Introduction to Python

Tom Paskhalis

RECSM Summer School 2021, Python Basics, Part 2, Day 1

Python basics

- Python is an *intepreted* 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 *intepreted* 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 [1]:
```
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 [2]: 1 + 1
```

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

Basic mathematical operations in Python

In [2]:
```
1 + 1
```

Out[2]: 2

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

Out[3]: 2

Basic mathematical operations in Python

In [2]:
```
1 + 1
```

Out[2]: 2

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

Out[3]: 2

In [4]:
```
6 / 2
```

Out[4]: 3.0

Basic mathematical operations in Python

```
In [2]: 1 + 1
```

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

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

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

```
In [4]: 6 / 2
```

```
Out[4]: 3.0
```

```
In [5]: 4 * 4
```

```
Out[5]: 16
```

Basic mathematical operations in Python

In [2]:
```
1 + 1
```

Out[2]: 2

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

Out[3]: 2

In [4]:
```
6 / 2
```

Out[4]: 3.0

In [5]:
```
4 * 4
```

Out[5]: 16

In [6]:
```
# Exponentiation <- Python comments start with #
2 ** 4
```

Basic logical operations in Python

Basic logical operations in Python

In [7]:
```
3 != 1 # Not equal
```

Out[7]: True

Basic logical operations in Python

In [7]:
```
3 != 1 # Not equal
```

Out[7]: True

In [8]:
```
3 > 3 # Greater than
```

Out[8]: False

Basic logical operations in Python

In [7]:
```
3 != 1 # Not equal
```

Out[7]: True

In [8]:
```
3 > 3 # Greater than
```

Out[8]: False

In [9]:
```
3 >= 3 # Greater than or equal
```

Out[9]: True

Basic logical operations in Python

In [7]:
```
3 != 1 # Not equal
```

Out[7]: True

In [8]:
```
3 > 3 # Greater than
```

Out[8]: False

In [9]:
```
3 >= 3 # Greater than or equal
```

Out[9]: True

In [10]:
```
False or True # True if either first or second operand is True, False otherwise
```

Out[10]: True

Basic logical operations in Python

In [7]:
```
3 != 1 # Not equal
```

Out[7]: True

In [8]:
```
3 > 3 # Greater than
```

Out[8]: False

In [9]:
```
3 >= 3 # Greater than or equal
```

Out[9]: True

In [10]:
```
False or True # True if either first or second operand is True, False otherwise
```

Out[10]: True

In [11]:
```
3 > 3 or 3 >= 3 # Combining 3 Boolean expressions
```

Out[11]: True

Assignment operations

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

Assignment operations

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

In [12]:
```
x = 3
```

Assignment operations

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

```
In [12]: x = 3
```

```
In [13]: x
```

```
Out[13]: 3
```

Assignment operations

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

In [12]:
```
x = 3
```

In [13]:
```
x
```

Out[13]: 3

In [14]:
```
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 rigth.

In [12]:
```
x = 3
```

In [13]:
```
x
```

Out[13]: 3

In [14]:
```
x += 2 # Increment assignment, equivalent to x = x + 2
```

In [15]:
```
x
```

Out[15]: 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 [16]:
```
x = 3
```

Assignment vs Comparison Operators

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

```
In [16]: x = 3
```

```
In [17]: x
```

```
Out[17]: 3
```

Assignment vs Comparison Operators

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

```
In [16]: x = 3

In [17]: x

Out[17]: 3

In [18]: x == 3

Out[18]: 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 [19]: 'a' in 'abc'
```

```
Out[19]: True
```

Membership operations

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

In [19]:
```
'a' in 'abc'
```

Out[19]:
```
True
```

In [20]:
```
4 in [1, 2, 3] # [1,2,3] is a list
```

Out[20]:
```
False
```

Membership operations

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

```
In [19]: 'a' in 'abc'
```

```
Out[19]: True
```

```
In [20]: 4 in [1, 2, 3] # [1,2,3] is a list
```

```
Out[20]: False
```

```
In [21]: 4 not in [1, 2, 3]
```

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

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 [22]:
```
type(7)
```

Out[22]:
```
int
```

Scalar types

In [22]:
```
type(7)
```

Out[22]:
```
int
```

In [23]:
```
type(3.14)
```

Out[23]:
```
float
```

Scalar types

In [22]:
```
type(7)
```

Out[22]:
```
int
```

In [23]:
```
type(3.14)
```

Out[23]:
```
float
```

In [24]:
```
type(True)
```

Out[24]:
```
bool
```

Scalar types

In [22]:
```
type(7)
```

Out[22]: int

In [23]:
```
type(3.14)
```

Out[23]: float

In [24]:
```
type(True)
```

Out[24]: bool

In [25]:
```
type(None)
```

Out[25]: NoneType

Scalar types

In [22]:
```
type(7)
```

Out[22]: int

In [23]:
```
type(3.14)
```

Out[23]: float

In [24]:
```
type(True)
```

Out[24]: bool

In [25]:
```
type(None)
```

Out[25]: NoneType

In [26]:
```
int(3.14) # Scalar type conversion (casting)
```

Out[26]: 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:

Examples of non-scalar types

Examples of non-scalar types

In [27]:
```
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 [27]:
```
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 [28]:
```
type(s)
```

Out[28]:
```
str
```

Examples of non-scalar types

In [27]:
```
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 [28]:
```
type(s)
```

Out[28]: str

In [29]:
```
type(t)
```

Out[29]: tuple

Examples of non-scalar types

In [27]:
```
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 [28]:
```
type(s)
```

Out[28]: str

In [29]:
```
type(t)
```

Out[29]: tuple

In [30]:
```
type(l)
```

Out[30]: list

Examples of non-scalar types

In [27]:
```
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 [28]:
```
type(s)
```

Out[28]: str

In [29]:
```
type(t)
```

Out[29]: tuple

In [30]:
```
type(l)
```

Out[30]: list

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

Examples of non-scalar types

In [27]:
```
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 [28]:
```
type(s)
```

Out[28]: str

In [29]:
```
type(t)
```

Out[29]: tuple

In [30]:
```
type(l)
```

Out[30]: list

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

Strings

Strings

In [33]:
```
s
```

Out[33]:
```
'time flies like a banana'
```

Strings

In [33]:
```
s
```

Out[33]:
```
'time flies like a banana'
```

In [34]:
```
len(s) # length of string (including whitespaces)
```

Out[34]:
```
24
```

Strings

In [33]:
```
s
```

Out[33]:
```
'time flies like a banana'
```

In [34]:
```
len(s) # length of string (including whitespaces)
```

Out[34]:
```
24
```

In [35]:
```
s[0] # Subset 1st element (indexing in Python starts from zero!)
```

Out[35]:
```
't'
```

Strings

In [33]:
```
s
```

Out[33]:
```
'time flies like a banana'
```

In [34]:
```
len(s) # length of string (including whitespaces)
```

Out[34]:
```
24
```

In [35]:
```
s[0] # Subset 1st element (indexing in Python starts from zero!)
```

Out[35]:
```
't'
```

In [36]:
```
s[5:] # Subset all elements starting from 6th
```

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

Strings

In [33]:
```
s
```

Out[33]: 'time flies like a banana'

In [34]:
```
len(s) # length of string (including whitespaces)
```

Out[34]: 24

In [35]:
```
s[0] # Subset 1st element (indexing in Python starts from zero!)
```

Out[35]: 't'

In [36]:
```
s[5:] # Subset all elements starting from 6th
```

Out[36]: 'flies like a banana'

In [37]:
```
s + '!' # Strings can be concatenated together
```

Out[37]: '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 [38]: len(s) # Function

Out[38]: 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 [38]: len(s) # Function
```

```
Out[38]: 24
```

```
In [39]: s.upper() # Method (makes string upper-case)
```

```
Out[39]: '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 [40]:
```
s.capitalize() # Note that only the first character gets capitalized
```

Out[40]:
```
'Time flies like a banana'
```

String methods

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

In [40]:
```
s.capitalize() # Note that only the first character gets capitalized
```

Out[40]:
```
'Time flies like a banana'
```

In [41]:
```
s.split(sep = ' ') # Here we supply an argument 'sep' to our methods call
```

Out[41]:
```
['time', 'flies', 'like', 'a', 'banana']
```

String methods

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

In [40]:
```
s.capitalize() # Note that only the first character gets capitalized
```

Out[40]:
```
'Time flies like a banana'
```

In [41]:
```
s.split(sep = ' ') # Here we supply an argument 'sep' to our methods call
```

Out[41]:
```
['time', 'flies', 'like', 'a', 'banana']
```

In [42]:
```
s.replace(' ', '-') # Arguments can also be matched by position, not just name
```

Out[42]:
```
'time-flies-like-a-banana'
```

String methods

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

In [40]:
```
s.capitalize() # Note that only the first character gets capitalized
```

Out[40]:
```
'Time flies like a banana'
```

In [41]:
```
s.split(sep = ' ') # Here we supply an argument 'sep' to our methods call
```

Out[41]:
```
['time', 'flies', 'like', 'a', 'banana']
```

In [42]:
```
s.replace(' ', '-') # Arguments can also be matched by position, not just name
```

Out[42]:
```
'time-flies-like-a-banana'
```

In [43]:
```
'-'.join(s.split(sep = ' ')) # Methods calls can be nested within each other
```

Out[43]:
```
'time-flies-like-a-banana'
```

Tuples

Tuples

In [44]:
```
t # Tuples can contain elements of different types
```

Out[44]:
```
(0, 'one', 1, 2)
```

Tuples

In [44]:
```
t # Tuples can contain elements of different types
```

Out[44]:
```
(0, 'one', 1, 2)
```

In [45]:
```
len(t)
```

Out[45]:
```
4
```

Tuples

In [44]:
```
t # Tuples can contain elements of different types
```

Out[44]:
```
(0, 'one', 1, 2)
```

In [45]:
```
len(t)
```

Out[45]:
```
4
```

In [46]:
```
t[1:]
```

Out[46]:
```
('one', 1, 2)
```

Tuples

In [44]:
```
t # Tuples can contain elements of different types
```

Out[44]:
```
(0, 'one', 1, 2)
```

In [45]:
```
len(t)
```

Out[45]:
```
4
```

In [46]:
```
t[1:]
```

Out[46]:
```
('one', 1, 2)
```

In [47]:
```
t + ('three', 5) # Like strings tuples can be concatenated
```

Out[47]:
```
(0, 'one', 1, 2, 'three', 5)
```

Lists

Lists

In [48]:
```
l # Like tuples lists can contain elements of different types
```

Out[48]:
```
[0, 'one', 1, 2]
```

Lists

In [48]:
```
l # Like tuples lists can contain elements of different types
```

Out[48]:
```
[0, 'one', 1, 2]
```

In [49]:
```
l[1] = 1 # Unlike tuples lists are mutable
```

Lists

In [48]:
```
l # Like tuples lists can contain elements of different types
```

Out[48]:
```
[0, 'one', 1, 2]
```

In [49]:
```
l[1] = 1 # Unlike tuples lists are mutable
```

In [50]:
```
l
```

Out[50]:
```
[0, 1, 1, 2]
```

Lists

In [48]:
```
l # Like tuples lists can contain elements of different types
```

Out[48]:
```
[0, 'one', 1, 2]
```

In [49]:
```
l[1] = 1 # Unlike tuples lists are mutable
```

In [50]:
```
l
```

Out[50]:
```
[0, 1, 1, 2]
```

In [51]:
```
t[1] = 1 # Compare to tuple
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-51-4e4114da061e> in <module>
----> 1 t[1] = 1 # Compare to tuple

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

More on subsetting

More on subsetting

```
In [52]: l
```

```
Out[52]: [0, 1, 1, 2]
```

More on subsetting

In [52]:
```
l
```

Out[52]: [0, 1, 1, 2]

In [53]:
```
l[1:] # Subset all elements starting from 2nd
```

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

More on subsetting

In [52]:
```
l
```

Out[52]: [0, 1, 1, 2]

In [53]:
```
l[1:] # Subset all elements starting from 2nd
```

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

In [54]:
```
l[-1] # Subset the last element
```

Out[54]: 2

More on subsetting

In [52]:
```
l
```

Out[52]: [0, 1, 1, 2]

In [53]:
```
l[1:] # Subset all elements starting from 2nd
```

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

In [54]:
```
l[-1] # Subset the last element
```

Out[54]: 2

In [55]:
```
l[::2] # Subset every second element, list[start:stop:step]
```

Out[55]: [0, 1]

More on subsetting

In [52]:
```
l
```

Out[52]: [0, 1, 1, 2]

In [53]:
```
l[1:] # Subset all elements starting from 2nd
```

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

In [54]:
```
l[-1] # Subset the last element
```

Out[54]: 2

In [55]:
```
l[::2] # Subset every second element, list[start:stop:step]
```

Out[55]: [0, 1]

In [56]:
```
l[::-1] # Subset all elements in reverse order
```

Out[56]: [2, 1, 1, 0]

Sets

Sets

In [57]:
```
o
```

Out[57]:
```
{'apple', 'banana', 'watermelon'}
```

Sets

In [57]:
```
o
```

Out[57]:
```
{'apple', 'banana', 'watermelon'}
```

In [58]:
```
{'apple', 'apple', 'banana', 'watermelon'} # Sets retain only unique values
```

Out[58]:
```
{'apple', 'banana', 'watermelon'}
```

Sets

In [57]:
```
o
```

Out[57]:
```
{'apple', 'banana', 'watermelon'}
```

In [58]:
```
{'apple', 'apple', 'banana', 'watermelon'} # Sets retain only unique values
```

Out[58]:
```
{'apple', 'banana', 'watermelon'}
```

In [59]:
```
{'apple'} < o # Sets can be compared (e.g. one being subset of another)
```

Out[59]:
```
True
```

Sets

In [57]:
```
o
```

Out[57]:
```
{'apple', 'banana', 'watermelon'}
```

In [58]:
```
{'apple', 'apple', 'banana', 'watermelon'} # Sets retain only unique values
```

Out[58]:
```
{'apple', 'banana', 'watermelon'}
```

In [59]:
```
{'apple'} < o # Sets can be compared (e.g. one being subset of another)
```

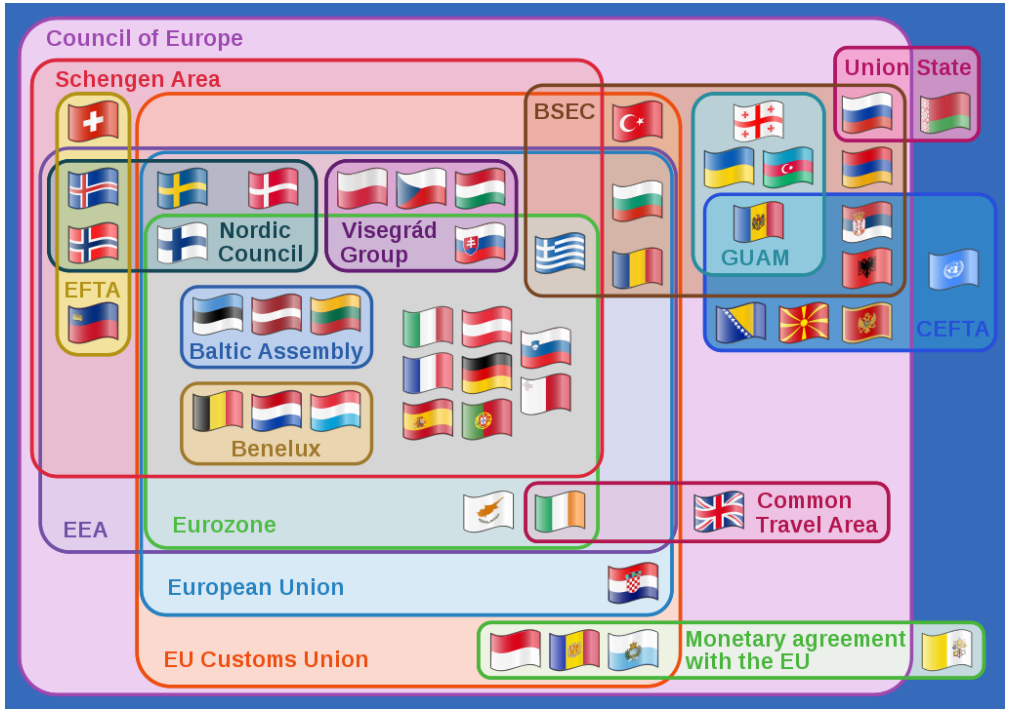
Out[59]:
```
True
```

In [60]:
```
o[1] # Unlike strings, tuples and lists, sets are unordered
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-60-6a3d97725b65> in <module>
----> 1 o[1] # Unlike strings, tuples and lists, sets are unordered

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

Set methods



Source: Wikipedia

Set methods

Set methods

In [61]:
```
nordic = {'Denmark', 'Iceland', 'Finland', 'Norway', 'Sweden'}
eu = {'Denmark', 'Finland', 'Sweden'}
krones = {'Denmark', 'Sweden'}
```

Set methods

In [61]:
```
nordic = {'Denmark', 'Iceland', 'Finland', 'Norway', 'Sweden'}
eu = {'Denmark', 'Finland', 'Sweden'}
krones = {'Denmark', 'Sweden'}
```

In [62]:
```
euro = eu.difference(krones) # Same can expressed using infix operators `eu - krone
euro
```

Out[62]:
```
{'Finland'}
```

Set methods

In [61]:
```
nordic = {'Denmark', 'Iceland', 'Finland', 'Norway', 'Sweden'}
eu = {'Denmark', 'Finland', 'Sweden'}
krones = {'Denmark', 'Sweden'}
```

In [62]:
```
euro = eu.difference(krones) # Same can expressed using infix operators `eu - krone
euro
```

Out[62]:
```
{'Finland'}
```

In [63]:
```
efta = nordic.difference(eu).union({'Liechtenstein', 'Switzerland'}) # Methods call
efta
```

Out[63]:
```
{'Iceland', 'Liechtenstein', 'Norway', 'Switzerland'}
```

Set methods

In [61]:
```
nordic = {'Denmark', 'Iceland', 'Finland', 'Norway', 'Sweden'}
eu = {'Denmark', 'Finland', 'Sweden'}
krones = {'Denmark', 'Sweden'}
```

In [62]:
```
euro = eu.difference(krones) # Same can expressed using infix operators `eu - krone
euro
```

Out[62]:
```
{'Finland'}
```

In [63]:
```
efta = nordic.difference(eu).union({'Liechtenstein', 'Switzerland'}) # Methods call
efta
```

Out[63]:
```
{'Iceland', 'Liechtenstein', 'Norway', 'Switzerland'}
```

In [64]:
```
efta.intersection(nordic) # efta & nordic
```

Out[64]:
```
{'Iceland', 'Norway'}
```

Set methods

In [61]:
```
nordic = {'Denmark', 'Iceland', 'Finland', 'Norway', 'Sweden'}
eu = {'Denmark', 'Finland', 'Sweden'}
krones = {'Denmark', 'Sweden'}
```

In [62]:
```
euro = eu.difference(krones) # Same can expressed using infix operators `eu - krone
euro
```

Out[62]: {'Finland'}

In [63]:
```
efta = nordic.difference(eu).union({'Liechtenstein', 'Switzerland'}) # Methods call
efta
```

Out[63]: {'Iceland', 'Liechtenstein', 'Norway', 'Switzerland'}

In [64]:
```
efta.intersection(nordic) # efta & nordic
```

Out[64]: {'Iceland', 'Norway'}

In [65]:
```
schengen = efta.union(eu) # efta | eu
```

Dictionaries

Dictionaries

In [66]:
```
d
```

Out[66]:
```
{'apple': 150.0, 'banana': 120.0, 'watermelon': 3000.0}
```

Dictionaries

In [66]:
```
d
```

Out[66]:
```
{'apple': 150.0, 'banana': 120.0, 'watermelon': 3000.0}
```

In [67]:
```
d['apple'] # Unlike strings, tuples and lists, dictionaries are indexed by 'keys'
```

Out[67]:
```
150.0
```

Dictionaries

In [66]:
```
d
```

Out[66]:
```
{'apple': 150.0, 'banana': 120.0, 'watermelon': 3000.0}
```

In [67]:
```
d['apple'] # Unlike strings, tuples and lists, dictionaries are indexed by 'keys'
```

Out[67]:
```
150.0
```

In [68]:
```
d[0] # Rather than integers
```

```
---------------------------------------------------------------------------
KeyError                                  Traceback (most recent call last)
<ipython-input-68-3cd4cfa8b308> in <module>
----> 1 d[0] # Rather than integers

KeyError: 0
```

Dictionaries

In [66]:
```
d
```

Out[66]:
```
{'apple': 150.0, 'banana': 120.0, 'watermelon': 3000.0}
```

In [67]:
```
d['apple'] # Unlike strings, tuples and lists, dictionaries are indexed by 'keys'
```

Out[67]:
```
150.0
```

In [68]:
```
d[0] # Rather than integers
```

```
---------------------------------------------------------------------------
KeyError                                  Traceback (most recent call last)
<ipython-input-68-3cd4cfa8b308> in <module>
----> 1 d[0] # Rather than integers

KeyError: 0
```

In [69]:
```
d['strawberry'] = 12.0 # They are, however, mutable like lists and sets
d
```

Conversion between non-scalar types

Conversion between non-scalar types

In [70]:
```
t ## Tuple
```

Out[70]:
```
(0, 'one', 1, 2)
```

Conversion between non-scalar types

In [70]:
```
t ## Tuple
```

Out[70]:
```
(0, 'one', 1, 2)
```

In [71]:
```
list(t) ## Convert to list with a `list` function
```

Out[71]:
```
[0, 'one', 1, 2]
```

Conversion between non-scalar types

In [70]:
```
t ## Tuple
```

Out[70]:
```
(0, 'one', 1, 2)
```

In [71]:
```
list(t) ## Convert to list with a `list` function
```

Out[71]:
```
[0, 'one', 1, 2]
```

In [72]:
```
[x for x in t] ## List comprehesion, [expr for elem in iterable if test]
```

Out[72]:
```
[0, 'one', 1, 2]
```

Conversion between non-scalar types

In [70]:
```
t ## Tuple
```

Out[70]:
```
(0, 'one', 1, 2)
```

In [71]:
```
list(t) ## Convert to list with a `list` function
```

Out[71]:
```
[0, 'one', 1, 2]
```

In [72]:
```
[x for x in t] ## List comprehesion, [expr for elem in iterable if test]
```

Out[72]:
```
[0, 'one', 1, 2]
```

In [73]:
```
set([0, 1, 1, 2]) ## Conversion to set retains only unique values
```

Out[73]:
```
{0, 1, 2}
```

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

Extensive 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 [74]:
```
import statistics # Standard Python module
fib = [0, 1, 1, 2, 3, 5]
```

Basic statistical operations

In [74]:
```
import statistics # Standard Python module
fib = [0, 1, 1, 2, 3, 5]
```

In [75]:
```
statistics.mean(fib) # Mean
```

Out[75]: 2

Basic statistical operations

In [74]:
```
import statistics # Standard Python module
fib = [0, 1, 1, 2, 3, 5]
```

In [75]:
```
statistics.mean(fib) # Mean
```

Out[75]: 2

In [76]:
```
statistics.median(fib) # Median
```

Out[76]: 1.5

Basic statistical operations

In [74]:
```
import statistics # Standard Python module
fib = [0, 1, 1, 2, 3, 5]
```

In [75]:
```
statistics.mean(fib) # Mean
```

Out[75]: 2

In [76]:
```
statistics.median(fib) # Median
```

Out[76]: 1.5

In [77]:
```
statistics.mode(fib) # Mode
```

Out[77]: 1

Basic statistical operations

In [74]:
```
import statistics # Standard Python module
fib = [0, 1, 1, 2, 3, 5]
```

In [75]:
```
statistics.mean(fib) # Mean
```

Out[75]: 2

In [76]:
```
statistics.median(fib) # Median
```

Out[76]: 1.5

In [77]:
```
statistics.mode(fib) # Mode
```

Out[77]: 1

In [78]:
```
statistics.stdev(fib) # Standard deviation
```

Out[78]: 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 [79]: ?s
```

Help!

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

In [79]:
```
?s
```

In [80]:
```
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'
```

Help!

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

In [79]:
```
?s
```

In [80]:
```
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'
```

Next

- Pandas
- Data I/O