Week 8: Fundamentals of Python Programming I

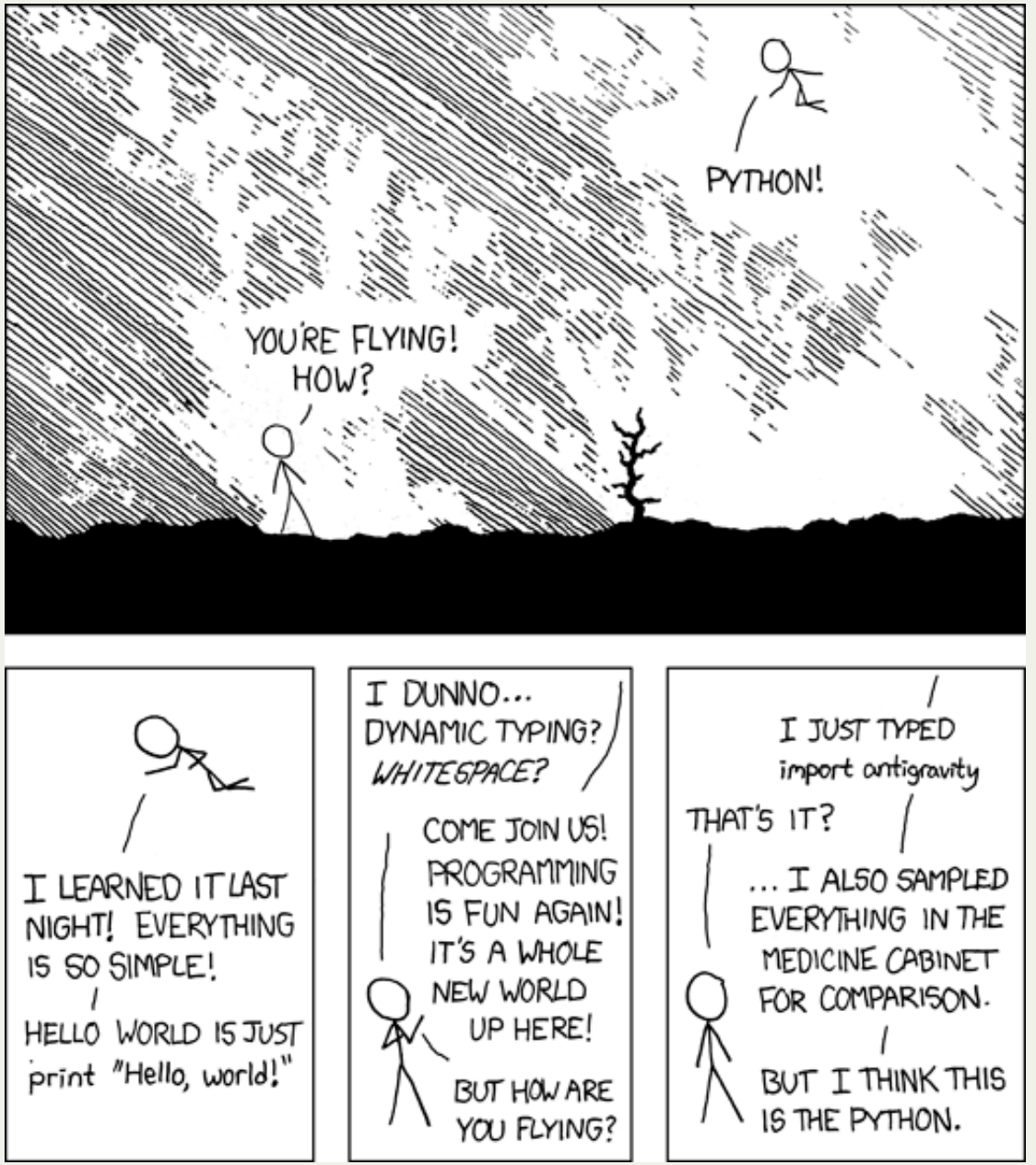
POP77001 Computer Programming for Social Scientists

Tom Paskhalis

Overview

- Python programs and their components
- Objects and operators
- Scalar and non-scalar types
- Indexing
- Methods and functions

Introduction to Python


PYTHON!
YOU'RE FLYING! HOW?
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.

xkcd

Python Background

- Started as a side-project in 1989 by Guido van Rossum, BDFL (benevolent dictator for life) until 2018.
- Python 3, first released in 2008, is the current major version.
- Python 2 support stopped on 1 January 2020.

The Zen of Python

```
1  import this
```

```
The Zen of Python, by Tim Peters

Beautiful is better than ugly.
Explicit is better than implicit.
Simple is better than complex.
Complex is better than complicated.
Flat is better than nested.
Sparse is better than dense.
Readability counts.
Special cases aren't special enough to break the rules.
Although practicality beats purity.
Errors should never pass silently.
Unless explicitly silenced.
In the face of ambiguity, refuse the temptation to guess.
There should be one-- and preferably only one --obvious way to do it.
Although that way may not be obvious at first unless you're Dutch.
Now is better than never.
Although never is often better than *right* now.
If the implementation is hard to explain, it's a bad idea.
If the implementation is easy to explain, it may be a good idea.
Namespaces are one honking great idea -- let's do more of those!
```

Python Basics

- Python is an **interpreted** language (like R and Stata).
- Every program is executed one *command* (aka *statement*) at a time.
- Which also means that work can be done interactively.

```
1 'POP' + '77001'
```

```
'POP77001'
```

Python Conceptual Hierarchy

- Python programs can be decomposed into modules, statements, expressions, and objects, as follows:

  1. *Programs* are composed of *modules*
  2. *Modules* contain *statements*
  3. *Statements* contain *expressions*
  4. *Expressions* create and process *objects*

Python Objects

- Everything that Python operates on is an object.
- This includes numbers, strings, data structures, functions, etc.
- Each object has a **type** (e.g. string or function) and internal data
- Objects can be **mutable** (e.g. list) and **immutable** (e.g. string)

Operations

Operators

- Objects and operators are combined to form **expressions**.
- Key **operators** are:
  - Assignment (`=`, `+=`, `-=`, `*=`, `/=`)
  - Arithmetic (`+`, `-`, `*`, `**`, `/`, `//`, `%`)
  - Boolean (`and`, `or`, `not`)
  - Relational (`==`, `!=`, `>`, `>=`, `<`, `<=`)
  - Membership (`in`)

Mathematical Operations

Arithmetic operations:

```
1 1 + 1
```

```
2
```

```
1 5 - 3
```

```
2
```

```
1 6 / 2
```

```
3.0
```

```
1 4 * 4
```

```
16
```

```
1 # As in R, Python comments start with #
2 # Exponentiation
3 2 ** 4
```

```
16
```

Advanced mathematical operations:

```
1 # Integer division (remainder is discarded)
2 7 // 3
```

```
2
```

```
1 # Modulo operation (only remainder is retained)
2 7 % 3
```

```
1
```

Logical Operations

```
3 != 1 # Not equal
```

```
True
```

```
3 > 3 # Greater than
```

```
False
```

```
3 >= 3 # Greater than or equal
```

```
True
```

```
# True if either first or second operand is True, False otherwise
# Analogous to R's | operator
False or True
```

```
True
```

```
# True if both first and second operand are True, False otherwise
# Analogous to R's & operator
False and True
```

```
False
```

```
3 > 3 or 3 >= 3 # Combining 3 Boolean expressions
```

```
True
```

Membership Operations

Operator `in` returns `True` if an object of the left side is in a sequence on the right.

```
1 # Strings are also sequences in Python
2 'a' in 'abc'
```

```
True
```

```
1 3 in [1, 2, 3] # [1,2,3] is a list
```

```
True
```

```
1 3 not in [1, 2, 3]
```

```
False
```

Operator Precedence

Operator	Description
(expressions…),	Binding or parenthesized expression,
[expressions…],	list display,
{key: value…},	dictionary display,
{expressions…}	Set display
x[index],	subscription,
x[index:index],	slicing,
x(arguments…),	call,
x.attribute	Attribute reference
await x	Await expression
**	Exponentiation
+x, -x, ~x	Positive, negative, bitwise NOT
*, @, /, //, %	Multiplication, matrix multiplication, division, floor division, remainder
+, -	Addition and subtraction
<<, >>	Shifts
&	Bitwise AND
^	Bitwise XOR
|	Bitwise OR
`in`, `not in`, `is`, `is not`, <, <=, >, >=, !=, ==	Comparisons, including membership tests and identity tests
not x	Boolean NOT
and	Boolean AND
or	Boolean OR
if – else	Conditional expression
lambda	Lambda expression
:=	Assignment expression

Extra

Python Documentation on Operator Precedence

Assignment

Assignment Operations

- Assignments create object references.
- **Target** (or **name**) on the left is assigned to **object** on the right.

```
1 x = 3
```

```
1 x
```

```
3
```

Memory Address

```
1 x += 2 # Increment assignment, equivalent to x = x + 2
```

```
1 x
```

```
5
```

Assignment vs Comparison

As = (assignment) and == (equality comparison) operators appear very similar, they sometimes can create confusion.

```
1 x = 3 # Assignment
```

```
1 x
```

```
3
```

```
1 x == 3 # Equality comparison
```

```
True
```

Object Types

Divisibility & Mutability

- In Python it is useful to think of objects storing:
  - Individual values (**scalars**) or
  - **Sequences** of elements
- Scalar objects are indivisible and immutable.
- Sequences can be both **mutable** and **immutable**.
- 4 main types of scalar objects in Python:
  - Integer (`int`)
  - Real number (`float`)
  - Boolean (`bool`)
  - Null value (`None`)

Scalar Types

- All scalar types are indivisible and immutable

```
1 type(7)
```

```
<class 'int'>
```

```
1 type(3.14)
```

```
<class 'float'>
```

```
1 type(True)
```

```
<class 'bool'>
```

```
1 # None is the only object of NoneType
2 type(None)
```

```
<class 'NoneType'>
```

- Scalar type conversion (casting) can be done using type names as functions:

```
1 int(3.14)
```

```
3
```

```
1 str(42)
```

```
'42'
```

Non-scalar Types

- Non-scalar objects are all types of **sequences**.
- This allows indexing, slicing and other interesting operations.
- Most common sequences in Python are:
  - String (`str`) - *immutable ordered* sequence of Unicode characters
  - Tuple (`tuple`) - *immutable ordered* sequence of elements
  - List (`list`) - *mutable ordered* sequence of elements
  - Set (`set`) - *mutable unordered* collection of unique elements
  - Dictionary (`dict`) - *mutable unordered* collection of key-value pairs

Sequences: Example

```
1 s = 'time flies like a banana'
2 t = (0, 'one', 1, 2)
3 l = [0, 'one', 1, 2]
4 o = {'apple', 'banana', 'watermelon'}
5 d = {'apple': 150.0, 'banana': 120.0, 'watermelon': 3000.0}
```

```
1 type(s)
```

```
<class 'str'>
```

```
1 type(t)
```

```
<class 'tuple'>
```

```
1 type(l)
```

```
<class 'list'>
```

```
1 type(o)
```

```
<class 'set'>
```

```
1 type(d)
```

```
<class 'dict'>
```

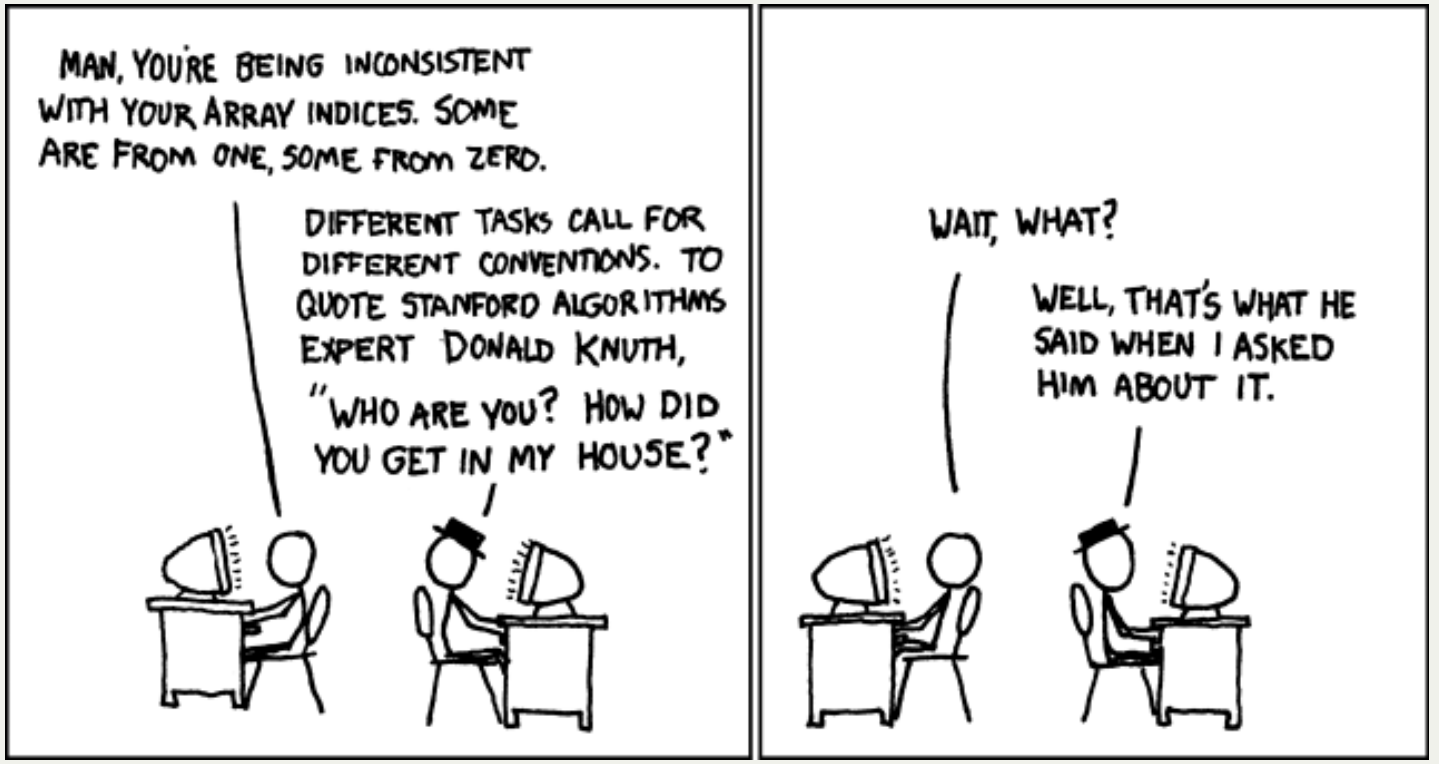
Working with Objects

Indexing and Subsetting in Python

- **Indexing** can be used to subset individual elements from a sequence.
- **Slicing** can be used to extract sub-sequence of arbitrary length.
- Use square brackets `[]` to supply the index (indices) of elements:

```
object[index]
```

Indexing in Python Starts from 0



xkcd

Extra

Why Python uses 0-based indexing by Guido van Rossum

Why numbering should start at zero by Edsger Dijkstra

Subsetting Strings

```
1 s
```

```
'time flies like a banana'
```

```
1 # Length of string (including whitespaces)
2 len(s)
```

```
24
```

```
1 # Subset 1st element (indexing in Python starts from zero!)
2 s[0]
```

```
't'
```

```
1 # Subset all elements starting from 6th
2 s[5:]
```

```
'flies like a banana'
```

```
1 # Strings can be concatenated together
2 s + '!'
```

```
'time flies like a banana!'
```

Objects Have Methods

- Python objects have **methods** associated with them.
- They can be thought of function-like objects.
- However, their syntax is `object.method()`
- As opposed to `function(object)`.

```
1 len(s) # Function
```

```
24
```

```
1 s.upper() # Method (makes string upper-case)
```

```
'TIME FLIES LIKE A BANANA'
```

String Methods

Some examples of methods available for strings:

```
1 # Note that only the first character gets capitalized
2 s.capitalize()
```

```
'Time flies like a banana'
```

```
1 # Here we supply an argument 'sep' to our methods call
2 s.split(sep = ' ')
```

```
['time', 'flies', 'like', 'a', 'banana']
```

```
1 # Arguments can also be matched by position, not just name
2 s.replace(' ', '-')
```

```
'time-flies-like-a-banana'
```

```
1 # Methods calls can be nested within each other
2 '-'.join(s.split(sep = ' '))
```

```
'time-flies-like-a-banana'
```

Extra

Python string methods

Method Chaining

- Methods can be chained together to perform multiple operations on the same object.
- The output of one method becomes the input of the next method.

```
1 s
```

```
'time flies like a banana'
```

- Instead of applying string methods one by one, we can chain them together:

```
1 s.replace(' a ', ' an ').replace('banana', 'arrow').capitalize().split(sep = ' ')
```

```
['Time', 'flies', 'like', 'an', 'arrow']
```

- For ease of reading, we can break the chain into multiple lines:

```
1 s_as_l = (
2   s
3   .replace(' a ', ' an ')
4   .replace('banana', 'arrow')
5   .capitalize()
6   .split(sep = ' ')
7 )
8 s_as_l
```

```
['Time', 'flies', 'like', 'an', 'arrow']
```

Tuples

Tuples can contain elements of different types:

```
1 t
```

```
(0, 'one', 1, 2)
```

```
1 len(t)
```

```
4
```

```
1 t[1:]
```

```
('one', 1, 2)
```

Like strings tuples can be concatenated:

```
1 t + ('three', 5)
```

```
(0, 'one', 1, 2, 'three', 5)
```

Lists

Like tuples lists can contain elements of different types:

```
1 l
```

```
[0, 'one', 1, 2]
```

Unlike tuples lists are mutable:

```
1 l[1] = 1
```

```
1 l
```

```
[0, 1, 1, 2]
```

```
1 # Compare to tuple
2 t[1] = 1
```

```
TypeError: 'tuple' object does not support item assignment
```

Indexing and Slicing Lists

- Indexing and slicing lists (and other ordered sequences) is similar to using `seq()` function in R.
- The general syntax for slicing in Python is `seq[start:stop:step]`.
- Note that the `stop` index is not included in the slice.
- If `start` or `stop` are omitted, they default to the beginning and end of the sequence respectively.
- If `step` is omitted, it defaults to 1.

Indexing and Slicing Lists: Example

```
l
```

```
[0, 1, 1, 2]
```

```
# Subset all elements starting from 2nd
l[1:]
```

```
[1, 1, 2]
```

```
# Subset the last element
l[-1]
```

```
2
```

```
# Subset every second element,
# list[start:stop:step]
l[::2]
```

```
[0, 1]
```

```
# Subset all elements in reverse order
l[::-1]
```

```
[2, 1, 1, 0]
```

Sets

```
1 o
```

```
{'apple', 'banana', 'watermelon'}
```

```
1 # Sets retain only unique values
2 {'apple', 'apple', 'banana', 'watermelon'}
```

```
{'apple', 'banana', 'watermelon'}
```

```
1 # Sets also have methods
2 o.difference({'banana'})
```

```
{'apple', 'watermelon'}
```

```
1 # Some methods can be expressed as operators
2 o - {'banana'}
```

```
{'apple', 'watermelon'}
```

```
1 # Sets can be compared (e.g. one being subset of another)
2 {'apple'} < o
```

```
True
```

```
1 # Unlike strings, tuples and lists, sets are unordered
2 o[1]
```

```
TypeError: 'set' object is not subscriptable
```

Dictionaries

```
1 # key:value pair, fruit_name:average_weight
2 d
```

```
{'apple': 150.0, 'banana': 120.0, 'watermelon': 3000.0}
```

```
1 # Unlike strings, tuples and lists, dictionaries are indexed by 'keys'
2 d['apple']
```

```
150.0
```

```
1 # Rather than integers
2 d[0]
```

```
KeyError: 0
```

```
1 # They are, however, mutable like lists and sets
2 d['strawberry'] = 12.0
3 d
```

```
{'apple': 150.0, 'banana': 120.0, 'watermelon': 3000.0, 'strawberry': 12.0}
```

Conversion of Non-scalar Types

```
## Tuple
t
```

```
(0, 'one', 1, 2)
```

```
## Convert to list with a `list` function
list(t)
```

```
[0, 'one', 1, 2]
```

```
## Conversion to set retains only unique values
set([0, 1, 1, 2])
```

```
{0, 1, 2}
```

- **List comprehension**, a more Pythonic way of implementing loops and conditionals, can also be used for converting sequences.
- It has the the general form of `[expr for elem in iterable if test]`.

```
[x for x in t]
```

```
[0, 'one', 1, 2]
```

```
[x for x in t if type(x) != str]
```

```
[0, 1, 2]
```

`None` Value

- `None` is a Python null object.
- It is often used to initialize objects.
- And it is a return value in some functions (more on that later).

```
# Initialization of some temporary variable, which can re-assigned to another value later
none = None
none
```

```
# Here we are initializing a list of length 10
none_l = [None] * 10
none_l
```

```
[None, None, None, None, None, None, None, None, None, None]
```

```
# Note the difference with R's NA
None == None
```

```
True
```

Aliasing vs Copying in Python

- Assignment binds the name on the left of = sign to the object on the right.
- But the same object can have different names (**aliases**).
- If the expression on the right is a name, the name on the left becomes an alias.
- In R (almost) all objects behave the same as the vast majority are immutable.
- In Python the object's type determines its behavior.
- Specifically, operations on immutable types overwrite the object if it gets modified.
- But for mutable types the object is modified in place.

Copying - Immutable Types

- Recall **copy-on-modify** semantics from R.
- Immutable types in Python behave similarly.

```
1 x = 'test' # Object of type string is assigned to variable `x`
2 id(x) # Function `id` returns the memory address of the object
```

```
127083386354736
```

```
1 y = x # `y` is created an alias (alternative name) of `x`
2 id(y)
```

```
127083386354736
```

```
1 x = 'rest' # Another object of type string is assigned to `x`
2 x
```

```
'rest'
```

```
1 id(x)
```

```
127083395212224
```

```
1 y
```

```
'test'
```

```
1 id(y)
```

```
127083386354736
```

Copying - Mutable Types

- Mutable types in Python, however, behave differently.
- Changing the object modifies it in place without copying.

```
1 d
```

```
{'apple': 150.0, 'banana': 120.0, 'watermelon': 3000.0, 'strawberry': 12.0}
```

```
1 d1 = d # Just an alias
2 d2 = d.copy() # Create a copy
```

```
1 d1['watermelon'] = 500 # Modify original dictionary
```

```
1 d1
```

```
{'apple': 150.0, 'banana': 120.0, 'watermelon': 500, 'strawberry': 12.0}
```

```
1 d
```

```
{'apple': 150.0, 'banana': 120.0, 'watermelon': 500, 'strawberry': 12.0}
```

```
1 d2
```

```
{'apple': 150.0, 'banana': 120.0, 'watermelon': 3000.0, 'strawberry': 12.0}
```

Summary of Built-in Object Types

Type	Description	Scalar	Mutability	Order
`int`	integer	scalar	immutable	
`float`	real number	scalar	immutable	
`bool`	Boolean	scalar	immutable	
`None`	Python 'Null'	scalar	immutable	
`str`	string	non-scalar	immutable	ordered
`tuple`	tuple	non-scalar	immutable	ordered
`list`	list	non-scalar	mutable	ordered
`set`	set	non-scalar	mutable	unordered
`dict`	dictionary	non-scalar	mutable	unordered

Extra

Python documentation on built-it types

Modules

- Python's power lies in its extensibility.
- This is usually achieved by loading additional modules (libraries).
- Module can be just a `.py` file that you import into your program (script).
- However, often this refers to external libraries installed using `pip` or `conda`.
- Standard Python installation also includes a number of modules (full list here).

Basic Statistical Operations

- Unlike R, Python does not have built-in support for statistical operations.

```
1 import statistics # Part of standard Python module
2 lst = [0, 1, 1, 2, 3, 5]
```

```
1 statistics.mean(lst) # Mean
```

```
2
```

```
1 statistics.median(lst) # Median
```

```
1.5
```

```
1 statistics.mode(lst) # Mode
```

```
1
```

```
1 statistics.stdev(lst) # Standard deviation
```

```
1.7888543819998317
```

Help!

Python has an inbuilt help facility which provides more information about any object:

```
1 ?s
```

```
invalid syntax (<string>, line 1)
```

```
1 help(s.join)
```

```
Help on built-in function join:

join(iterable, /) method of builtins.str instance
    Concatenate any number of strings.

    The string whose method is called is inserted in between each given string.
    The result is returned as a new string.

    Example: '.'.join(['ab', 'pq', 'rs']) -> 'ab.pq.rs'
```

- The quality of the documentation varies hugely across libraries
- Stackoverflow is a good resource for many standard tasks
- For custom packages it is often helpful to check the **issues** page on the GitHub
- E.g. for `pandas`: https://github.com/pandas-dev/pandas/issues
- Or, indeed, any search engine #LMDDGTFY

Next

- Tutorial: Python objects, types and basic operations
- Next week: Control flow and functions in Python