


Handout: Python cheat sheets

Introduction

This is a reference for the Python elements covered in this unit. The sheets include short explanations, brief notes, syntax, and selected examples.

The content has been grouped into categories:

- Lists
- List methods
- List functions
- List operators
- Strings
- String functions
- String operators
- Iterating over sequences

There is also additional information that is not covered in the unit but may be useful in solving relevant problems. It is signposted with the Explorer icon: 

This resource is available online at ncce.io/pysheets-seqs. Resources are updated regularly — please check that you are using the latest version.

This resource is licensed under the Open Government Licence, version 3. For more information on this licence, see ncce.io/ogl.

Lists

Lists are a type of data structure that involve individual items organised in a sequence.

Lists are dynamic data structures: items can be modified, added, or removed during program execution.

Create a list

Syntax

```
[comma-separated list of items]
```

Examples

```
summer = ["June", "July", "August"]
```

Lists are usually **assigned** when they are created (so they can be referred to and modified later on).

```
numbers = []
```

A list can be **empty**.

```
data = [8, True, "Hello", 3.14]
```

Lists can feature items of different data types.

Access individual list items

Syntax

```
list[index]
```

The items in a list can be accessed through an **index**, i.e. their current position in the list, with numbering starting at zero.

Examples

```
month = summer[0]
```

Retrieve the value of the **first** item (zero-based index 0) in a list.

```
data[1] = False
```

Assign a new value to the **second** item (zero-based index 1) in a list.

```
previous = planets[position-1]  
sum = numbers[i] + numbers[i+1]
```

The index can be the value of an **expression**.

List slices

Syntax

```
list[start index:end index:step]
```

A slice of a list is a new list that includes list items from a start index up to (but not including) an end index. Specifying a step skips over items.

Examples

```
summer = months[5:8]
```

The new list is a slice containing items 6 to 8.

```
head = data[:100]
```

You can omit the start index (start from the first item) and the end index (stop at the last item).

```
skipped = values[::2]
```

Skip every other item.

List methods

You can think of **list methods** as special functions that are applied to lists. To call a list method, you need to use **dot notation** (as shown in the examples that follow).

Add or remove items

Syntax

```
list.append(item)
```

Add an item to the end of the list.

Example

```
numbers.append(42)
```

Syntax

```
list.insert(index, item)
```

Insert an item at a given (zero-based) index.

Example

```
cities.insert(2, "Oslo")
```

Insert a new item at the third position (zero-based index 2) in the list.

Syntax

```
list.pop(index)
```

Remove the item at the given (zero-based) index in the list, and return it. If no index is specified, remove and return the last item in the list.

Examples

```
tasks.pop()
```

```
last = values.pop()
```

The value removed from the list and returned by **pop** can be assigned to a variable.

```
queue.pop(0)
```

Remove the first item (zero-based index 0) from the list.

Syntax

```
list.remove(item)
```

Remove the first item from the list with a particular value. Raises a **ValueError** if there is no such item.

Example

```
countries.remove("Japan")
```

List methods

You can think of **list methods** as special functions that are applied to lists. To call a list method, you need to use **dot notation** (as shown in the examples that follow).

Find and count items

Syntax

```
list.index(item)
```

Search for the first occurrence of an item in the list and return its (zero-based) index. Raises a **ValueError** if there is no such item.

Example

```
pos = planets.index("Mars")
```

Syntax

```
list.count(item)
```

Return the number of times an item appears in the list.

Example

```
nb_the = words.count("the")
```

Other list operations

Syntax

```
list.reverse()
```

Reverse the items of the list.

Example

```
values.reverse()
```

Syntax

```
list.sort()
```

Sort the items in the list in ascending order.

Examples

```
names.sort()
```

```
numbers.sort(reverse=True)
```

The items can be strings (and sorting arranges them in alphabetical order).

Use the **reverse=True** argument to sort in descending order.

List functions

Some functions can accept lists as arguments, process them, and return a result.

Length of a list: the `len` function

Syntax

```
len(list)
```

Return the length (number of items) of a list.

Example

```
len(planets)
```

Other functions

Syntax

```
sum(list)  
min(list)  
max(list)
```

Return the sum of the list elements, the lowest and greatest values in the list, respectively.

List operators

List operators allow you to form expressions that involve lists and can be evaluated.

List membership: the `in` operator

Syntax

```
item in list
```

Check if the list contains items with a specific value. This expression evaluates to **True** or **False**.

Examples

```
"Pluto" in planets  
answer in ["yes", "no"]  
name in guests
```

```
not "London" in destinations  
"London" not in destinations
```

There are two ways to check if a list does **not** contain a specific value.

Adding lists together

Syntax

```
list + list
```

This expression evaluates to a new list that comprises the two lists, joined together in sequence.

Examples

```
numbers = [4, 9, 3] + [6, 3, 2]  
pupils = year7 + year8 + year9
```

Strings

Strings are a type of data structure where individual characters are organised in a sequence.

Strings **cannot** be modified during program execution.

Create a string

Syntax

```
"character sequence"
```

Examples

```
month = "August"
```

Strings can be **assigned** to variables when they are created (so they can be referred to later on).

```
empty = ""
```

A string can be **empty**.

Access individual string characters

Syntax

```
string[index]
```

String character can be accessed through an **index**, i.e. their current position in the string, with numbering starting at zero.

Examples

```
letter = month[0]
```

Retrieve the **first** character (zero-based index **0**) in a string.

```
character = password[position-1]
```

The index can be the value of an **expression**.

```
language[1] = "A"
```

An individual character in a string **cannot** be assigned a new value.

String slices

Syntax

```
list[start index:end index:step]
```

A slice of a string is a new string that includes the characters from a start index up to (but not including) an end index. Specifying a step skips over items.

Examples

```
substring = word[5:8]
```

The new list is a slice containing items 6 to 8.

```
prefix = word[:3]
```

You can omit the start index (start from the first character) and the end index (stop at the last character).

```
skipped = name[::2]
```

Skip every other item.

String functions

Some functions can accept strings as arguments, process them, and return a result.

Length of a string: the `len` function

Syntax

```
len(list)
```

Return the length (number of characters) of a string.

Example

```
len(password)
```

String operators

String operators allow you to form expressions that involve strings and can be evaluated.

String membership: the `in` operator

Syntax

```
substring in string
```

Check if a string is contained within a larger string. This expression evaluates to **True** or **False**.

Examples

```
"sub" in word  
letter in "aeiou"  
word in text
```

Adding strings together

Syntax

```
string + string
```

This expression evaluates to a new string that comprises the two strings joined together in sequence.

Examples

```
greeting = "Hello " + name + "!"  
fullname = firstname + lastname
```

Split and join

It is often convenient to split a string into a list, or join the items of a list into a string.

Syntax

```
string.split(separator)  
separator.join(list)
```

Examples

```
names = line.split(", ")
```

```
"".join(letters)
```

Iterating over sequences

The **for**-loop is a special type of control structure that can be used to iterate over the elements of a sequence.

Syntax

```
for element in sequence:  
    block of statements
```

For every element in the sequence, execute the block of statements.

Iterating over list items

Syntax

```
for item in list:  
    block of statements
```

Execute the block of statements for every item in the list.

Example

```
for name in guests:  
    print(name)
```

Iterating over string characters

Syntax

```
for character in string:  
    block of statements
```

Execute the block of statements for every character in the string.

Example

```
for character in password:  
    print(character)
```

Using while instead of for

You can follow this pattern to use **while** to achieve a similar effect as when using **for**:

Pattern

```
index = 0  
while index < len(sequence):  
    element = sequence[index]  
    block of statements  
    index = index + 1
```

Iterate over all indices, retrieve the corresponding element in the sequence, and execute the block of statements.