# Week 8: Fundamentals of Python Programming I

## POP77001 Computer Programming for Social Scientists

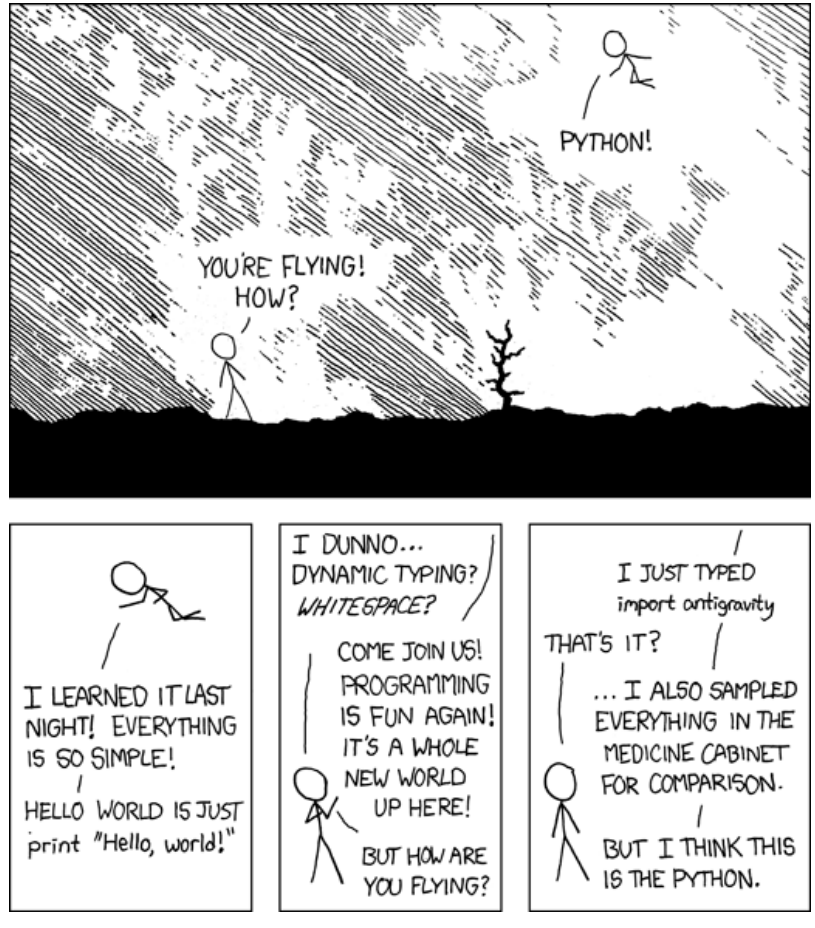
Tom Paskhalis

1 November 2022

Module website: tinyurl.com/POP77001

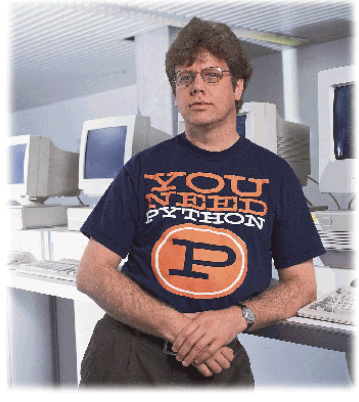
# Overview

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


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.

Source: xkcd

# Python background




Source: Guido van Rossum, Python Software Foundation

- 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

# The Zen of Python

In [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 Dut
ch.
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 ide
a.
```

# 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

# 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

In [2]:
```
print("Hello World!")
```

```
Hello World!
```

# 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.
- Eact object has a *type* (e.g. string or function) and internal data
- Objects can be *mutable* (e.g. list) and *immutable* (e.g. string)

# Operators

*Objects* and *operators* are combined to form *expressions*. Key *operators* are:

- Arithmetic ( `+` , `-` , `*` , `**` , `/` , `//` , `%` )
- Boolean ( `and` , `or` , `not` )
- Relational ( `==` , `!=` , `>` , `>=` , `<` , `<=` )
- Assignment ( `=` , `+=` , `-=` , `*=` , `/=` )
- Membership ( `in` )

# Basic mathematical operations in Python

# Basic mathematical operations in Python

In [3]:
```
1 + 1
```

Out[3]:
```
2
```

# Basic mathematical operations in Python

In [3]:
```
1 + 1
```

Out[3]: 2

In [4]:
```
5 - 3
```

Out[4]: 2

# Basic mathematical operations in Python

In [3]:
```
1 + 1
```

Out[3]: 2

In [4]:
```
5 - 3
```

Out[4]: 2

In [5]:
```
6 / 2
```

Out[5]: 3.0

# Basic mathematical operations in Python

In [3]:
```
1 + 1
```

Out[3]: 2

In [4]:
```
5 - 3
```

Out[4]: 2

In [5]:
```
6 / 2
```

Out[5]: 3.0

In [6]:
```
4 * 4
```

Out[6]: 16

# Basic mathematical operations in Python

In [3]:
```
1 + 1
```

Out[3]: 2

In [4]:
```
5 - 3
```

Out[4]: 2

In [5]:
```
6 / 2
```

Out[5]: 3.0

In [6]:
```
4 * 4
```

Out[6]: 16

In [7]:
```
# Exponentiation, Python comments start with #
2 ** 4
```

Out[7]: 16

# Advanced mathematical operations in Python

# Advanced mathematical operations in Python

In [8]:
```
# Integer division (remainder is discarded)
7 // 3
```

Out[8]:
```
2
```

# Advanced mathematical operations in Python

In [8]:
```
# Integer division (remainder is discarded)
7 // 3
```

Out[8]: 2

In [9]:
```
# Modulo operation (only remainder is retained)
7 % 3
```

Out[9]: 1

# Basic logical operations in Python

# Basic logical operations in Python

In [10]:
```
3 != 1 # Not equal
```

Out[10]:
```
True
```

# Basic logical operations in Python

In [10]:
```
3 != 1 # Not equal
```

Out[10]:
```
True
```

In [11]:
```
3 > 3 # Greater than
```

Out[11]:
```
False
```

# Basic logical operations in Python

In [10]:
```
3 != 1 # Not equal
```

Out[10]:
```
True
```

In [11]:
```
3 > 3 # Greater than
```

Out[11]:
```
False
```

In [12]:
```
3 >= 3 # Greater than or equal
```

Out[12]:
```
True
```

# Basic logical operations in Python

In [10]:
```
3 != 1 # Not equal
```

Out[10]: True

In [11]:
```
3 > 3 # Greater than
```

Out[11]: False

In [12]:
```
3 >= 3 # Greater than or equal
```

Out[12]: True

In [13]:
```
False or True # True if either first or second operand is True, False c
```

Out[13]: True

# Basic logical operations in Python

In [10]:
```
3 != 1 # Not equal
```

Out[10]: True

In [11]:
```
3 > 3 # Greater than
```

Out[11]: False

In [12]:
```
3 >= 3 # Greater than or equal
```

Out[12]: True

In [13]:
```
False or True # True if either first or second operand is True, False c
```

Out[13]: True

In [14]:
```
3 > 3 or 3 >= 3 # Combining 3 Boolean expressions
```

Out[14]: True

# Assignment operations

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

# Assignment operations

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

In [15]:
```
x = 3
```

# Assignment operations

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

In [15]:
```
x = 3
```

In [16]:
```
x
```

Out[16]:
```
3
```

# Assignment operations

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

In [15]:
```
x = 3
```

In [16]:
```
x
```

Out[16]:
```
3
```

In [17]:
```
x += 2 # Increment assignment, equivalent to x = x + 2
```

# Assignment operations

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

In [15]:
```
x = 3
```

In [16]:
```
x
```

Out[16]:
```
3
```

In [17]:
```
x += 2 # Increment assignment, equivalent to x = x + 2
```

In [18]:
```
x
```

Out[18]:
```
5
```

# Assignment vs Comparison Operators

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

# Assignment vs Comparison Operators

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

In [19]:
```
x = 3
```

# Assignment vs Comparison Operators

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

In [19]:
```
x = 3
```

In [20]:
```
x
```

Out[20]:
```
3
```

# Assignment vs Comparison Operators

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

In [19]:
```
x = 3
```

In [20]:
```
x
```

Out[20]:
```
3
```

In [21]:
```
x == 3
```

Out[21]:
```
True
```

# Membership operations

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

# Membership operations

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

In [22]:
```
'a' in 'abc'
```

Out[22]:
```
True
```

# Membership operations

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

In [22]:
```
'a' in 'abc'
```

Out[22]:
```
True
```

In [23]:
```
3 in [1, 2, 3] # [1,2,3] is a list
```

Out[23]:
```
True
```

# Membership operations

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

In [22]:
```
'a' in 'abc'
```

Out[22]:
```
True
```

In [23]:
```
3 in [1, 2, 3] # [1,2,3] is a list
```

Out[23]:
```
True
```

In [24]:
```
3 not in [1, 2, 3]
```

Out[24]:
```
False
```

# Object types

Python objects can have *scalar* and *non-scalar* types. Scalar objects are indivisible.

4 main types of scalar objects in Python:

- Integer ( `int` )
- Real number ( `float` )
- Boolean ( `bool` )
- Null value ( `None` )

# Scalar types

# Scalar types

In [25]:
```
type(7)
```

Out[25]:
```
int
```

# Scalar types

In [25]:
```
type(7)
```

Out[25]:
```
int
```

In [26]:
```
type(3.14)
```

Out[26]:
```
float
```

# Scalar types

In [25]:
```
type(7)
```

Out[25]:
```
int
```

In [26]:
```
type(3.14)
```

Out[26]:
```
float
```

In [27]:
```
type(True)
```

Out[27]:
```
bool
```

# Scalar types

In [25]:
```
type(7)
```

Out[25]:
```
int
```

In [26]:
```
type(3.14)
```

Out[26]:
```
float
```

In [27]:
```
type(True)
```

Out[27]:
```
bool
```

In [28]:
```
type(None) # None is the only object of NoneType
```

Out[28]:
```
NoneType
```

# Scalar types

In [25]:
```
type(7)
```

Out[25]:
```
int
```

In [26]:
```
type(3.14)
```

Out[26]:
```
float
```

In [27]:
```
type(True)
```

Out[27]:
```
bool
```

In [28]:
```
type(None) # None is the only object of NoneType
```

Out[28]:
```
NoneType
```

In [29]:
```
int(3.14) # Scalar type conversion (casting)
```

Out[29]:
```
3
```

# Non-scalar types

In contrast to scalars, non-scalar objects, *sequences*, have some internal structure. This allows indexing, slicing and other interesting operations.

Most common sequences in Python are:

- String ( `str` ) - *immutable* ordered sequence of 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

# Examples of non-scalar types

# Examples of non-scalar types

In [30]:
```
s = 'time flies like a banana'
t = (0, 'one', 1, 2)
l = [0, 'one', 1, 2]
o = {'apple', 'banana', 'watermelon'}
d = {'apple': 150.0, 'banana': 120.0, 'watermelon': 3000.0}
```

# Examples of non-scalar types

In [30]:
```
s = 'time flies like a banana'
t = (0, 'one', 1, 2)
l = [0, 'one', 1, 2]
o = {'apple', 'banana', 'watermelon'}
d = {'apple': 150.0, 'banana': 120.0, 'watermelon': 3000.0}
```

In [31]:
```
type(s)
```

Out[31]:
```
str
```

# Examples of non-scalar types

In [30]:
```
s = 'time flies like a banana'
t = (0, 'one', 1, 2)
l = [0, 'one', 1, 2]
o = {'apple', 'banana', 'watermelon'}
d = {'apple': 150.0, 'banana': 120.0, 'watermelon': 3000.0}
```

In [31]:
```
type(s)
```

Out[31]:
```
str
```

In [32]:
```
type(t)
```

Out[32]:
```
tuple
```

# Examples of non-scalar types

In [30]:
```
s = 'time flies like a banana'
t = (0, 'one', 1, 2)
l = [0, 'one', 1, 2]
o = {'apple', 'banana', 'watermelon'}
d = {'apple': 150.0, 'banana': 120.0, 'watermelon': 3000.0}
```

In [31]:
```
type(s)
```

Out[31]:
```
str
```

In [32]:
```
type(t)
```

Out[32]:
```
tuple
```

In [33]:
```
type(l)
```

Out[33]:
```
list
```

# Examples of non-scalar types

In [30]:
```
s = 'time flies like a banana'
t = (0, 'one', 1, 2)
l = [0, 'one', 1, 2]
o = {'apple', 'banana', 'watermelon'}
d = {'apple': 150.0, 'banana': 120.0, 'watermelon': 3000.0}
```

In [31]:
```
type(s)
```

Out[31]:
```
str
```

In [32]:
```
type(t)
```

Out[32]:
```
tuple
```

In [33]:
```
type(l)
```

Out[33]:
```
list
```

In [34]:
```
type(o)
```

Out[34]:
```
set
```

# Examples of non-scalar types

In [30]:
```
s = 'time flies like a banana'
t = (0, 'one', 1, 2)
l = [0, 'one', 1, 2]
o = {'apple', 'banana', 'watermelon'}
d = {'apple': 150.0, 'banana': 120.0, 'watermelon': 3000.0}
```

In [31]:
```
type(s)
```

Out[31]:
```
str
```

In [32]:
```
type(t)
```

Out[32]:
```
tuple
```

In [33]:
```
type(l)
```

Out[33]:
```
list
```

In [34]:
```
type(o)
```

Out[34]:
```
set
```

In [35]:
```
type(d)
```

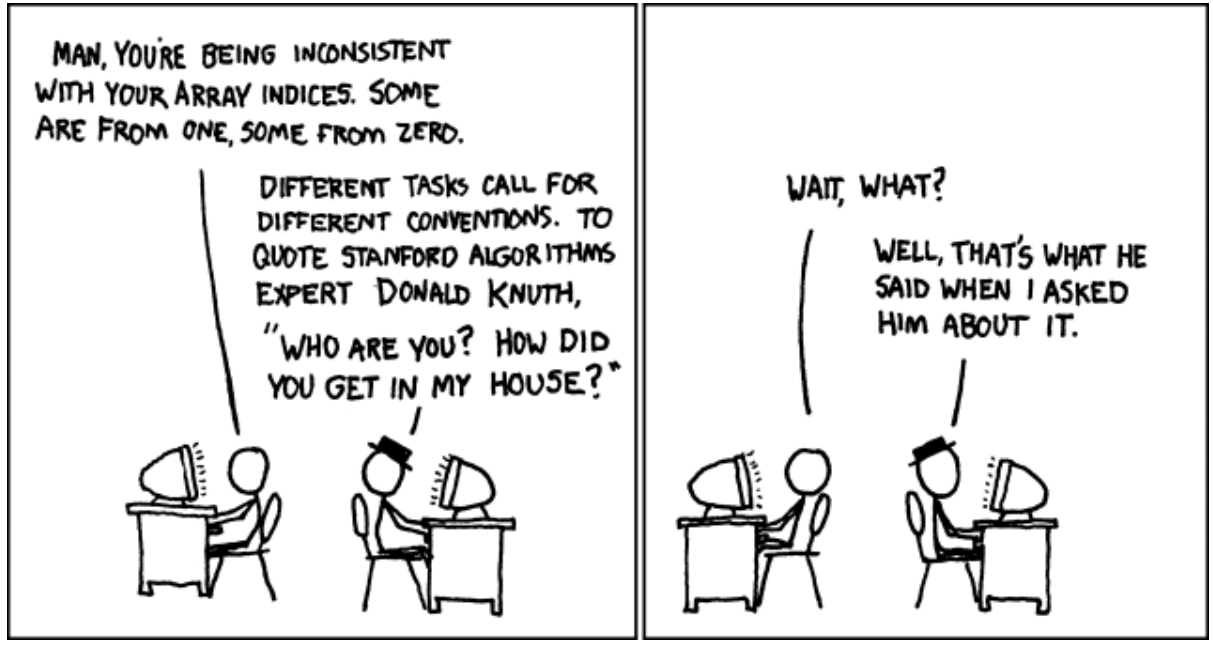
Out[35]: dict

# 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




Source: xkcd
Extra: Why Python uses 0-based indexing by Guido van Rossum
Extra: Why numbering should start at zero by Edsger Dijkstra

# Strings

# Strings

In [36]:
```
s
```

Out[36]:
```
'time flies like a banana'
```

# Strings

In [36]:
```
s
```

Out[36]:
```
'time flies like a banana'
```

In [37]:
```
len(s) # length of string (including whitespaces)
```

Out[37]:
```
24
```

# Strings

In [36]:
```
s
```

Out[36]:
```
'time flies like a banana'
```

In [37]:
```
len(s) # length of string (including whitespaces)
```

Out[37]:
```
24
```

In [38]:
```
s[0] # Subset 1st element (indexing in Python starts from zero!)
```

Out[38]:
```
't'
```

# Strings

In [36]:
```
s
```

Out[36]:
```
'time flies like a banana'
```

In [37]:
```
len(s) # length of string (including whitespaces)
```

Out[37]:
```
24
```

In [38]:
```
s[0] # Subset 1st element (indexing in Python starts from zero!)
```

Out[38]:
```
't'
```

In [39]:
```
s[5:] # Subset all elements starting from 6th
```

Out[39]:
```
'flies like a banana'
```

# Strings

In [36]:
```
s
```

Out[36]:
```
'time flies like a banana'
```

In [37]:
```
len(s) # length of string (including whitespaces)
```

Out[37]:
```
24
```

In [38]:
```
s[0] # Subset 1st element (indexing in Python starts from zero!)
```

Out[38]:
```
't'
```

In [39]:
```
s[5:] # Subset all elements starting from 6th
```

Out[39]:
```
'flies like a banana'
```

In [40]:
```
s + '!' # Strings can be concatenated together
```

Out[40]:
```
'time flies like a banana!'
```

# Objects have methods

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

# Objects have methods

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

In [41]:
```
len(s) # Function
```

Out[41]:
```
24
```

# Objects have methods

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

In [41]:
```
len(s) # Function
```

Out[41]:
```
24
```

In [42]:
```
s.upper() # Method (makes string upper-case)
```

Out[42]:
```
'TIME FLIES LIKE A BANANA'
```

# String methods

Some examples of methods associated with strings. More details here.

# String methods

Some examples of methods associated with strings. More details here.

In [43]:
```
s.capitalize() # Note that only the first character gets capitalized
```

Out[43]:
```
'Time flies like a banana'
```

# String methods

Some examples of methods associated with strings. More details here.

In [43]:
```
s.capitalize() # Note that only the first character gets capitalized
```

Out[43]:
```
'Time flies like a banana'
```

In [44]:
```
s.split(sep = ' ') # Here we supply an argument 'sep' to our methods ca
```

Out[44]:
```
['time', 'flies', 'like', 'a', 'banana']
```

# String methods

Some examples of methods associated with strings. More details here.

In [43]:
```
s.capitalize() # Note that only the first character gets capitalized
```

Out[43]:
```
'Time flies like a banana'
```

In [44]:
```
s.split(sep = ' ') # Here we supply an argument 'sep' to our methods ca
```

Out[44]:
```
['time', 'flies', 'like', 'a', 'banana']
```

In [45]:
```
s.replace(' ', '-') # Arguments can also be matched by position, not ju
```

Out[45]:
```
'time-flies-like-a-banana'
```

# String methods

Some examples of methods associated with strings. More details here.

In [43]:
```
s.capitalize() # Note that only the first character gets capitalized
```

Out[43]:
```
'Time flies like a banana'
```

In [44]:
```
s.split(sep = ' ') # Here we supply an argument 'sep' to our methods ca
```

Out[44]:
```
['time', 'flies', 'like', 'a', 'banana']
```

In [45]:
```
s.replace(' ', '-') # Arguments can also be matched by position, not ju
```

Out[45]:
```
'time-flies-like-a-banana'
```

In [46]:
```
'-'.join(s.split(sep = ' ')) # Methods calls can be nested within each
```

Out[46]:
```
'time-flies-like-a-banana'
```

# Tuples

# Tuples

In [47]:
```
t # Tuples can contain elements of different types
```

Out[47]:
```
(0, 'one', 1, 2)
```

# Tuples

In [47]:
```
t # Tuples can contain elements of different types
```

Out[47]:
```
(0, 'one', 1, 2)
```

In [48]:
```
len(t)
```

Out[48]:
```
4
```

# Tuples

In [47]:
```
t # Tuples can contain elements of different types
```

Out[47]:
```
(0, 'one', 1, 2)
```

In [48]:
```
len(t)
```

Out[48]:
```
4
```

In [49]:
```
t[1:]
```

Out[49]:
```
('one', 1, 2)
```

# Tuples

In [47]:
```
t # Tuples can contain elements of different types
```

Out[47]:
```
(0, 'one', 1, 2)
```

In [48]:
```
len(t)
```

Out[48]:
```
4
```

In [49]:
```
t[1:]
```

Out[49]:
```
('one', 1, 2)
```

In [50]:
```
t + ('three', 5) # Like strings tuples can be concatenated
```

Out[50]:
```
(0, 'one', 1, 2, 'three', 5)
```

# Lists

# Lists

In [51]:
```
l # Like tuples lists can contain elements of different types
```

Out[51]:
```
[0, 'one', 1, 2]
```

# Lists

In [51]:
```
l # Like tuples lists can contain elements of different types
```

Out[51]:
```
[0, 'one', 1, 2]
```

In [52]:
```
l[1] = 1 # Unlike tuples lists are mutable
```

# Lists

In [51]:
```
l # Like tuples lists can contain elements of different types
```

Out[51]:
```
[0, 'one', 1, 2]
```

In [52]:
```
l[1] = 1 # Unlike tuples lists are mutable
```

In [53]:
```
l
```

Out[53]:
```
[0, 1, 1, 2]
```

# Lists

In [51]:
```
l # Like tuples lists can contain elements of different types
```

Out[51]:
```
[0, 'one', 1, 2]
```

In [52]:
```
l[1] = 1 # Unlike tuples lists are mutable
```

In [53]:
```
l
```

Out[53]:
```
[0, 1, 1, 2]
```

In [54]:
```
t[1] = 1 # Compare to tuple
```

```
---------------------------------------------------------------------
------------
TypeError                                 Traceback (most recen
t call last)
<ipython-input-54-4e4114da061e> in <module>
----> 1 t[1] = 1 # Compare to tuple

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

# Indexing and slicing lists

# Indexing and slicing lists

In [55]:
```
l
```

Out[55]:
```
[0, 1, 1, 2]
```

# Indexing and slicing lists

In [55]:
```
l
```

Out[55]:
```
[0, 1, 1, 2]
```

In [56]:
```
l[1:] # Subset all elements starting from 2nd
```

Out[56]:
```
[1, 1, 2]
```

# Indexing and slicing lists

In [55]:
```
l
```

Out[55]:
```
[0, 1, 1, 2]
```

In [56]:
```
l[1:] # Subset all elements starting from 2nd
```

Out[56]:
```
[1, 1, 2]
```

In [57]:
```
l[-1] # Subset the last element
```

Out[57]:
```
2
```

# Indexing and slicing lists

In [55]:
```
l
```

Out[55]:
```
[0, 1, 1, 2]
```

In [56]:
```
l[1:] # Subset all elements starting from 2nd
```

Out[56]:
```
[1, 1, 2]
```

In [57]:
```
l[-1] # Subset the last element
```

Out[57]:
```
2
```

In [58]:
```
l[::2] # Subset every second element, list[start:stop:step]
```

Out[58]:
```
[0, 1]
```

# Indexing and slicing lists

In [55]:
```
l
```

Out[55]:
```
[0, 1, 1, 2]
```

In [56]:
```
l[1:] # Subset all elements starting from 2nd
```

Out[56]:
```
[1, 1, 2]
```

In [57]:
```
l[-1] # Subset the last element
```

Out[57]:
```
2
```

In [58]:
```
l[::2] # Subset every second element, list[start:stop:step]
```

Out[58]:
```
[0, 1]
```

In [59]:
```
l[::-1] # Subset all elements in reverse order
```

Out[59]:
```
[2, 1, 1, 0]
```

# Sets

# Sets

In [60]:
```
o
```

Out[60]:
```
{'apple', 'banana', 'watermelon'}
```

# Sets

In [60]:
```
o
```

Out[60]:
```
{'apple', 'banana', 'watermelon'}
```

In [61]:
```
{'apple', 'apple', 'banana', 'watermelon'} # Sets retain only unique va
```

Out[61]:
```
{'apple', 'banana', 'watermelon'}
```

# Sets

In [60]:
```
o
```

Out[60]:
```
{'apple', 'banana', 'watermelon'}
```

In [61]:
```
{'apple', 'apple', 'banana', 'watermelon'} # Sets retain only unique va
```

Out[61]:
```
{'apple', 'banana', 'watermelon'}
```

In [62]:
```
{'apple'} < o # Sets can be compared (e.g. one being subset of another)
```

Out[62]:
```
True
```

# Sets

In [60]:
```
o
```

Out[60]:
```
{'apple', 'banana', 'watermelon'}
```

In [61]:
```
{'apple', 'apple', 'banana', 'watermelon'} # Sets retain only unique va
```

Out[61]:
```
{'apple', 'banana', 'watermelon'}
```

In [62]:
```
{'apple'} < o # Sets can be compared (e.g. one being subset of another)
```

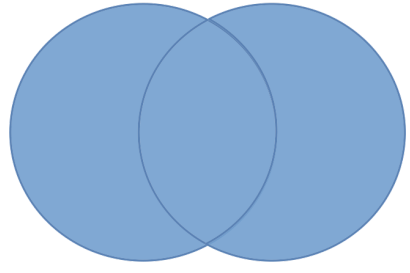
Out[62]:
```
True
```

In [63]:
```
o[1] # Unlike strings, tuples and lists, sets are unordered
```

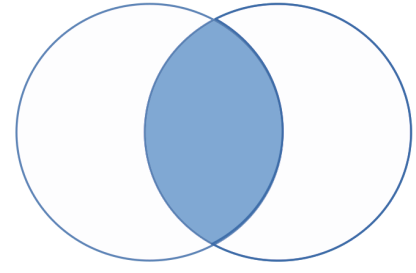
```
---------------------------------------------------------------------
------------
TypeError                                 Traceback (most recen
t call last)
<ipython-input-63-6a3d97725b65> in <module>
----> 1 o[1] # Unlike strings, tuples and lists, sets are unord
ered

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

# Set methods in Python

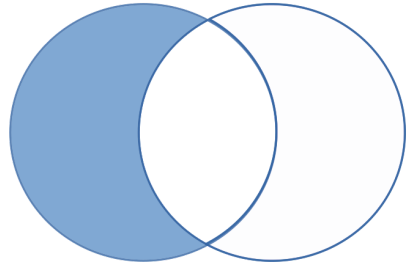
`union`



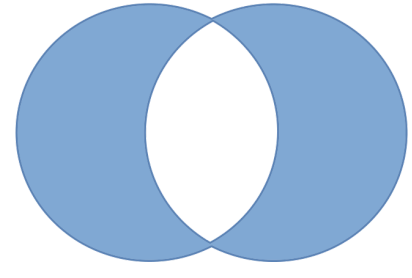
`intersection`



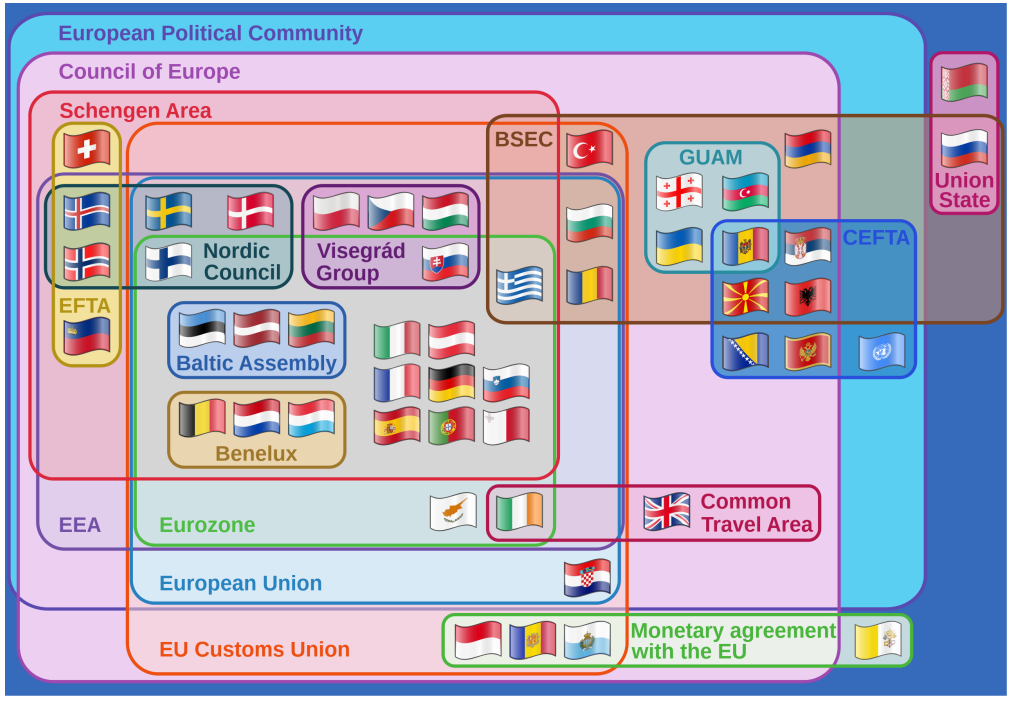
`difference`



`symmetric_difference`



# Set methods example




Source: Wikipedia

# Set methods example continued

# Set methods example continued

In [64]:
```
nordic = {'Denmark', 'Iceland', 'Finland', 'Norway', 'Sweden'}
eu = {'Denmark', 'Finland', 'Sweden'}
krones = {'Denmark', 'Sweden'}
```

# Set methods example continued

In [64]:
```
nordic = {'Denmark', 'Iceland', 'Finland', 'Norway', 'Sweden'}
eu = {'Denmark', 'Finland', 'Sweden'}
krones = {'Denmark', 'Sweden'}
```

In [65]:
```
euro = eu.difference(krones) # Same can expressed using infix operators
euro
```

Out[65]:
```
{'Finland'}
```

# Set methods example continued

In [64]:
```
nordic = {'Denmark', 'Iceland', 'Finland', 'Norway', 'Sweden'}
eu = {'Denmark', 'Finland', 'Sweden'}
krones = {'Denmark', 'Sweden'}
```

In [65]:
```
euro = eu.difference(krones) # Same can expressed using infix operators
euro
```

Out[65]:
```
{'Finland'}
```

In [66]:
```
efta = nordic.difference(eu).union({'Liechtenstein', 'Switzerland'}) #
efta
```

Out[66]:
```
{'Iceland', 'Liechtenstein', 'Norway', 'Switzerland'}
```

# Set methods example continued

In [64]:
```
nordic = {'Denmark', 'Iceland', 'Finland', 'Norway', 'Sweden'}
eu = {'Denmark', 'Finland', 'Sweden'}
krones = {'Denmark', 'Sweden'}
```

In [65]:
```
euro = eu.difference(krones) # Same can expressed using infix operators
euro
```

Out[65]:
```
{'Finland'}
```

In [66]:
```
efta = nordic.difference(eu).union({'Liechtenstein', 'Switzerland'}) #
efta
```

Out[66]:
```
{'Iceland', 'Liechtenstein', 'Norway', 'Switzerland'}
```

In [67]:
```
efta.intersection(nordic) # efta & nordic
```

Out[67]:
```
{'Iceland', 'Norway'}
```

# Set methods example continued

In [64]:
```
nordic = {'Denmark', 'Iceland', 'Finland', 'Norway', 'Sweden'}
eu = {'Denmark', 'Finland', 'Sweden'}
krones = {'Denmark', 'Sweden'}
```

In [65]:
```
euro = eu.difference(krones) # Same can expressed using infix operators
euro
```

Out[65]:
```
{'Finland'}
```

In [66]:
```
efta = nordic.difference(eu).union({'Liechtenstein', 'Switzerland'}) #
efta
```

Out[66]:
```
{'Iceland', 'Liechtenstein', 'Norway', 'Switzerland'}
```

In [67]:
```
efta.intersection(nordic) # efta & nordic
```

Out[67]:
```
{'Iceland', 'Norway'}
```

In [68]:
```
schengen = efta.union(eu) # efta | eu
schengen
```

Out[68]:
```
{'Denmark',
 'Finland',
 'Iceland',
```

```
'Liechtenstein',
'Norway',
'Sweden',
'Switzerland'}
```

# Dictionaries

# Dictionaries

In [69]:
```
d # key:value pair, fruit_name:average_weight
```

Out[69]:
```
{'apple': 150.0, 'banana': 120.0, 'watermelon': 3000.0}
```

# Dictionaries

In [69]:
```
d # key:value pair, fruit_name:average_weight
```

Out[69]:
```
{'apple': 150.0, 'banana': 120.0, 'watermelon': 3000.0}
```

In [70]:
```
d['apple'] # Unlike strings, tuples and lists, dictionaries are indexed
```

Out[70]:
```
150.0
```

# Dictionaries

In [69]:
```
d # key:value pair, fruit_name:average_weight
```

Out[69]:
```
{'apple': 150.0, 'banana': 120.0, 'watermelon': 3000.0}
```

In [70]:
```
d['apple'] # Unlike strings, tuples and lists, dictionaries are indexed
```

Out[70]:
```
150.0
```

In [71]:
```
d[0] # Rather than integers
```

```
---------------------------------------------------------------------------
KeyError                                  Traceback (most recent call last)
<ipython-input-71-3cd4cfa8b308> in <module>
----> 1 d[0] # Rather than integers

KeyError: 0
```

# Dictionaries

In [69]:
```
d # key:value pair, fruit_name:average_weight
```

Out[69]:
```
{'apple': 150.0, 'banana': 120.0, 'watermelon': 3000.0}
```

In [70]:
```
d['apple'] # Unlike strings, tuples and lists, dictionaries are indexed
```

Out[70]:
```
150.0
```

In [71]:
```
d[0] # Rather than integers
```

```
---------------------------------------------------------------------
------------
KeyError                                  Traceback (most recen
t call last)
<ipython-input-71-3cd4cfa8b308> in <module>
----> 1 d[0] # Rather than integers

KeyError: 0
```

In [72]:
```
d['strawberry'] = 12.0 # They are, however, mutable like lists and sets
d
```

Out[72]:
```
{'apple': 150.0, 'banana': 120.0, 'watermelon': 3000.0, 'strawb
erry': 12.0}
```

# Conversion between non-scalar types

# Conversion between non-scalar types

In [73]:
```
t ## Tuple
```

Out[73]:
```
(0, 'one', 1, 2)
```

# Conversion between non-scalar types

In [73]:
```
t ## Tuple
```

Out[73]:
```
(0, 'one', 1, 2)
```

In [74]:
```
list(t) ## Convert to list with a `list` function
```

Out[74]:
```
[0, 'one', 1, 2]
```

# Conversion between non-scalar types

In [73]:
```
t ## Tuple
```

Out[73]:
```
(0, 'one', 1, 2)
```

In [74]:
```
list(t) ## Convert to list with a `list` function
```

Out[74]:
```
[0, 'one', 1, 2]
```

In [75]:
```
[x for x in t] ## List comprehesion, [expr for elem in iterable if test
```

Out[75]:
```
[0, 'one', 1, 2]
```

# Conversion between non-scalar types

In [73]:
```
t ## Tuple
```

Out[73]:
```
(0, 'one', 1, 2)
```

In [74]:
```
list(t) ## Convert to list with a `list` function
```

Out[74]:
```
[0, 'one', 1, 2]
```

In [75]:
```
[x for x in t] ## List comprehesion, [expr for elem in iterable if test
```

Out[75]:
```
[0, 'one', 1, 2]
```

In [76]:
```
set([0, 1, 1, 2]) ## Conversion to set retains only unique values
```

Out[76]:
```
{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)

# 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)

In [77]:
```
# Initialization of some temporary variable, which can re-assigned to a
tmp = None
```

# 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)

In [77]:
```
# Initialization of some temporary variable, which can re-assigned to a
tmp = None
```

In [78]:
```
# Here we are initializing a list of length 10
tmp_l = [None] * 10
tmp_l
```

Out[78]:
```
[None, None, None, None, None, None, None, None, None, None]
```

# 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)

In [77]:
```
# Initialization of some temporary variable, which can re-assigned to a
tmp = None
```

In [78]:
```
# Here we are initializing a list of length 10
tmp_l = [None] * 10
tmp_l
```

Out[78]:
```
[None, None, None, None, None, None, None, None, None, None]
```

In [79]:
```
None == None
```

Out[79]:
```
True
```

# Aliasing vs copying in Python

- Assignment binds the varible name on the left of `=` sign to the object of certain type on the right.
- But the same object can have different names.
- Operations on immutable types typically overwrite the object if it gets modified.
- But for mutable objects (lists, sets, dictionaries) this can create hard-to-track problems.

# Example of aliasing/copying for immutable types

# Example of aliasing/copying for immutable types

In [80]:
```
x = 'test' # Object of type string is assinged to variable 'x'
x
```

Out[80]:
```
'test'
```

# Example of aliasing/copying for immutable types

In [80]:
```
x = 'test' # Object of type string is assinged to variable 'x'
x
```

Out[80]:
```
'test'
```

In [81]:
```
y = x # y is created an alias (alternative name) of x
y
```

Out[81]:
```
'test'
```

# Example of aliasing/copying for immutable types

In [80]:
```
x = 'test' # Object of type string is assinged to variable 'x'
x
```

Out[80]:
```
'test'
```

In [81]:
```
y = x # y is created an alias (alternative name) of x
y
```

Out[81]:
```
'test'
```

In [82]:
```
x = 'rest' # Another object of type string is assigned to 'x'
x
```

Out[82]:
```
'rest'
```

# Example of aliasing/copying for immutable types

In [80]:
```
x = 'test' # Object of type string is assinged to variable 'x'
x
```

Out[80]:
```
'test'
```

In [81]:
```
y = x # y is created an alias (alternative name) of x
y
```

Out[81]:
```
'test'
```

In [82]:
```
x = 'rest' # Another object of type string is assigned to 'x'
x
```

Out[82]:
```
'rest'
```

In [83]:
```
y
```

Out[83]:
```
'test'
```

# Example of aliasing/copying for mutable types

# Example of aliasing/copying for mutable types

In [84]:
```
d
```

Out[84]:
```
{'apple': 150.0, 'banana': 120.0, 'watermelon': 3000.0, 'strawb
erry': 12.0}
```

# Example of aliasing/copying for mutable types

In [84]:
```
d
```

Out[84]:
```
{'apple': 150.0, 'banana': 120.0, 'watermelon': 3000.0, 'strawb
erry': 12.0}
```

In [85]:
```
d1 = d # Just an alias
d2 = d.copy() # Create a copy
d['watermelon'] = 500 # Modify original dictionary
```

# Example of aliasing/copying for mutable types

In [84]:
```
d
```

Out[84]:
```
{'apple': 150.0, 'banana': 120.0, 'watermelon': 3000.0, 'strawb
erry': 12.0}
```

In [85]:
```
d1 = d # Just an alias
d2 = d.copy() # Create a copy
d['watermelon'] = 500 # Modify original dictionary
```

In [86]:
```
d1
```

Out[86]:
```
{'apple': 150.0, 'banana': 120.0, 'watermelon': 500, 'strawberr
y': 12.0}
```

# Example of aliasing/copying for mutable types

In [84]:
```
d
```

Out[84]:
```
{'apple': 150.0, 'banana': 120.0, 'watermelon': 3000.0, 'strawb
erry': 12.0}
```

In [85]:
```
d1 = d # Just an alias
d2 = d.copy() # Create a copy
d['watermelon'] = 500 # Modify original dictionary
```

In [86]:
```
d1
```

Out[86]:
```
{'apple': 150.0, 'banana': 120.0, 'watermelon': 500, 'strawberr
y': 12.0}
```

In [87]:
```
d2
```

Out[87]:
```
{'apple': 150.0, 'banana': 120.0, 'watermelon': 3000.0, 'strawb
erry': 12.0}
```

# Summary of built-in object types in Python

| 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: Extensive 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

# Basic statistical operations

In [88]:
```
import statistics # Standard Python module
fib = [0, 1, 1, 2, 3, 5]
```

# Basic statistical operations

In [88]:
```
import statistics # Standard Python module
fib = [0, 1, 1, 2, 3, 5]
```

In [89]:
```
statistics.mean(fib) # Mean
```

Out[89]: 2

# Basic statistical operations

In [88]:
```
import statistics # Standard Python module
fib = [0, 1, 1, 2, 3, 5]
```

In [89]:
```
statistics.mean(fib) # Mean
```

Out[89]: 2

In [90]:
```
statistics.median(fib) # Median
```

Out[90]: 1.5

# Basic statistical operations

In [88]:
```
import statistics # Standard Python module
fib = [0, 1, 1, 2, 3, 5]
```

In [89]:
```
statistics.mean(fib) # Mean
```

Out[89]: 2

In [90]:
```
statistics.median(fib) # Median
```

Out[90]: 1.5

In [91]:
```
statistics.mode(fib) # Mode
```

Out[91]: 1

# Basic statistical operations

In [88]:
```
import statistics # Standard Python module
fib = [0, 1, 1, 2, 3, 5]
```

In [89]:
```
statistics.mean(fib) # Mean
```

Out[89]: 2

In [90]:
```
statistics.median(fib) # Median
```

Out[90]: 1.5

In [91]:
```
statistics.mode(fib) # Mode
```

Out[91]: 1

In [92]:
```
statistics.stdev(fib) # Standard deviation
```

Out[92]: 1.7888543819998317

# Help!

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

# Help!

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

In [93]:
```
?s
```

# Help!

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

In [93]:
```
?s
```

In [94]:
```
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 ea
ch given string.
    The result is returned as a new string.

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

# Help!

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

In [93]:
```
?s
```

In [94]:
```
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 ea
ch 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, basic operations and methods
- Next week: Control flow and functions in Python