Introduction to Python

Session 03

Container objects: Lists, tuples and dictionaries
Modules

Summary

- In python, everything is an object!
- Some objects we have seen:
  - Integers
  - Float
  - Boolean (True/False)
  - None
  - Files
  - String: Non-mutable sequence of characters

upf.

Container objects

- Objects that contain other objects: containers
- Organize objects according to different requirements:
  - How I should be able to find objects?
    - By **position**? (Ordered container)
    - By an **identifier**/value?
    - Should I remove **duplicate** objects, or allow duplicates.
    - Should I be able to **modify** my group of objects?

upf.

Container objects

- Lists
- Tuples
- Dictionaries
- Sets

upf.

Container objects

- **Lists**
- Tuples
- Dictionaries
- Sets

upf.

Lists

- A **List** is an ordered **sequence** of Objects.
- Creation with: []

```
my_first_list = [ 1, 2, 3, 4, 5 ]
```

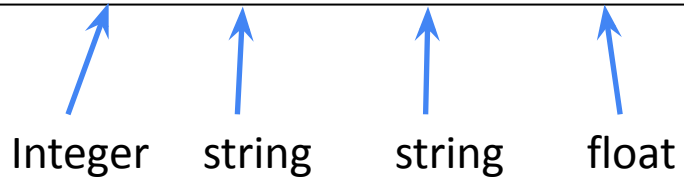
```
empty_list = []
```

upf.

Lists

- A **List** can contain objects of different classes.

```
a = [1,"a","b",3.14]
```



Lists

- **Index**: position in the list. **Lists are ordered!**

```
>>> L = [1,"a","b",3.14]
>>> L[1]
'a'
>>> L[-1]
3.14
```

Positive indices

Negative indices

Lists

- List operators:
  - **Concatenate: +**

```
>>> l = [1,2,3,4]
>>> l+l
[1, 2, 3, 4, 1, 2, 3, 4]
```

  - **Replicate: ***

```
>>> l*4
[1, 2, 3, 4, 1, 2, 3, 4, 1, 2, 3, 4, 1, 2, 3, 4]
```

  - **Indexing: []**
  - **Slicing: [:]**

- **Slicing**

Slice: sublist of a list `[Start index:End index+1:step]`

```
>>> l = [1,2,3,4]
>>> l[1:3]
[2, 3]
```

```
>>> l = [1,2,3,4]
>>> l[1:-1]
[2, 3]
```

```
>>> l = [1,2,3,4]
>>> l[::-1]
[4, 3, 2, 1]
```

upf.

Lists

- Lists are **mutable**!

```
>>> l = [1,2,3,4]
>>> l[0]="a"
>>> l
['a', 2, 3, 4]
```

upf.

Lists

- **len**: built-in function to get the length of a list

```
>>> my_list = [1,”a”,”hello”]
>>> len(my_list)
3
```

upf.

Lists

- Traversal of a List

```
>>> my_list = [1,None,"A",True]
>>> i = 0
>>> while i<len(my_list):
...     print(type(my_list[i]))
...     i+=1
...
<type 'int'>
<type 'NoneType'>
<type 'str'>
<type 'bool'>
```

upf.

Lists

- Traversal of a List: **for** *element* **in** *list*

```
>>> my_list = [1,None,"A",True]
>>> for element in my_list:
...     print element
...
1
None
A
True
```

upf.

Lists

- **in** operator:

```
>>> my_list = [1,None,"A",True]
>>> "A" in my_list
True
>>> "a" in my_list
False
```

- Negation: **not** in

```
>>> my_list = [1,None,"A",True]
>>> "A" not in my_list
False
>>> "a" not in my_list
True
```

upf.

List methods


```
>>> my_list = [1,None,"A",True]
>>> dir(my_list)
['__add__', '__class__', '__contains__',
'__delattr__', '__delitem__', '__delslice__',
'__doc__', '__eq__', '__format__', '__ge__',
'__getattribute__', '__getitem__',
'__getslice__', '__gt__', '__hash__',
'__iadd__', '__imul__', '__init__', '__iter__',
'__le__', '__len__', '__lt__', '__mul__',
'__ne__', '__new__', '__reduce__',
'__reduce_ex__', '__repr__', '__reversed__',
'__rmul__', '__setattr__', '__setitem__',
'__setslice__', '__sizeof__', '__str__',
'__subclasshook__', 'append', 'count', 'extend',
'index', 'insert', 'pop', 'remove', 'reverse',
'sort']
```

List methods

- **append**: append and object to the end of the list
- **count**: return an integer corresponding to the number of occurrences of a value
- **extend**: append all the elements of a list at the end of the other list
- **index**: return the first index of a value (raises error if it is not found)

upf.

List methods

- **insert**: insert an object before the specified index
- **pop**: return and remove the object at specified index. If no index is specified, return the object in the last position. Raises exception if list is empty or index is out of range.
- **remove**: removes the first occurrence of a value. Raises exception if it is not found
- **reverse**: reverses the order of the list. **Modifies** the same list, it does not create a new list!
- **sort**: orders the element of the list. **Modifies** the same list, it does not create a new list!

upf.

Lists vs Strings

- **Strings**: ordered sequence of characters
- **Lists**: ordered sequence of Objects

- A list of characters is not the same as a string!

```
>>> word = "bioinformatics"
>>> word_list = list(word)
>>> print(word_list)
['b', 'i', 'o', 'i', 'n', 'f', 'o', 'r', 'm',
'a', 't', 'i', 'c', 's']
```

upf.

Lists vs Strings

- When we split a string, we obtain a list:

```
>>> sentence = "This is a sentence."
>>> sentence.split(" ")
['This', 'is', 'a', 'sentence.’]
```

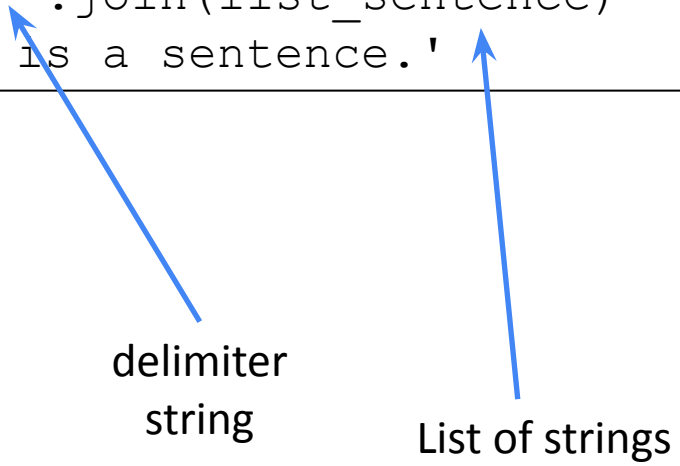
```
>>> paragraph = "This is the first sentence.
This is the second sentence. This is the third
sentence."
>>> sentences = paragraph.split(".")
>>> print(sentences)
['This is the first sentence', ' This is the
second sentence', ' This is the third sentence',
'']
>>> sentences[0]
'This is the first sentence'
```

upf.

Lists vs Strings

- **join**: inverse of split. Join the strings of a list with the given delimiter. Join is a method of the string.

```
>>> list_sentence = ['This', 'is', 'a',
'sentence.']
>>> " ".join(list_sentence)
'This is a sentence.'
```



upf.

Container objects

- Lists
- **Tuples**
- Dictionaries
- Sets

upf.

Tuples

- A **Tuple** is an ordered sequence of Objects.

- Creation with: ()

```
my_first_tuple= ( 1, 2, 3, 4, 5 )
```

```
empty_tuple= ()
empty_tuple = tuple()
```

```
my_second_tuple= ( 1, )
```

To create a tuple with a single object!

upf.

Tuples

- Tuples and Lists are very similar:
  - Concatenation: +
  - Replication: *
  - Indexing: [index]
  - Slicing: [start:end:step]

upf.

Tuples

- Tuples and Lists are very similar… but:
  - Lists are mutable: you can change the elements of a list (append, remove, pop,…)
  - Tuples are **immutable!**

```
>>> t = (0,1,2,3)
>>> t[3]=1
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'tuple' object does not support item assignment
```

upf.

Tuples

- Why are tuples useful?
  - When we need an immutable ordered sequence of objects.
  - A function only can return a **single value**. If in some cases we need to return more values, we can use a tuple object. This is the default behavior when multiple values are returned.

upf.

Container objects

- Lists
- Tuples
- **Dictionaries**
- Sets

upf.

Dictionaries

- A dictionary is a collection of object in which objects are indexed with a key value:
  - Mapping between a set of indices (keys) and a set of objects (values)
    - KEY – VALUE pair
  - To create a dictionary: **{ }**
  - To create a filled dictionary: **{ key: value, key: value}**

```
eng2sp = { "one": "uno",
           "two": "dos",
           "three": "tres"}
```

upf.

Dictionaries

- Adding new key-values to dictionaries: [ ]

```
dictionary[KEY] = VALUE_OBJECT
```

```
>>> eng2sp = { “one”: “uno”,
               “two”: “dos”,
               “three”: “tres”}

>>> eng2sp["four"] = "cuatro"
```

- Accessing value for a given key:

```
value = dictionary[KEY]
```

upf.

Dictionaries

- len: function to get the number of key-value pairs of a dictionary

```
>>> eng2sp = { "one": "uno", "two": "dos", "three": "tres"}
>>> len(eng2sp)
3
```

Dictionaries

- Traversal of a dictionary: **for** *key* **in** *dictionary*

```
>>> eng2sp = {“one”: “uno”,
              “two”: “dos”,
              “three”: “tres”}

>>> eng2sp["four"] = "cuatro”

>>> for english_word in eng2sp:
...     print(english_word, eng2sp[english_word]

('four', 'cuatro')
('three', 'tres')
('two', 'dos')
('one', 'uno')
```

upf.

Dictionaries

- Traversal of a dictionary: **iteritems**

```
>>> eng2sp = {“one”: “uno”,
              “two”: “dos”,
              “three”: “tres”,
              “four”: “cuatro”}

>>> for eng, esp in eng2sp.items():
...     print(eng, esp)
...
('four', 'cuatro')
('three', 'tres')
('two', 'dos')
('one', 'uno')
```

upf.

Dictionaries

- Check if a dictionary has a key:
  - *key* **in** *dictionary*

```
>>> eng2sp = {“one”: “uno”,
              “two”: “dos”,
              “three”: “tres”}

>>> "one" in eng2sp
True
>>> "five" in eng2sp
False
```

upf.

Dictionary methods

- **get:** returns the value for a key. In not in found returns a default value instead of raising a KeyNotFound Error
- **keys**: return a list with all the keys.
- **values**: return a list with all the values.
- **setdefault**(key, default_value): return the value for a given key. If this key is not found, insert it to the dictionary with the given default value.
- **pop**: remove specified key and return the corresponding value

upf.

Dictionary methods

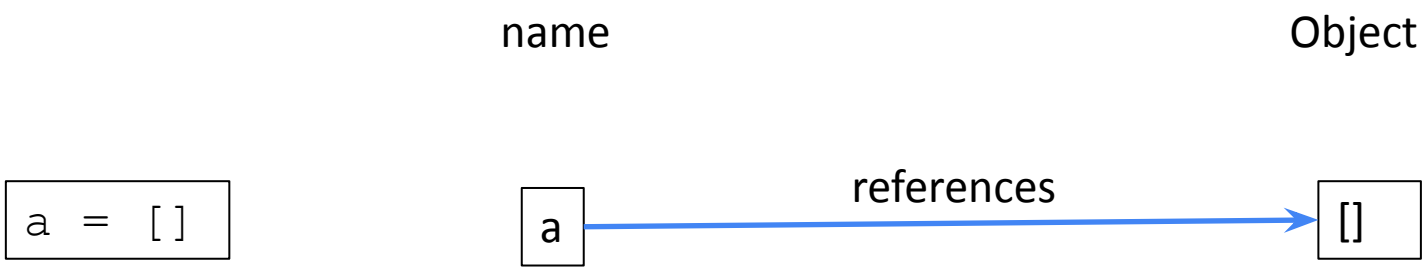
- **popitem**

- **items**: returns an iterator over the (key, value) of the dictionary
- …

upf.

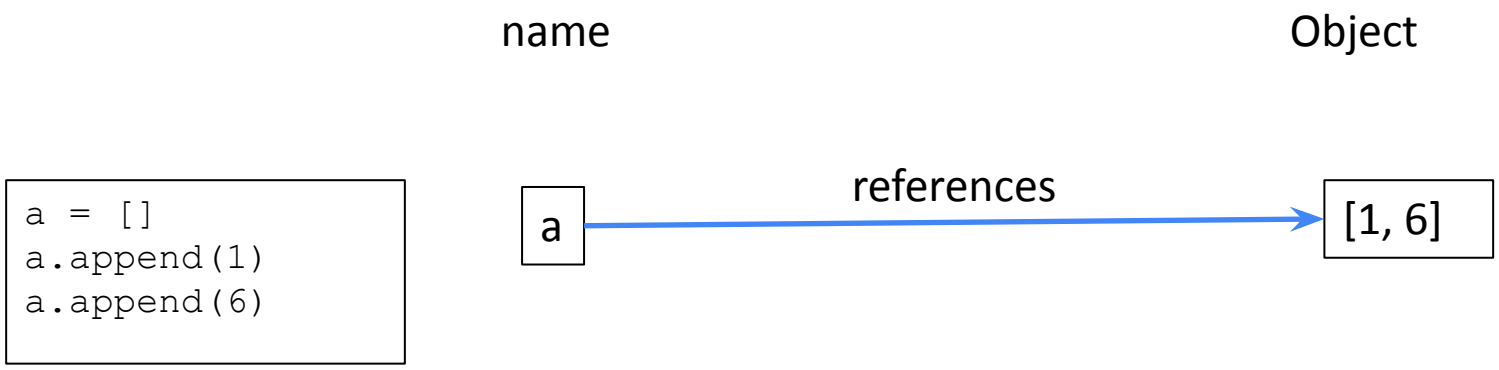
Variable names are references to objects

Variable assignment: Create a new variable and assign a value. In Python, we assign a reference of an object to a name.



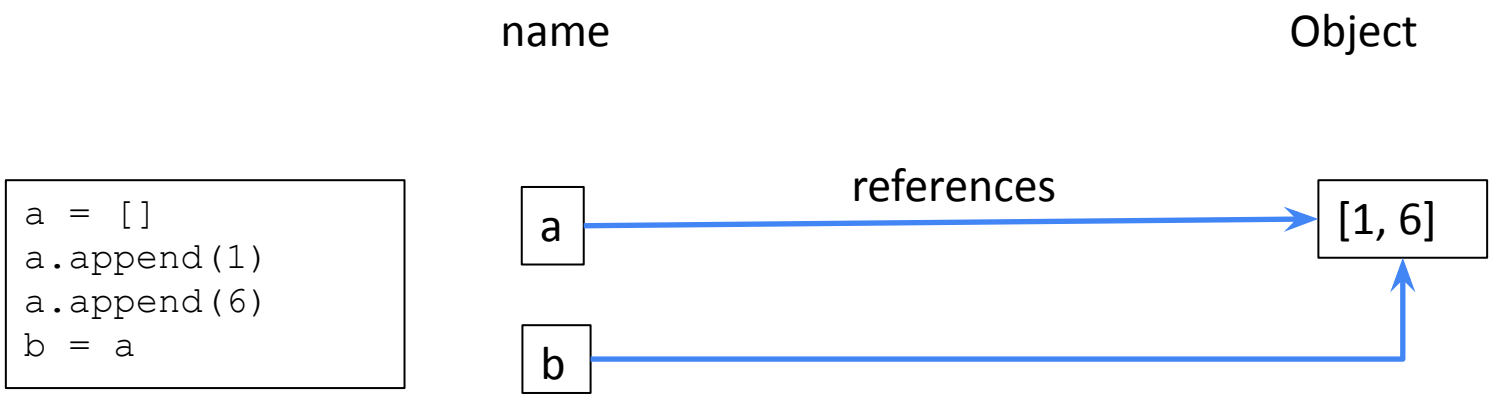
Variable names are references to objects

Variable assignment: Create a new variable and assign a value. In Python, we assign a reference of an object to a name.



Variable names are references to objects

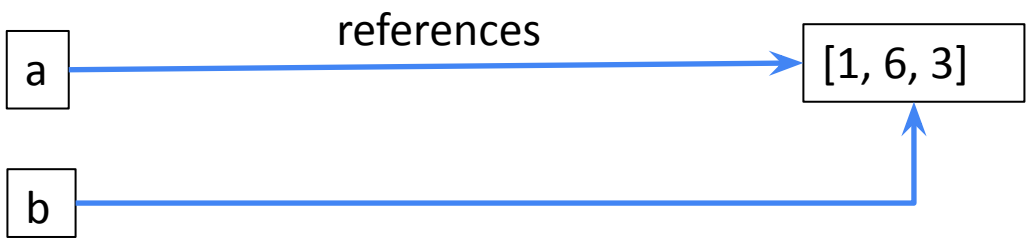
Variable assignment: Create a new variable and assign a value. In Python, we assign a reference of an object to a name.



Variable names are references to objects

Variable assignment: Create a new variable and assign a value. In Python, we assign a reference of an object to a name.

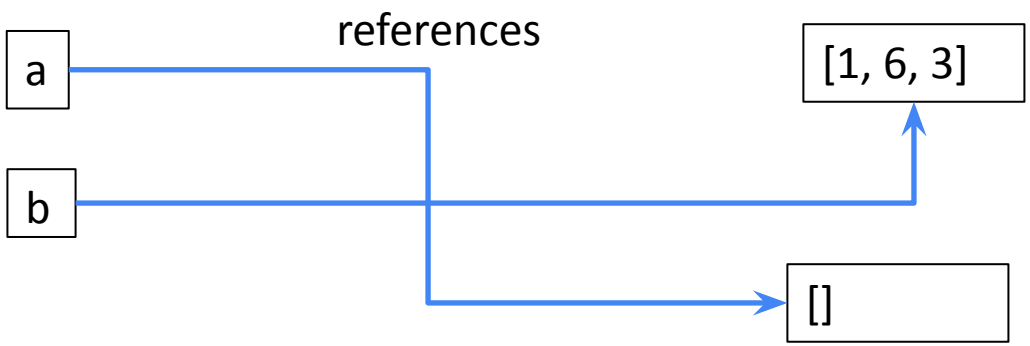
```
a = []
a.append(1)
a.append(6)
b = a
b.append(3)
```



Variable names are references to objects

Variable assignment: Create a new variable and assign a value. In Python, we assign a reference of an object to a name.

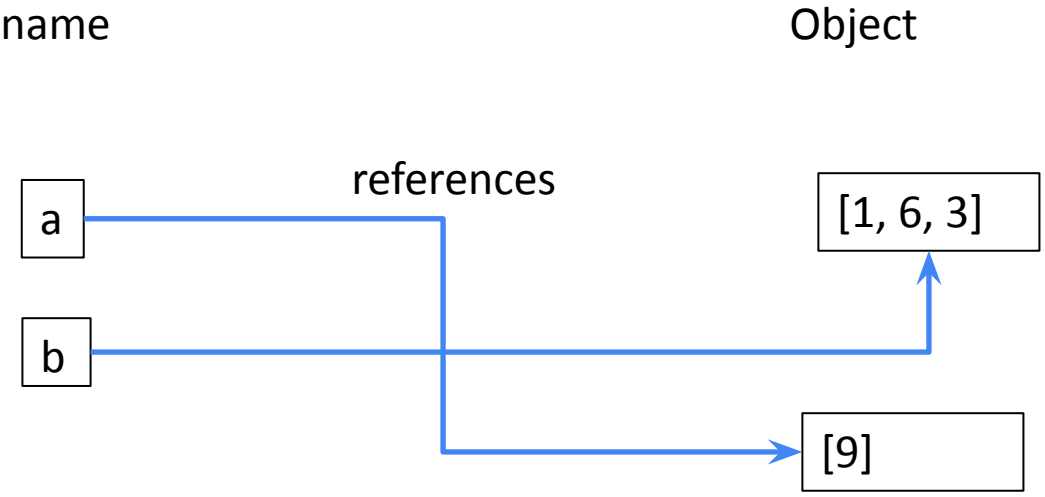
```
a = []
a.append(1)
a.append(6)
b = a
b.append(3)
a = []
```



Variable names are references to objects

Variable assignment: Create a new variable and assign a value. In Python, we assign a reference of an object to a name.

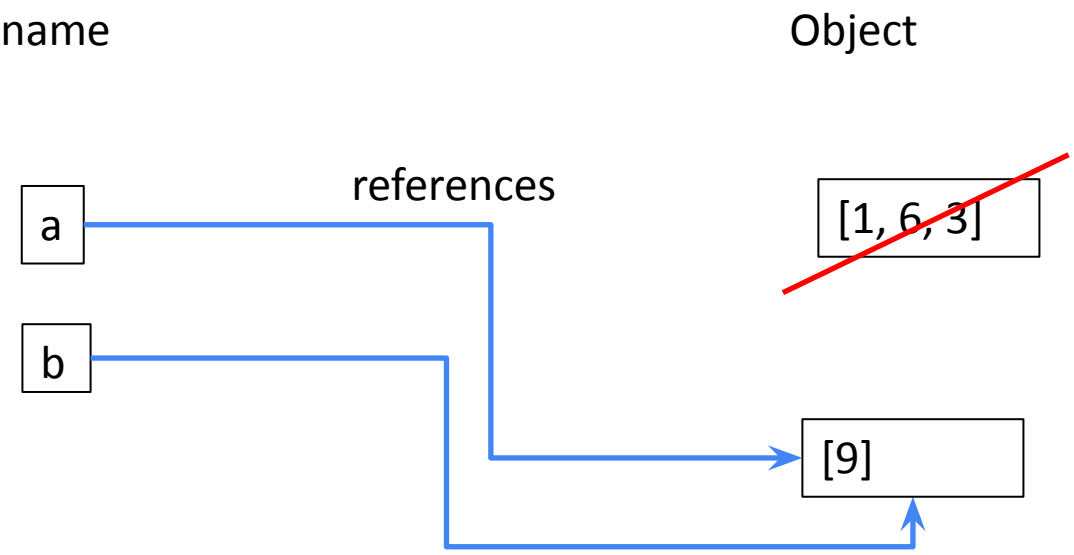
```
a = []
a.append(1)
a.append(6)
b = a
b.append(3)
a = []
a.append(9)
```



Variable names are references to objects

Variable assignment: Create a new variable and assign a value. In Python, we assign a reference of an object to a name.

```
a = []
a.append(1)
a.append(6)
b = a
b.append(3)
a = []
a.append(9)
b = a
```



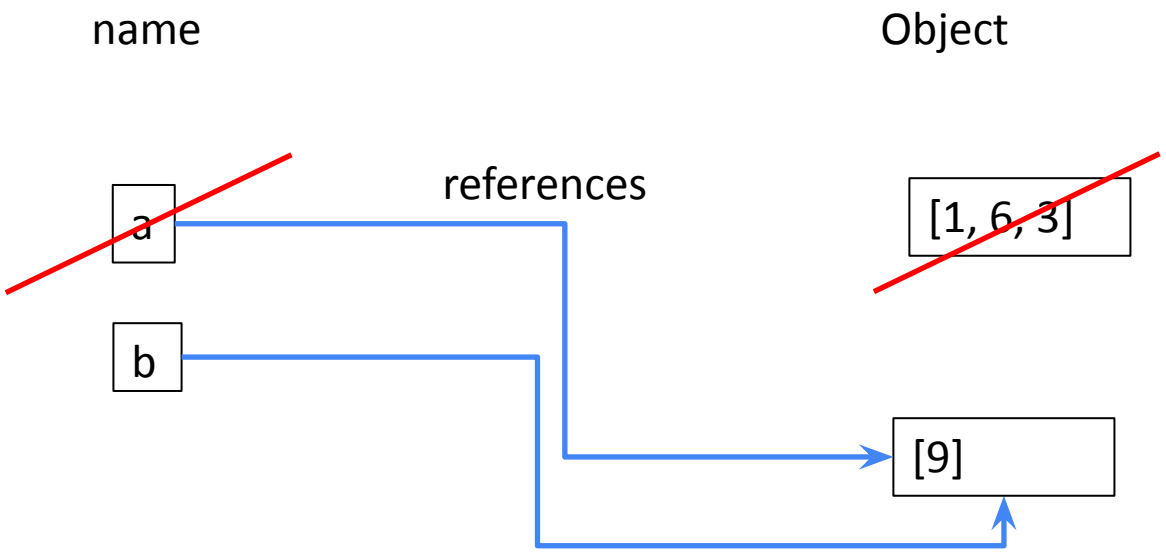
- **Garbage Collection:** When an object is not referenced by any variable, it is automatically destroyed.

upf.

Variable names are references to objects

Variable assignment: Create a new variable and assign a value. In Python, we assign a reference of an object to a name.

```
a = []
a.append(1)
a.append(6)
b = a
b.append(3)
a = []
a.append(9)
b = a
del a
```

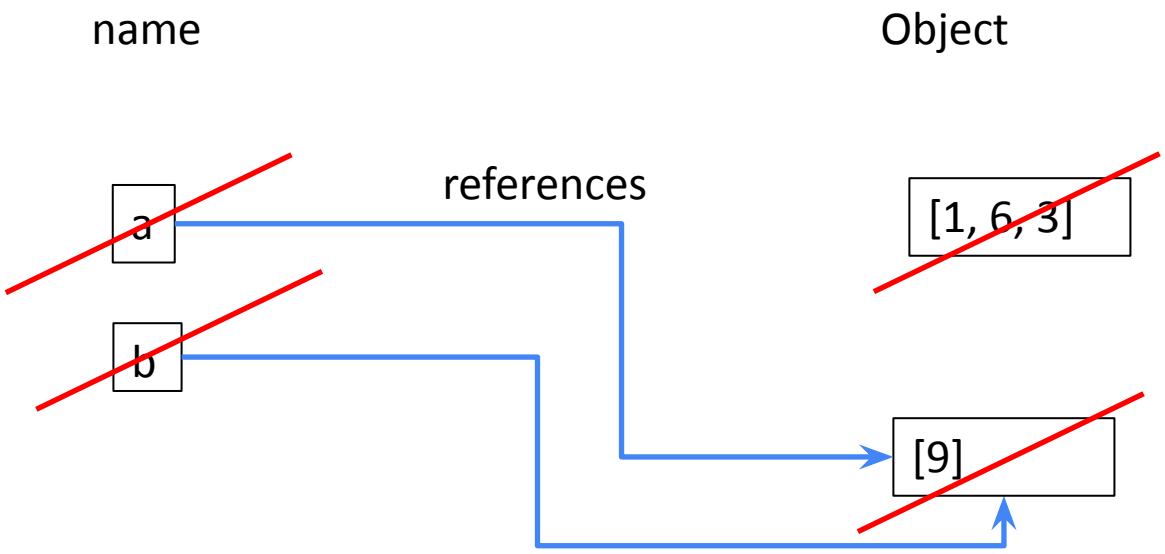


- **Garbage Collection:** When an object is not referenced by any variable, it is automatically destroyed.

Variable names are references to objects

Variable assignment: Create a new variable and assign a value. In Python, we assign a reference of an object to a name.

```
a = []
a.append(1)
a.append(6)
b = a
b.append(3)
a = []
a.append(9)
b = a
del a
del b
```



- **Garbage Collection:** When an object is not referenced by any variable, it is automatically destroyed.

Introduction to modules

- A file with .py extension containing python code is a module.
- A module can contain:
  - Variables and functions
- Why modules?
  - Organizing code in different files.
  - Highest level way of organizing a program.
    - A large program can have several modules, each one with a section of related code.

upf.

Introduction to modules

- Example: a module in file named exercise1.py

Import a module

```
import exercise1

volume = exercise1.get_sphere_volume(10, 5)

a = exercise1.recursive_factorial(20)

dir(exercise1)
```

List all variables and functions available in the module

Introduction to modules

- Example: a module in file named exercise1.py

Different ways to import a module

```
import exercise1
volume = exercise1.get_cone_volume(10, 6)
```

```
import exercise1 as ex1
volume = ex1.get_cone_volume(10, 6)
```

```
from exercise1 import get_cone_volume
volume = get_cone_volume(10, 6)
```

```
from exercise1 import *
volume = get_cone_volume(10, 6)
```

upf.

Introduction to modules

- Example: the **math** module

```
import math

math.factorial(10)

math.pi

math.cos(2*math.pi)


dir(math)
```

upf.

Exercises. Block 1. Part 3.

Create a python script called <uID>_S03.py with the following function:

1) Create a function that, given a multi-line protein FASTA file (*fasta_filename*) and a "sub-sequences" file (*subsequences_filename*) (one sequence in each line), calculates the proportion of proteins in the FASTA file containing at least N-times (*number_of_repetitions*) each of the sub-sequences (exactly equal). **Save** it in an **output file** with the specified format, ordered by the proportion value (descending order)

```
calculate_aminoacid_frequencies(fasta_filename,
                                subsequences_filename,
                                number_of_repetitions,
                                output_filename)
```

Output format:

```
#Number of proteins:                12131
#Number of subsequences:               45
#Subsequence proportions:
RTAW                    1621       0.1336
MV                       261       0.0215
…
```

Right aligned.

Aligned at position: 40

Right aligned. Position: 20

4 decimal positions