

## Contents

<b>1</b>	<b>Strings (str), revisited</b>	<b>2</b>
1.1	The <code>len()</code> function . . . . .	2
1.2	Indexing . . . . .	2
1.3	Slicing . . . . .	2
1.4	Arithmetic operators on strings . . . . .	3
1.5	Membership operator ( <code>in</code> ) . . . . .	3
1.6	String methods . . . . .	3
1.7	Immutability of strings . . . . .	3
<b>2</b>	<b>Lists</b>	<b>4</b>
2.1	Indexing, slicing . . . . .	4
2.2	Arithmetic operators on lists . . . . .	4
2.3	Membership operator ( <code>in</code> ) . . . . .	4
2.4	Mutability of lists . . . . .	4
2.5	List methods . . . . .	4
<b>3</b>	<b>Tuples</b>	<b>5</b>
<b>4</b>	<b>List comprehension</b>	<b>5</b>
<b>5</b>	<b>Dictionaries</b>	<b>6</b>
5.1	Membership operator ( <code>in</code> ) . . . . .	6
5.2	Dictionary methods . . . . .	6
<b>6</b>	<b>Iteration</b>	<b>6</b>

## 1 Strings (str), revisited

We briefly introduced strings in a previous lab, simply as a sequence of characters enclosed by quotation marks. There are many more operations that can be performed on strings, some of which apply to the other data types covered in this lab.

### 1.1 The len() function

You can get the length of a string using the `len()` function.

```
print(len("Hello, World!")) # 13 characters
print(len(""))              # 0 characters
```

### 1.2 Indexing

The characters in a string have an order to them. For example, given the string `"Hello, World!"`, we can say what the first, second, last, etc. characters are. We can access each character using an *index*, which is indicative of its position.

Each position in a string can be referenced two ways, with a non-negative index, and with a negative index. Python is zero-indexed, so the first element has index 0, and increases forwards. The last element has an index of -1, and decreases backwards.

letter	H	e	l	l	o	,		W	o	r	l	d	!
Positive index	0	1	2	3	4	5	6	7	8	9	10	11	12
Negative index	-13	-12	-11	-10	-9	-8	-7	-6	-5	-4	-3	-2	-1

Indexing allows us to get a *single character* from a string, and in Python is done with *square bracket notation* (these brackets: `[]`). Referring to the example and indices (plural of index) above:

```
x = "Hello, World!"
print(x[0])      # character at the 0th index, 'H'
print(x[4])      # character at the 4th index, 'o'
print(x[-1])     # character at the -1th index, '!'
```

### 1.3 Slicing

Slicing gets *multiple characters* from a string, and uses the syntax `my_string[start:end:step]`. Like the `range()` function, you can pass a start, end, and step, and the slice will always be exclusive of the end.

`start` takes the index 0 if unspecified, `end` takes the index `len(my_string)` if unspecified, and `step` takes a default value of 1 if unspecified.

You can mix both positive indices and negative indices, and should think of it as getting characters between two positions. This is rather uncommon, though.

```
print(x[0:3])      # characters between index 0 and 3, "Hel"
print(x[:6])       # characters until index 6, "Hello,"
print(x[-6:])      # characters from index -6, "World!"
print(x[-10:10])   # characters between index -10 and 10, "lo, Wor"
print(x[::-1])     # characters in the string backwards, "!dlroW ,olleH"
```

## 1.4 Arithmetic operators on strings

As in the lab 3 exercises where you experimented with operations on strings, you can concatenate (add) two strings together with the `+` operator, and also repeat a string with the `*` operator.

```
print("abc" + "def" + "ghi")
print("abc" * 2 + 2 * "def")
```

## 1.5 Membership operator (in)

The `in` operator tests whether some *substring* is contained in another string. It must be a *complete match*, as shown in the last two examples.

```
print(" " in "Hello, World!")  # True
print("ello" in "Hello, World!") # True
print("abc" in "aabbcc")       # False, 'abc' does not occur.
print("abc" in "aabcc")        # True
```

## 1.6 String methods

Methods are functions that are attached to, and typically act on, a specific object. For example, in the last lab, `str.upper()` was used to convert a string to all uppercase characters.

```
print("abc".upper())  # upper() acts on "abc", results in "ABC"
print("pqr".upper())  # upper() acts on "pqr", results in "PQR"
```

To see what methods are defined on an object, you can use the `help()` function on a type, i.e. `help(str)`. Some methods you may find useful: `str.find()`, `str.strip()`, `str.split()`, and `str.join()`.

```
print("abcba".find('b'))  # 1 - the index of the first occurrence
                        # of 'b'
print(" a b c ".strip())  # 'a b c' - with trailing spaces removed
print("abcbcb".split('b')) # ['a', 'c', 'c', 'a']
print('!'.join(['a', 'b', 'c'])) # 'a!b!c'
```

## 1.7 Immutability of strings

Mutability refers to whether an object can mutate or change. Strings are immutable, and you cannot modify a string – though you can create a new one from slices.

```
x = "immutable"
#x[3] = "c"  # causes an error
y = x[:2] + "c" + x[3:]  # "imcutable"
```

## 2 Lists

Lists are a sequential collection of items in Python, and are denoted by square brackets. While strings consist of characters, lists can consist of objects of any type. Like strings, you can get the length of a list with the `len()` function.

```
l = list('abc')           # must be some iterable object
l = ['a', 'b', 'c']       # the above line makes the same list
l = [1, 3.14159, 'a', True] # can contain different types
print(len(l))             # 4 items in the list.
```

### 2.1 Indexing, slicing

Indexing and slicing in lists is the same as in strings.

```
print(l[0])               # element at 0th index
print(l[:2])              # elements until 2nd index
print(l[-3:2])            # elements from -3th index to 2nd index
```

### 2.2 Arithmetic operators on lists

Similar to strings, you can concatenate two lists and repeat a list with the `+` and `*` operators respectively.

```
print(["abc"] + ["def"] + ["ghi"])
print(["abc"] * 2 + 2 * ["def"])
```

### 2.3 Membership operator (in)

The `in` operator works the same as in strings.

```
print(1 in [1, 2, 3])      # True
print("abc" in ['a', 'b', 'c']) # False
print("a" in ['abc'])      # False
```

### 2.4 Mutability of lists

Unlike strings, lists are mutable, and its contents can change freely.

```
l = [0, 1, 2, 3]
print(l)           # [0, 1, 2, 3]
l[2] = 8           # change the 2nd index to be 8
print(l)           # [0, 1, 8, 3]
```

### 2.5 List methods

Lists also have their own methods, which act and modify the list.

```
list.append(e)           # Add element e to the end of the list.
list.extend(iterable)    # Adds all the elements passed to the list.
x = list.pop(i)          # Removes and returns the element at index i.
list.insert(i, e)        # Inserts before index i the element e.
list.remove(e)           # Remove the first occurrence of e, if found.
list.sort()              # Sorts the list in-place. Returns nothing.
list.reverse()           # Reverses the list in-place. Returns nothing.
```

Some methods do not return anything. It is a common mistake to do the following:

```
l = [3, 2, 1]            # create a list
l = l.sort()             # l.sort() sorts in-place, and evaluates to None.
print(l)                 # l is None
l = l.reverse()          # causes an error
```

For now, just remember that assigning a variable `list.sort()` and `list.reverse()` like above will cause that variable to be `None`. You will learn more about this behaviour next week, when we learn to define our own functions.

### 3 Tuples

Tuples are denoted with parentheses. They function exactly like lists, but are immutable. Indexing, slicing, and the `in` operator all work the same, but trying to change the contents of a tuple will cause an error.

```
t = (1, 2, 3)            # create a tuple
print(len(t))            # 3
print(t[0])              # 1
#t[1] = 4                 # causes an error
#t.append(4)              # causes an error
```

### 4 List comprehension

A list comprehension is a concise way to create a new list from an existing list or other iterable. It consists of:

1. an iterable: The list or other iterable that you want to process.
2. an expression: The operation you want to apply to each element of the input iterable.
3. a condition (optional): A condition that filters out elements from the input iterable.

```
# [expression for i in iterable if condition]

# apply f(n) = 2*n for every n in [0, 1, 2, 3, 4]
l = [2*n for n in range(5)]          # [0, 2, 4, 6, 8]

# apply f(c) = ord(c) for every c in ['A', 'B', 'C']
l = [ord(c) for c in "ABC"]          # [65, 66, 67]

# apply f(x, y) = (x, y) for every x in [1, 2], y in ['A', 'B']
l = [(x, y) for x in [1, 2] for y in ("A", "B")]
# result: [(1, 'A'), (1, 'B'), (2, 'A'), (2, 'B')]

l = [c for c in "Hello, World!" if c.isupper()] # ['H', 'W']
```

You can create tuples and dictionaries with similar syntax.

## 5 Dictionaries

Dictionaries are key-value pairs, and are denoted with curly braces.

Like a language dictionary, when you want to look up the *definition of a word* (the ‘value’), you search for the *word* (the ‘key’). Dictionaries in Python generalise this for any information you wish to store. For example, you may want to keep track of account values:

```
dollars = {'alice': 839, 'bob': 1098, 'eve': 373}
dollars['caleb'] = 4      # add a new entry
```

### 5.1 Membership operator (in)

The membership operator `in` checks if a key is in the dictionary.

```
print('alice' in dollars)    # True
print('bobby' in dollars)    # False
print(839 in dollars)        # False: 839 is a value, not a key
```

### 5.2 Dictionary methods

Again, there are some methods that act on the dictionary:

```
dict.items()                # Get all key:value pairs in the dictionary
dict.keys()                  # Get all keys of the dictionary
dict.values()                # Get all values in the dictionary
dict.clear()                 # Empty the dictionary
dict.update(new_dict)        # Update dict with key:value pairs in new_dict
```

## 6 Iteration

You can iterate over strings, lists, tuples, and dictionaries.

```
for char in "this is a string":  
    print(char, end="")  
for e in ["this", "is", "a", "list"]: # same for tuples  
    print(e, end=" ")  
for k in {"key": "value", "dictionary": "pair"}:  
    print(k)
```