Sequence data types: String, List, Tuple

Sequence data types are non-scalar objects, which means they can be subdivided.

They are an ordered collection of items (elements), and each element is referenced by an index

-5	-4	-3	-2	-1
elem1	**elem2**	**elem3**	**elem4**	**elem5**
0	1	2	3	4

```
s = 'Hello World'  # string elements are Unicode characters

L = ['pop',46,[1,4,5]]  # list elements can be of any data type

T = ('pop',46,[1,4,5])  # tuple elements can be of any data type
```

Access one element of a sequence - Indexing (String)


```
seq[index] #access one element referenced by one index
```

We can use positive indexes, which start from 0.
Or we can use negative indexes, which start for -1.

-11	-10	-9	-8	-7	-6	-5	-4	-3	-2	-1
H	e	l	l	o		W	o	r	l	d
0	1	2	3	4	5	6	7	8	9	10

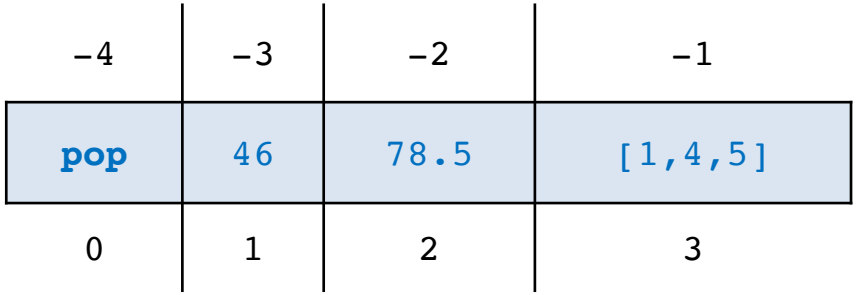
```
s='Hello World'
s[-7]  # 'o'

s[-1]  # 'd'
type(s[-1]) # <class 'str'>

c=s[0]
print(c)
H

s[14] #out of range error
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: string index out of range
```

Access one element of a sequence - Indexing (List)

A list is a container of objects

```
seq[index] # access one element referenced by one index
```

```
L = ['pop',46,78.5, [1,4,5]]
```




```
L[1]        # 46
type(L[1]) # <class 'int'>
L[1]*10     # 460

L[0] # pop
type(L[0]) # <class 'str'>
```

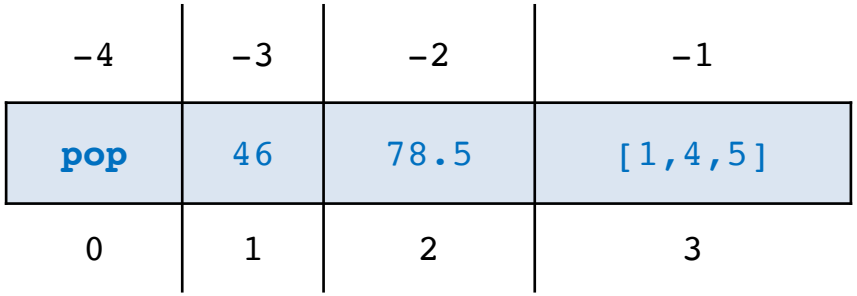
Since the first element is a string type, you can use another [ ] to access an element of that string

```
L[0][0] # p

L[-1]     # [1,4,5]
L[-1][0] # access 1st element of [1,4,5], which is 1
```

Access one element of a sequence - Indexing(tuple)

A tuple is a container of objects

```
seq[ index ] # access one element referenced by one index
```




```
T = ('pop',46,78.5, [1,4,5])
```

```
T[1]      # 46
T[0]      # pop
```

```
T[-1]       # [1,4,5]
T[-1][-1]   # access last element of [1,4,5], which is 5
```

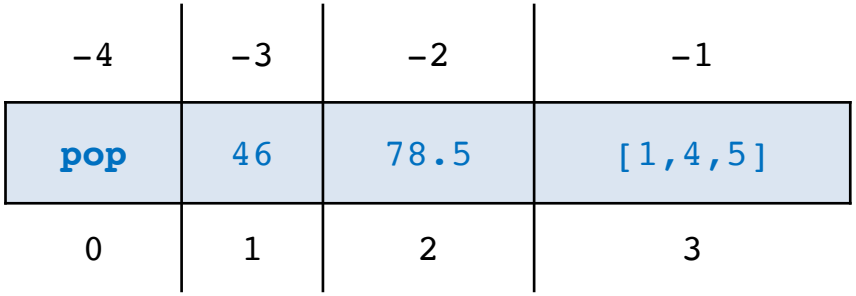
Subsequences can be created with the slice notation


```
seq[start:end]       # from index start through index end-1
seq[start:]          # from index start through last index
seq[:end]            # from index 0 through index end-1
seq[:]               # the whole sequence
seq[::-1]            # reverse the sequence
seq[start:end:step] # from index start through not past end, by step
seq[-2:]             # last two elements in the sequence
seq[:-2]             # everything except the last two elements
```

-11	-10	-9	-8	-7	-6	-5	-4	-3	-2	-1
H	**e**	**l**	**l**	**o**		**W**	**o**	**r**	**l**	**d**
0	1	2	3	4	5	6	7	8	9	10

```
s='Hello World'  # string type
s[1:4]   # from index 1 to index 3 'ell'
s[:4]    # from index 0 to index 3 'Hell'
s[6:]    # from index 6 to the end 'World'
s[1:11:2] # from index 1 to 10 in step of 2 'el ol'
s1=s[:]   # s and s1 reference the same string object (check id)

s2=s[::-1] # reverse the string
print(s2)
```

Subsequences can be created with the slice syntax




```
L = ['pop',46,78.5, [1,4,5]]  # list type

L[1:]   # element at index 1 through last index [46, 78.5, [1, 4, 5]]
L[:3]   # 1st element through element at index-1 ['pop', 46, 78.5]
L[::2]  # from index 0 to last index in step of 2 ['pop', [1, 4, 5]]
L[::-1] # reverse the list

L1=L[:]  # copy of the list - L1 references the copy (check the id)
```

Try to use slicing with a tuple type

```
T = ('pop',46,78.5, [1,4,5])
```

Creating a string object

To create a string, you enclose characters between single quotes or double quotes.

```
s1='Hello World'
type(s1)
```

```
s1="Hello World"
print(type(s1))
```

```
s1='' #we can also define an empty string – the null character
```

```
s1='678656'  #numbers enclosed in quotes are string objects
type(s1)
```

Creating a string object

Escape is used to remove the special meaning of single and double quotes

```
print("\"Hello Python\"")
```

but

```
print(" ' Hello Python ' ")
```

Triple quotes

```
"""
Or you can place COMMENTS inside triple quotes if you
have more than one line of text.
"""

greetings="""Hello word
I wanted to ask you if you are
happy """
```

Creating a string object from a number

the str() function

The str() function is used to convert an int number or float number to a string type

```
str(4) #return a string object
'4'

str(3.141592) #return a string object
'3.141592'
```

Create a string object - The join() method

The join() method is a string method that returns a string by joining all the elements of a container by a join separator.
The join method works with a list of strings or a tuple of strings.

```
'sep'.join(container of strings)
```

```
L=['one','two','three','four'] # homogeneous list of strings

s1=''.join(L1)
print(s1)

s2=' '.join(L)
print(s2)
```

If the container contains any non-string values, it raises a TypeError.

```
T=(1,2,3,4,5)
s3=' '.join(T)
TypeError: sequence item 0: expected str instance, int found
```

Creating a list object

A list type is a container of an ordered sequence of objects. The objects are items of the list. You can define a list with an expression of the form:

```
list_name = [elem1, elem2, elem3, elem4]
```

```
list1 = [1,3,5,7,9]   #homogeneous list, items are all numeric


list2 = ['red','blue','yellow'] #homogeneous list, items are all
string


list3 = ['pop',46,[1,4,5]] #non-homogeneous list, items are of
different types

list4 = []      # empty list
```

Creating a list object - list function

The list function is used to convert different types (like a string, or tuple) to a list type.
The list function creates a list object

```
list((1,2,3,4))  #tuple to list
[1, 2, 3, 4]

list('G+T+C')   #string to list
['G', '+', 'T', '+','C']
```

```
D={'lion':3,'elephant':10,'tiger':5}
list(D.keys()) #list of keys
['lion', 'elephant', 'tiger']

list(D.values()) #list of values
[3, 10, 5]

list(D.items()) #list of key,value tuple
[('lion', 3), ('elephant', 10), ('tiger', 5)]
```

Create a list from a string - the split() method

The split method is a string method that breaks up a string at the specified separator and returns a list of strings.

```
str.split(sep) #break up a string at the separator (sep)

str.split() #if sep is not specified, whitespace is the separator
```

Examples

```
s1="a b c d e a aa"
s1.split()
['a', 'b', 'c', 'd', 'e', 'a', 'aa']


s2="a:b:c:d:e"
s2.split(":")
['a', 'b', 'c', 'd', 'e']
```

Try the list function – what is the difference between the split method and list function?

```
list(s2)
['a', ':', 'b', ':', 'c', ':', 'd', ':', 'e']
```

Creating a tuple object

A tuple type is a container of an ordered sequence of objects, like a list type.
You can define a tuple with an expression of the form:

```
tuple_name= (elem1, elem2, elem3, elem4)
```

or

```
tuple_name= elem1, elem2, elem3, elem4
```

```
tuple1 = (1,3,5,7,9)     # homogeneous tuple, items are all numeric

tuple2 = ('a','b','c')   # homogeneous tuple, items are all string

tuple3 = ('pop',46,[1,4,5]) #non-homogeneous tuple, items are of
different types

tuple4 = ()                          # empty tuple
```

tuple object: packing and unpacking

When we create a tuple, we normally assign values to it.
This is called "packing" a tuple:

```
T=10,30,20,40
print(T)
(10, 30, 20, 40)
```

But, in Python, we are also allowed to extract the individual values into variables.
This is called "unpacking":

```
a,c,b,d=T
print(a)
10
```

This feature allows multiple variables assignments to be performed in this way

```
a,b="Hello",[1,3,5]
print(a) #Hello
print(b) #[1, 3, 5]
```

type conversion to tuple

You can use the tuple() function to create a tuple from certain objects

```
tuple('Mary') #string to tuple
('M', 'a', 'r', 'y')
```

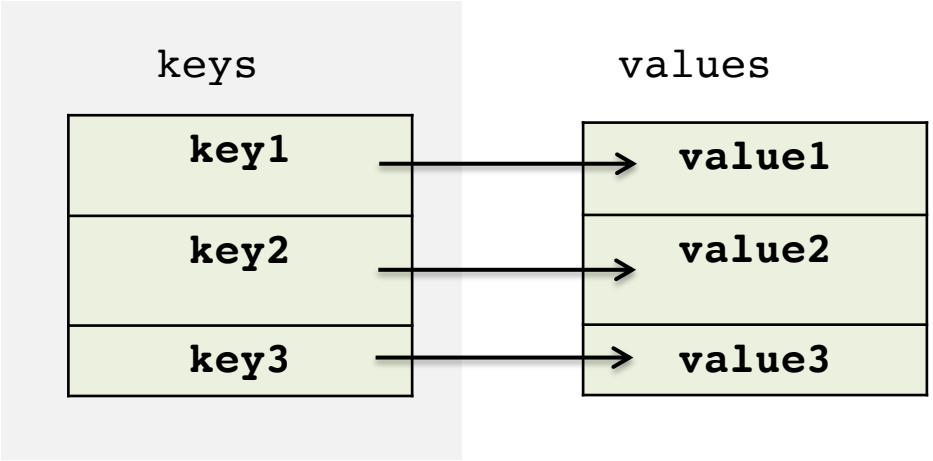
```
tuple([1,2,3,4])   #list to tuple
(1, 2, 3, 4)
```

Dictionaries

Dictionaries contain a collection of items that are pairs of keys and values, and it is a mapping data type.

```
d = {key1 :  value1, key2 : value2, key3 : value3}
```

- Keys can be **only** immutable data types, as strings or numbers
- Values can be of any data types




```
D={'lion': 3, 'elephant': 10, 'tiger': 5}
```

```
empty = {} # empty dictionary
```

Creating a Dictionary: dict() and zip()

Below are several ways of creating this same dictionary

```
D={'lion': 3, 'elephant': 10, 'tiger': 5}
```

You can use the dict() function

```
D1 = dict([('lion', 3), ('elephant', 10), ('tiger', 5)]) #from
a list of 2-elements tuple
```

You can create a dictionary from two lists, with dict(zip())

```
num=[3, 10, 5]
name=['lion', 'elephant', 'tiger']

D2=dict(zip(name,num)) #from two lists
```

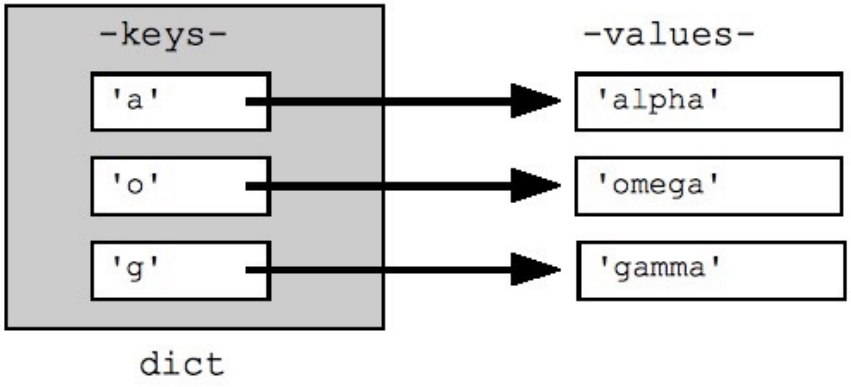
Dictionary - Accessing values

Dictionary is a mapping data type
You can access an individual value in a dictionary by **looking up a key** instead of an index.

```
d[key1] #access the value associated with key key1
```

You can access one individual value - only one key within brackets

```
d={'a':'alpha', 'o':'omega','g':'gamma'}
```




```
d['a']     #access value 'alpha'
d['a'][2]  #access character 'p' of 'alpha'
```

Dictionary does not support indexing, slicing, or other sequence-like behavior

Lists vs Dictionaries

List is a sequence data type

You can access an individual item in a list by its **index**.

```
seq[index]
['lion', 'elephant', 'tiger']
L[2] #tiger
```

Dictionary is a mapping data type

You can access an individual value in a dictionary by **looking up a key** instead of an index.

```
d[key1] #access the value associated with key key1

D={'lion': 3, 'elephant': 10, 'tiger': 5}
D["tiger"] #5
```

Dictionary can store more info, for example how many animals we have

The sys module and the argv attribute

`argv` is an attribute of the sys module used to pass a list of command line arguments to a Python script.
The argv attribute is similar to the $@ in bash.

Make a script called argv_test.py

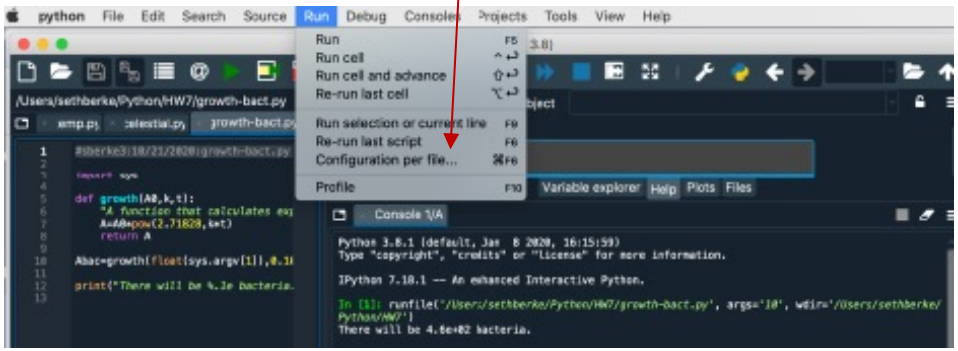
You can open a bash terminal and run the python script like this
`python3 argv_test.py test1 test2 test3 100 200`

```
from sys import argv
print(argv)      #argv is a list of command line arguments
['argv_test.py', 'test1', 'test2', 'test3', '100', '200']

print(argv[0])  #argv[0] is the name of your script
print(argv[1])  #argv[1] is the first argument
print(argv[1:])

#keep in mind that argv is a list of strings
```

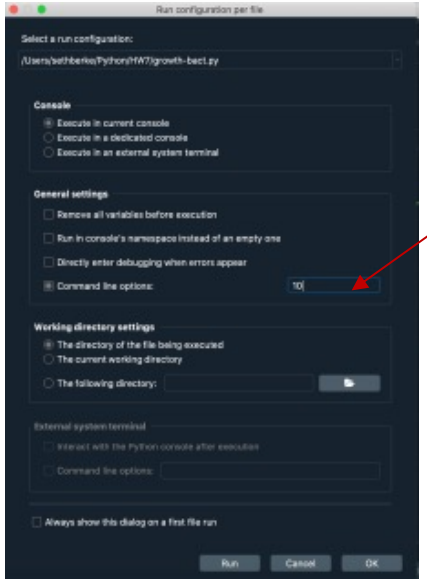
sys argv and editors

To pass arguments to a script in Spyder do the following:

1. Click Run → Configuration per files



2. Enter all the arguments in command line options




```
test1 test2 test3 100 200
```

3. Run the code

sys argv and editors

Passing arguments from IPython Console:

You can also pass arguments by running the script at IPython condole in Spyder like this:

```
runfile('path to the file', args='arg1 arg2 arg3')
```

You can use the absolute path

```
runfile('/Users/maria/argv_test.py', args='test1 test2 test3 100 200')
```

or a relative path, relative to your current directory in IPython – use pwd if you want to know your current directory

```
runfile('argv_test.py', args='test1 test2 test3 100 200')
```

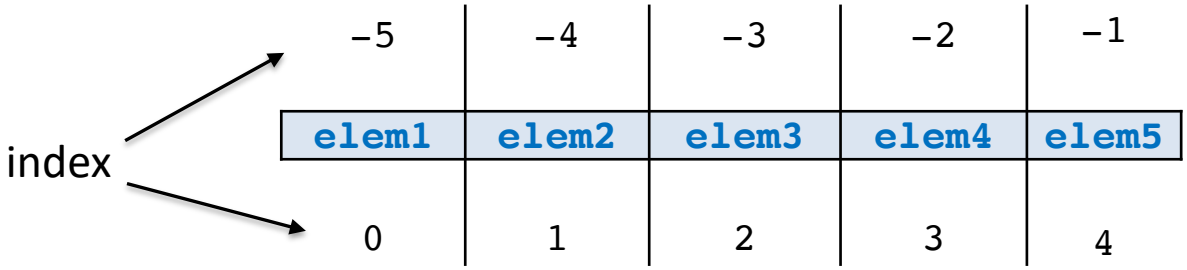
The arguments should be separated by a space, and all enclosed within single quotes.

Summary - Indexing a sequence type: string, list, tuple

Single indexing	
seq[i]	Return item at index i

Slicing	
seq[i:j]	Slice of seq from index i to index j-1
seq[i:j:k]	Slice of seq from index i to index j-1 with step k

```
seq[:]      # the whole sequence
seq[::-1]   # reverse the sequence
seq[-2:]    # last two elements in the sequence
seq[:-2]    # everything except the last two elements
```



Dictionary - Access values

```
d = {key1 :  value1, key2 : value2, key3 : value3}

d[key1] #access the value associated with key key1
```

Data Type Conversion examples

All these functions and methods return a new object which you can store in a variable

`int()`	string to integer	`int('2014')` `2014`
`int()`	floating point to integer	`int(3.141592)` `3`
`float()`	string to float	`float('1.99')` `1.99`
`float()`	integer to float	`float(5)` `5.0`
`str()`	integer to string	`str(4)` `'4'`
`str()`	float to string	`str(3.141592)` `'3.141592'`

`sep.join(iterable)`	list of strings to string	`L=['G', 'A', 'T', 'A']`
		`sep=''`
		`s=sep.join(L)`
		`print(s)`
		`'GATA'`

`sep.join(iterable`	tuple of string to string	`T=('1','2', '33'])`
		`sep='*'`
		`s=sep.join(T)`
		`print(s2)`
		`'1*2*33'`

Data Type Conversion examples

`tuple()`	string to tuple	`tuple('Mary')` `('M', 'a', 'r', 'y')`
`tuple()`	list to tuple	`tuple([1,2,3,4])` `(1, 2, 3, 4)`
`list()`	tuple to list	`list((1,2,3,4))` `[1, 2, 3, 4]`
`list()`	string to list	`list('G+T+C')` `['G', '+', 'T', '+','C']`

`string.split(sep)`	string to list	`s='G+T+C'`
		`L=s.split('+')`
		`print(L)`
		`['G','T', 'C']`

`list(dict.keys())`	dictionary to list of keys	`D={1: 'a', 2: 'b', 3}`
		`list(D.keys())`
		`[1,2,3]`

`list(dict.values())`	dictionary to list of values	`D={1: 'a', 2: 'b', 3: 'c'}`
		`list(D.values())`
		`['a','b','c']`

`list(dict.items())`	dictionary to list of key-value tuple	`D={1: 'a', 2: 'b', 3: 'c'}`
		`list(D.items())`
		`[(1, 'a'), (2, 'b'), (3, 'c')]`

`dict(zip(list1,list2))`	Two lists to dictionary	`l1=[1,2,3]`
		`l2=['a','b','c']`
		`d1=dict(zip(l1,l2))`
		`{1: 'a', 2: 'b', 3: 'c'}`