1)/2\). Let’s check this:

```python
>>> l = fibonacci(100)
>>> l[-1] / float(l[-2])  # an approximation to the Golden Ratio
1.618033988749895
>>> (5**0.5 + 1)/2.0  # the true Golden Ratio
1.618033988749895
```
Functions: Scope

- Functions can **modify** container-type parameters.

```python
>>> def addanimal(a):
...     a.append("aardvark")
...
>>> l = ["horse"]
>>> addanimal(l)
>>> l
['horse', 'aardvark']
```

- But this doesn’t work the same way for numeric types:

```python
>>> def addone(x):
...     x += 1
...
>>> x = 3
>>> addone(x)
>>> x
3
```

- Solution: declare `x` with the `global` keyword.
Functions: Scope

- A function can define & use its own **local** variables.
- It can also use **global** variables, defined elsewhere in the program.

```
>>> def fn():
...    a = 5  # local variable a
...    print(a,b)  # global variable b
...    print(a,b)

>>> b = 3
>>> fn()
5 3
```

**Warning**: If the global variable `b` has not been defined, then calling the function leads to an error.
Q. What if a function defines a **local** variable which has the same name as a **global** variable?

A. Within the function, the local variable is used. When the function ends, the local variable ‘falls out of scope’ (it is discarded). The global variable does not change.

Example:

```python
>>> def fn():
...     a = 3  # local variable
...     print(a)
...
>>> a = 4  # global variable
>>> fn()  # this will print the value of the local variable
3
>>> print(a)  # the global variable has not changed
4
```
Functions: Scope

- If a local & global variables exist with the same name, then Python uses the local variable.
- **LEGB**: Python searches scope in this order: Local, Enclosing, then Global, then Built-in (e.g., `print()`).
- **Warning**: It is very easy to ‘overwrite’ built-in functions!
- **Example**:

```python
>>> a = [3,4,5]
>>> sum(a)  # 'sum' is a built-in function
12
>>> sum = 1234  # But what if we use 'sum' as a variable name?
>>> sum(a)  # Now the 'sum' function doesn't work!
Traceback (most recent call last):
  File "<ipython-input-25-dd0a8bf65284>"", line 1, in <module>
    sum(a)
TypeError: 'int' object is not callable
```
We defined a variable ‘sum’ with the same name as a built-in function.

The variable did not overwrite the built-in function, but the variable took precedence.

Python uses the first definition it finds: Local, Enclosing, Global, Built-in.

If we ‘delete’ the variable, the built-in function works again

```python
>>> del sum
>>> sum(a)  # Now Python can find the built-in function again.
12
```

To delete all global variables, use %reset.
Functions: Good practice

- Split your programming challenge into small tasks, and write a function for each task.
- Use docstrings to describe the task, & the inputs and outputs.
- Avoid using global variables in functions, instead use function parameters, or,
- Make global variables explicit using the `global` keyword.
- Avoid repetition of code (i.e. don’t copy-and paste lines of code making small changes).
- Instead, write a function, then call it inside a loop, to modify its parameters.
Functions: Good practice

Example: Suppose you had some repetitive task:

```python
>>> print("The mouse eats cheese.")
>>> print("The bird eats seeds.")
>>> print("The cat eats fish.")  # etc
```

The task can be achieved using a function inside a loop:

```python
>>> def eats(a, f):
...     print("The %s eats %s" % (a, f))
>>> data = (("mouse", "cheese"), ("bird", "seeds"), ("cat", "fish"))
>>> for (a,f) in data:
...     eats(a, f)
The mouse eats cheese.
The bird eats seeds.
The cat eats fish.
```
Built-in functions

- Python has many built-in functions:
  https://docs.python.org/3/library/functions.html

- Here are just a few:
  - `abs` : find the absolute value
  - `chr` : convert an ASCII code to a character
  - `id` : find the unique ID of an object
  - `open` : open a file
  - `ord` : convert a character to an ASCII code
  - `print` : print to screen (e.g.)
  - `range` : make a list from an arithmetic progression
  - `repr` : convert object to string
  - `round` : round a number
  - `sum` : sum a list
  - `type` : get the data type
  - `zip` : zip a pair of lists into a list of tuples

- Use `?` to find out more about a function in ipython, e.g.:
  `?range`
Reserved words

- Certain words are reserved by the Python language, and cannot be used for (e.g.) variable names:

```
and, as, assert, break, class, continue, def, del, elif, else, except, False, finally, for, from, global, if, import, in, is, lambda, None, nonlocal, not, or, pass, raise, return, True, try, with, while, yield.
```

- The in-built function names should **not** be used for e.g. variable names

```
print, range, sorted, reversed, sum, # etc.
```
Namespace

- **Name collision**: When two or more variables or functions share the same name (e.g. ‘sum’).

- **Namespace**: A structured hierarchy that allows re-use of names in different contexts, to prevent ‘name collisions’.

Example #1: a file system:
/code/ and /mas212/code/ are different directories.

Example #2: Packages & modules in Python:
numpy.sin() and math.sin() are different functions
Modules

- A **module** is a file containing variable & function definitions and statements.

- Modules are loaded using the `import` statement.

- Example: the `math` module:

  ```python
  >>> import math
  >>> math.pi  # a variable, initialized to the mathematical constant pi
  3.141592653589793
  >>> math.sqrt(2)  # a function in the math module
  1.4142135623730951
  ```

- A **package** is a group of modules which can be referred to using “dotted modules names”: `package_name.module_name.function()`. For example, `scipy.linalg.det()` for determinants.
Modules

- Modules can be given shortened names:
  ```python
  >>> import math as m
  >>> m.exp(1)
  2.718281828459045
  ```

- Specific variables/functions can be loaded into the namespace:
  ```python
  >>> from math import sin, pi
  >>> sin(pi/4.0)
  0.7071067811865475
  ```

- It is **bad practice** to import all module contents into the namespace
  ```python
  >>> from math import *
  >>> cos(pi/6.0)
  0.8660254037844387
  ```
Modules

- Python comes with a standard library which includes built-in modules:
  https://docs.python.org/3/library/

- Useful built-in modules include:
  math, cmath, random, decimal, datetime, io, os, sys

- There are many more modules & packages beyond the standard library

- Three key packages for scientific computing are:
  numpy, matplotlib, scipy

- Others at
  https://wiki.python.org/moin/UsefulModules
**Modules: making your own**

- Any Python file (*.py) is a module.
- For example, I could save the Fibonacci function definition in a file called `fib.py`. Starting the interpreter in the same directory, I import it just like a built-in module:

```python
>>> import fib
>>> fib.fibonacci(10)
[0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55]
```

- Where does Python look for module files?
  1. First it checks for in-built modules.
  2. Then the current directory.
  3. Then in directories specified by `sys.path`

```python
>>> import sys
>>> sys.path  # a list of directories to search
```

- For more on creating modules and packages, see [https://docs.python.org/3/tutorial/modules.html](https://docs.python.org/3/tutorial/modules.html)
A **script** is a module that is intended to be executed.

Any Python module may be executed as a script. When imported as a *module* the filename is used as the basis for the module name. When run as a *script* the module is named `__main__`. So script-specific code can be enclosed in a block under `if __name__=="__main__"`

- The first line of a script: `#!/usr/bin/python`
- A script may be run from the command line: `python script_name.py`
- Scripts can process command-line arguments:

```python
import sys
print("Number of arguments: ", len(sys.argv), " arguments.")
print("Argument List: ", str(sys.argv))
```

- [https://docs.python.org/3/tutorial/stdlib.html#command-line-arguments](https://docs.python.org/3/tutorial/stdlib.html#command-line-arguments)
Object-orientated programming

- Programming styles broadly divide into two categories: **procedural** and **object-orientated**.

- Scientists typically write procedural-style code:
  - functions process data . . .
  - . . . but do not retain that data, or remember their state.

- Developers typically prefer the object-orientated paradigm:
  - An ‘object’ represents a concept or thing . . .
  - . . . which holds data about itself (‘attributes’)
  - . . . and has functions (‘methods’) to manipulate relevant data
  - The methods may change the attributes (i.e. modify the state of the object).

- An object is created (‘instantiated’) from a “blueprint” called a ‘**class**’.
Object-orientated programming

- In Python, **everything is an object**.
- For e.g., a Python string is an instance of the `str` class.
- The `str` class has attributes (i.e. the string itself) and methods (e.g. `split()`)
- It also has ‘hidden’ methods for implementing e.g. string slicing.

```python
>>> s = str("hello, there")  # Instantiates a new string object
>>> s.split(",")  # calls the 'split' method of the string object
['hello', ' there. ']
>>> s.capitalize()  # This method takes no parameters
'Hello, there.'
>>> s[3::2]  # This string slice is equivalent to ...
'l,tee'
>>> s.__getitem__(slice(3,None,2))  # ... this.
'l,tee'
```
You can make your own class. For example:

```python
# Save this in a file e.g. 'person.py'.
class Person:
    '''An abstract class representing a person.'''
    name = 'unknown'  # an attribute

    def __init__(self, name):
        '''Initialise the person by giving their name.'''
        self.name = name

    def sayhello(self):
        '''A method to introduce the person.'''
        print("Hello, my name is {}".format(self.name))
```
Object-orientated programming

- You can then use this class as follows:

```python
>>> from person import Person
>>> p = Person("Elvis Presley")  # instantiate
>>> p.sayhello()  # call the method
Hello, my name is Elvis Presley
>>> p.name = "Amelia Earhart"  # change the attribute
>>> p.sayhello()
Hello, my name is Amelia Earhart
```

- For more on object-orientated programming, see e.g. Sec. 4.6 in *Learning scientific programming with Python* (Christian Hill).