Software skills for librarians: Library carpentry

Module 3: Introduction to programming in Python

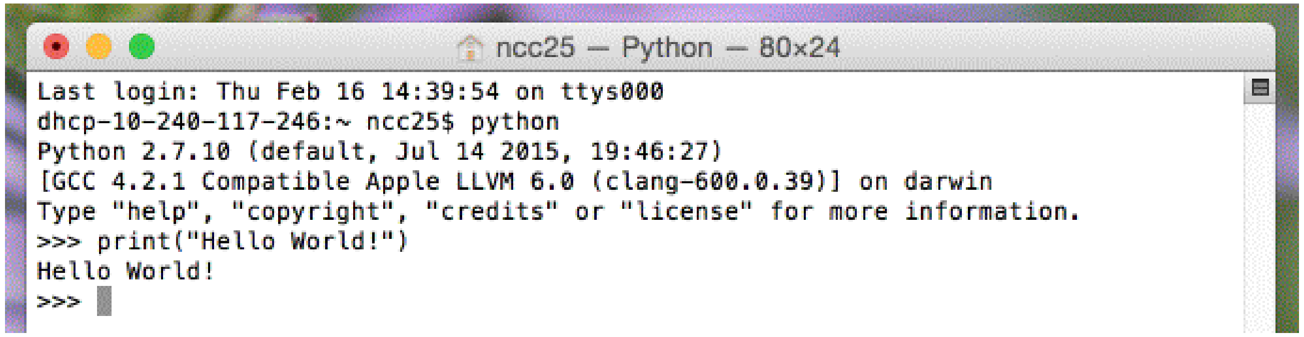
First program in Python

```
# Hello world in Python
print("Hello world")
```

- First line is a comment
- Second line is a statement
- “Hello world” is a string literal

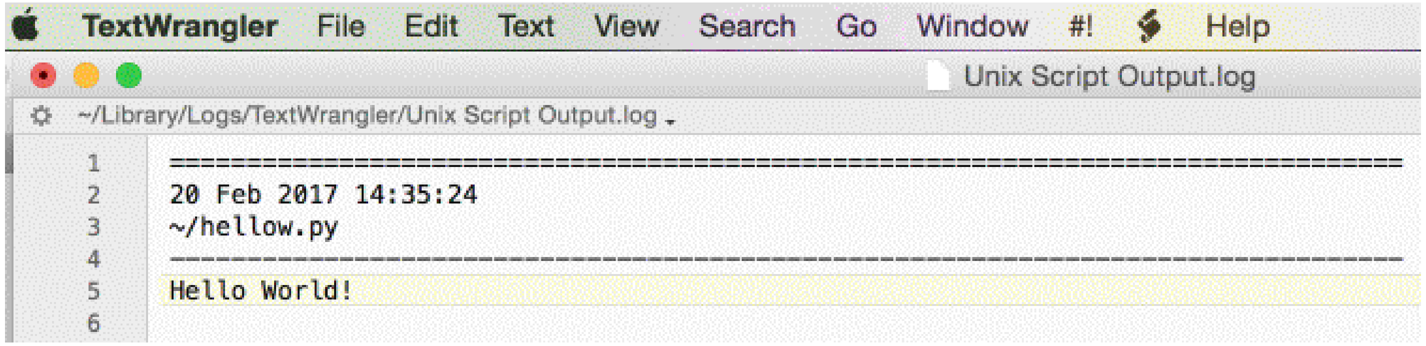
Running python

- Interactively



```
Last login: Thu Feb 16 14:39:54 on ttys000
dhcp-10-240-117-246:~ ncc25$ python
Python 2.7.10 (default, Jul 14 2015, 19:46:27)
[GCC 4.2.1 Compatible Apple LLVM 6.0 (clang-600.0.39)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> print("Hello World!")
Hello World!
>>> 
```

- From a file


TextWrangler File Edit Text View Search Go Window #! Help
Unix Script Output.log
~/Library/Logs/TextWrangler/Unix Script Output.log

```
================================================================================
20 Feb 2017 14:35:24
~/hellow.py
--------------------------------------------------------------------------------
Hello World!
```

Programming languages

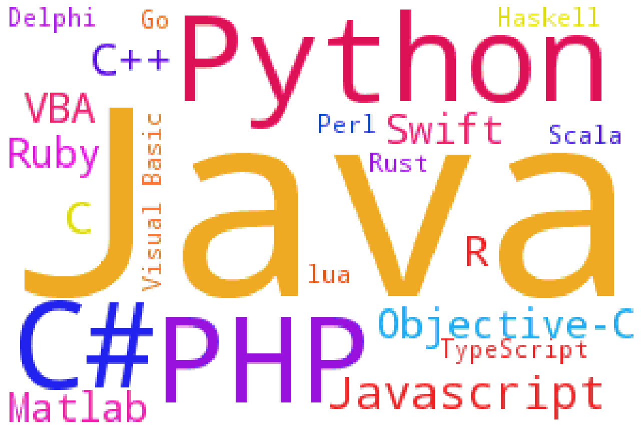
- Computers only understand machine code:

  A list of numbers or codes representing simple instructions like add

- High-level languages are more natural for humans
- Need to be translated for the computer:

  Either compiled or interpreted

- Advantages and disadvantages of each language



Programs

- A series of statements in a given language
- The individual steps needed to complete a task
- A statement can be:

  A function call

  A control statement like a loop

  An expression

  Or an assignment

Variables

- A named container for a piece of data
- Numbers, 42, 3.142
- Strings, “Great expectations”, “Charles Dickens”
- Lists, ["245", "260", "300"]
- Dictionaries, {"245": "Title", "260": "Publisher", "300": "Description"}
- Tuples, ("eng", "ger", "fre")

Assignments

- To use a variable assign to it:

```
name="Nicholas"
```

- We can now use that variable:

```
print("My name is ", name)
```

- Perform operations on it:

```
bigname=name.upper()
```

- Change it:

```
name="Alan Turing"
```

Operations on lists

- Accessing a single entry `marc[1]`
- Slicing `marc[2:3]`
- Concatenation `marc+["490", "650"]`
- Adding an item `marc.append("700")`
- Deleting an item `marc.pop(4)`
- Sorting `marc.sort()`
- Reversing `marc.reverse()`

Control expressions

- Conditional:

```
if condition:
    statement block
else:
    statement block
```

- Be careful with indentation
- Remember the colon!
- One statement per line

Control expressions

- Iteration:

```
while condition:
    statement block

for index in list:
    statement block
```

- List may be a range
- Infinite loops: `while True:`
- Exit using `if exittest: break`

Functions

- A named series of statements
- Used frequently in a program

    Saves space

    Makes code easier to understand

- Can take arguments
- And return a result

Function definition and arguments

```
def fn_name(arguments):
    statement block
    return result
```

- Arguments are values used in the function
- Names valid only within the function
- Values copied when function is called

Function calls

```
value=fn_name(arguments)
```

- Arguments are not modified by function
- `value` is set to return result (if any)

Classes and OOP

- A Class is a data structure
- And the functions which act on it
- An object is an instance of the class
- For example: a MARC record
- Functions include add field, remove field, display
- Shared between all MARC record objects

Initialising objects

- `object=class_name()`

  Calls special method `__init__`

- Then `object.method()`

  Calls method on object

- Like a function call with extra argument

Defining classes

```
class class_name:
    def __init__(self):
        self.data=0

    def method(self):
```

- All classes need `__init__`
- Called when object is created
- Special argument self refers to the object

Files

- Like standard input and output but on disc
- Can be opened for input, output or both
- Text and binary files
- Sequential files and pointer

Files in action

- First open a file: `infile=open("name.txt", "r")`

  `outfile=open("new.txt", "w")`
- Reading: `string=infile.read(n)`

  `string=infile.readline()`

  `list=infile.readlines()`
- Writing: `outfile.write(string)`

  `outfile.write(list)`
- Get current position: `p=infile.tell()`
- Set current position: `p=outfile.seek(p)`
- Finally close a file: `outfile.close()`

Iterators

- Similar to for loops over a list
- Can be used with any ‘iterable’ object
- Internally calls the `next()` method
- So: `for line in infile:`
- Equivalent to: `for line in infile.readlines():`
- With statement:

```
with open("name.txt", "r") as fh:
    str=fh.readline()
```

Modules

- Modules are library files
- Collections of code common to several projects
- Can contain: global variables, functions and classes
- Use with import keyword: `import math`
- Can import selected attributes:

```
from pymarc import MARCReader
```

Inside modules

- Modules can contain:

  Global variables: `math.pi`

  Functions: `math.sin(math.pi/2)`

  Classes: `regex=re.compile("\d{10}")`

- Similar notation for accessing class methods:

  Result depends on whether `re` is a module or a class

  Be careful when choosing names

- Importing a selection loads those attributes into current namespace

Exceptions

- Errors do sometimes happen!
- Exceptions allow us to handle these gracefully
- Easier than testing for an error after every operation
- Exceptions can also be used for:

  Event notification

  Handling special cases

  Termination conditions

Using exceptions

- Raise an exception when an error occurs:

```
if not i in dict.keys():
    raise KeyError
```

- Catch an exception:

```
try:
    fp=open("name.txt", "r")
except IOError:
    print("File not found")
```

Further reading

- Learning Python / Mark Lutz — 5th ed.

  O’Reilly, 2013 — ISBN 9781449355739

- Python in a nutshell / Alex Martelli — 2nd ed.

  O’Reilly, 2006 — ISBN 9780596100469

- Automate the boring stuff with Python / Al Sweigart.

  No Starch Press, 2015 — ISBN 9781593275990