Introduction to Python

Session1,2

AI

Python

Session-02

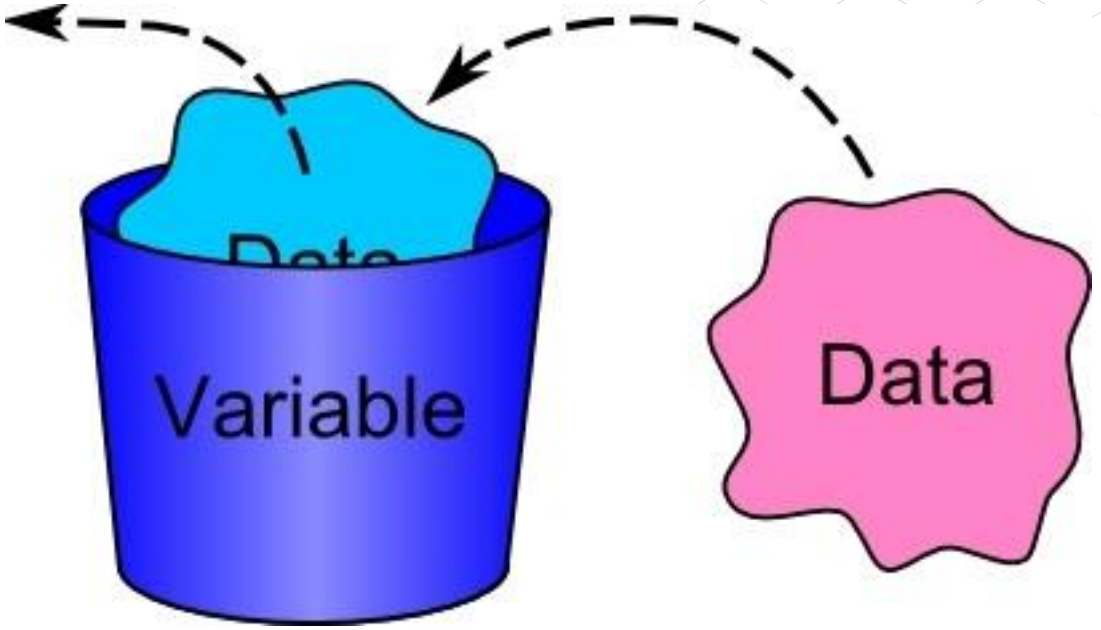
Objectives

- What is a variable
- Variable assignment
- Variables and operators
- What is a type
- Most common types
  - String
  - Integer
  - Float
  - Boolean
- Type methods
- Exercises with variables and types

Variables

What is a variable?

- Variable is a reserved location in the memory that stores a value. which can be easily retrieved later in the program.
- Python is not statically typed language which means you don't need to declare the type or the variable before using them.



Variable Name

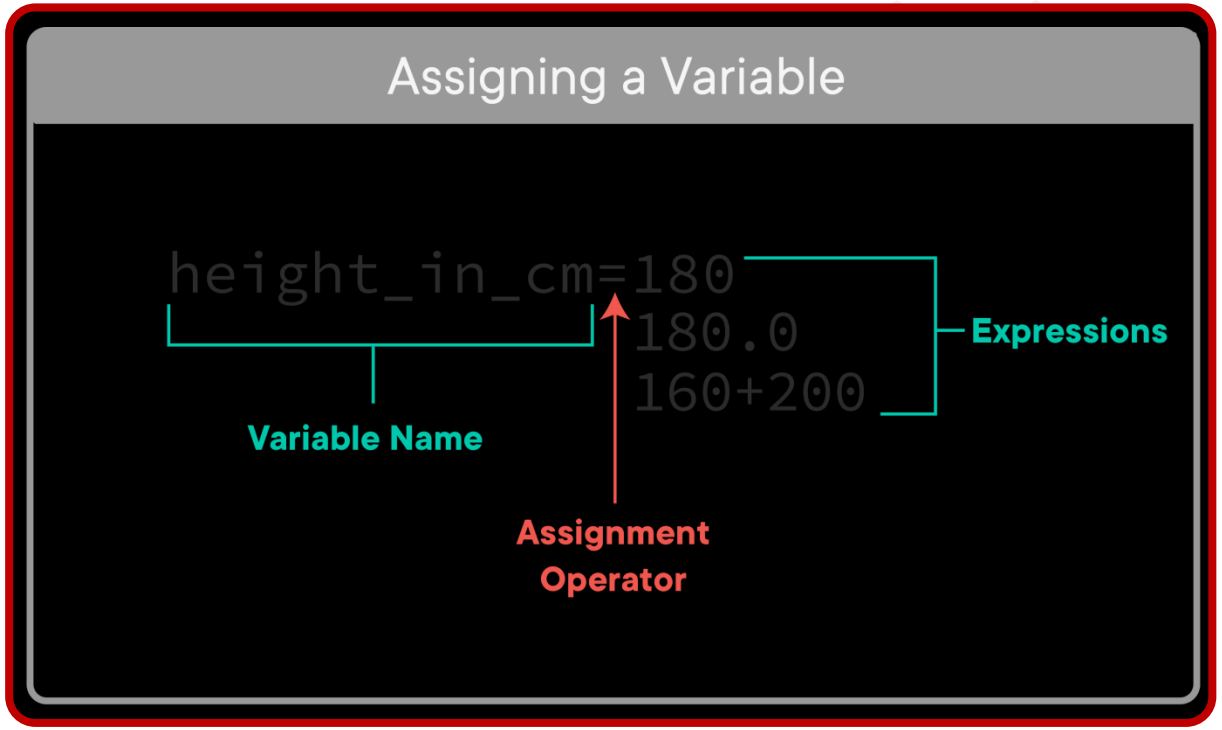
The variable name must follow the following criteria:

==========

- must start with a letter or underscore
- can't start with a number
- only alpha-numeric and underscore values accepted (A-Z, a-z, 0-9, _)
- case sensitive (ITI is different from iti)
- it can be one letter (x) or a name with a meaning (my_name)

Variable Assignment

Assignment is done with a single equals sign (=) with a meaning (my_name)



Variable Assignment (cont.)

```
year = 2021
print(year)
```

```
2021
```

Data type of variable may be changed after the variable has been set. or (re-declaring)

```
year = "last year"
print(year)
```

```
last year
```

Variable Assignment (cont.)

Assign values to multiple variables in one line

```
ali, shaymaa, mohamed = "pass", "pass", "fail"
print(ali)
print(shaymaa)
print(mohamed)
```

Variable Assignment (cont.)

Assign values to multiple variables in one line
(different data type)

```
ali, shaymaa, mohamed = "pass", 3 , "fail"
print(ali)
print(shaymaa)
print(mohamed)
print(shaymaa+2)
```

Variable Assignment (cont.)

Assign one value to multiple variables

```
mahmoud = zayed = saly = "pass"
print(mahmoud)
print(zayed)
print(saly)
```

Variables and operators

As discussed in Session 01 - Math Operators

```
# Addition
5 + 4
```

```
9
```

```
# power / Exponentiation
2 ** 4
```

```
16
```

```
# Subtraction
2 - 12
```

```
-10
```

```
# Division
5 / 2
```

```
2.5
```

```
# Modulus
18 % 7
```

```
4
```

```
# Multiplication
3 * 27
```

```
81
```

```
# round down division
5 // 2
```

```
2
```

Variables and operators (cont.)

Variable can be used with Math Operators

```
x = 39
y = 74
print(x>y)
```

```
False
```

```
print(x/y)
```

```
0.527027027027027
```

```
print(x+y)
```

```
113
```

```
z = x-y
print(z)
```

```
-35
```

```
pi = 3.14
radius = 5
circumference = 2*pi*radius
print(circumference)
```

```
31.400000000000002
```

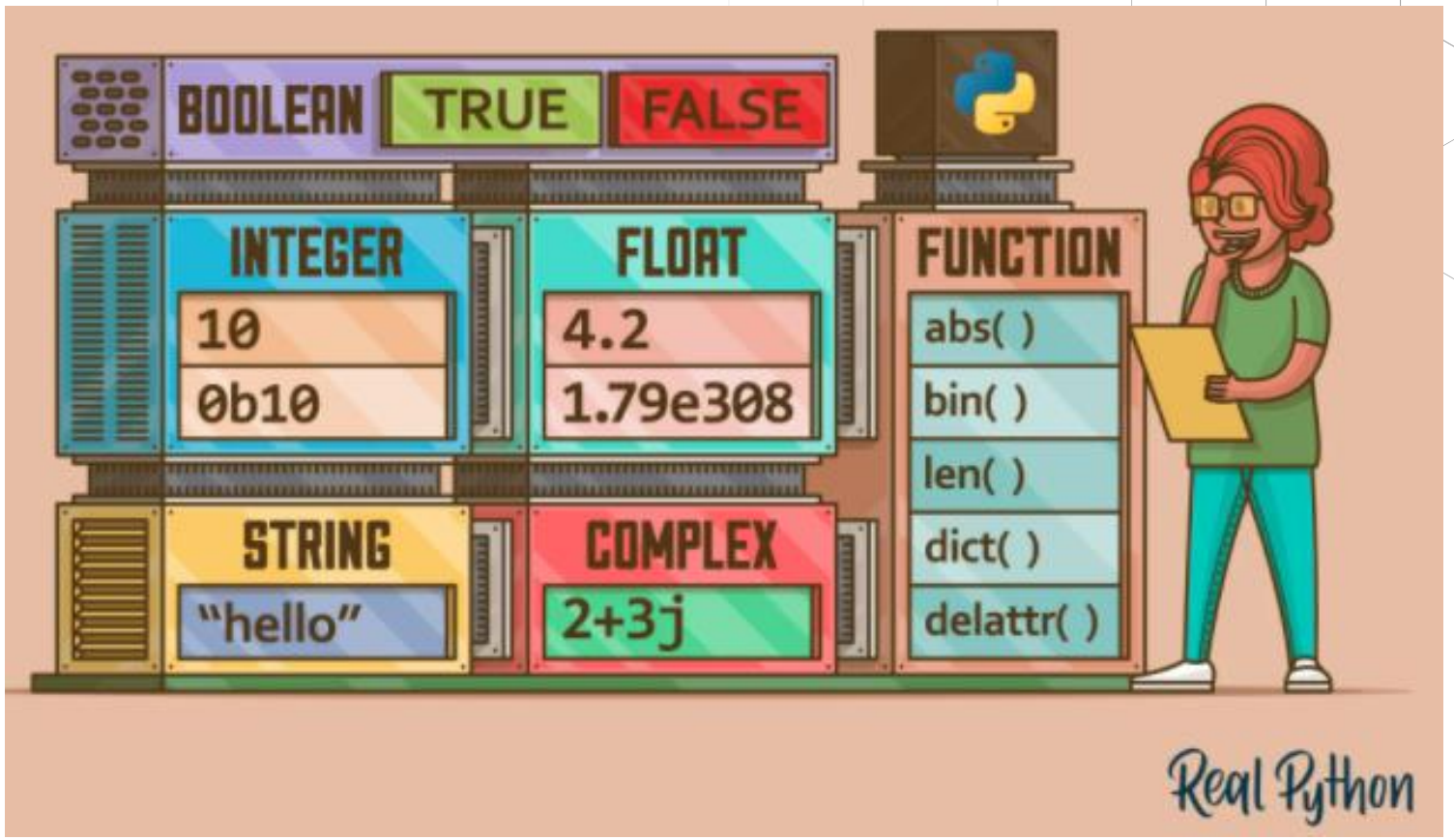
```
print(x)
s = 19
s += x
print(s)
```

```
39
58
```

```
welcome= "Hi"
print(welcome)
```

```
Hi
```

Variable - Types



https://realpython.com/python-data-types/

Variable Types

- A variable is created when you first assign a value to it.
- To get the variable type after you declared it, you can use *type(variable name)*

```
print(type(x))
print(type(y))
print(type(pi))
print(type(welcome))
```

```
<class 'int'>
<class 'int'>
<class 'float'>
<class 'str'>
```

Variable Types – Casting

- You can specify the data type of a variable

```
print(type(x))
x = str(x)
print(type(x))
x = 10
print(type(x))
print(x)
x = float(4)
print(x)
```

```
<class 'int'>
<class 'str'>
<class 'int'>
10
4.0
```

Scalar types

- Integers
- Floats
- Complex
- Boolean

Integer

- Positive or negative whole numbers (without a fractional part) e.g. -10, 10, 456, 4654654.
- In Python 3, there is effectively no limit to how long an integer value can be. Of course, it is constrained by the amount of memory your system has.
- Python interprets a sequence of decimal digits without any prefix to be a decimal number.

```
print(10)
```

```
10
```

Float

- The float type in Python designates a floating-point number.
- Float values are specified with a decimal point.
- Optionally, the character e or E followed by a positive or negative integer may be appended to specify scientific notation

Float (cont.)

```
4.2
```

```
4.2
```

```
type(4.2)
```

```
float
```

```
4.
```

```
4.0
```

```
.2
```

```
0.2
```

```
.4e7
```

```
4000000.0
```

```
type(.4e7)
```

```
float
```

```
4.2e-4
```

```
0.00042
```

Complex

- Complex numbers are specified as

  <real part>+<imaginary part>j

```
2+3j
```

```
(2+3j)
```

```
type(2+3j)
```

```
complex
```

Boolean

Boolean

- Python 3 provides a Boolean data type.
- Objects of Boolean type may have one of two values,
  True or **False**

```
type(True)
```

```
bool
```

```
type(False)
```

```
bool
```

Boolean – (cont.)

```
print(bool("text goes here"))
```

```
True
```

```
print(bool(1))
```

```
True
```

```
print(bool(0))
```

```
False
```

```
print(bool(""))
```

```
False
```

```
print(bool(True))
```

```
True
```

```
print(bool(False))
```

```
False
```

```
print(bool(None))
```

```
False
```

Sequence Type

Strings

String (str)

- Strings are sequences of character data. The string type in Python is called str.

- String literals may be delimited using either single or double quotes. All the characters between the opening delimiter and matching closing delimiter are part of the string.

String (str) – (cont.)

```
print("I am a string.")
```

```
I am a string.
```

```
type("I am a string.")
```

```
str
```

```
print('I am too.')
```

```
I am too.
```

```
type('I am too.')
```

```
str
```

String (str) – (cont.)

- A string in Python can contain as many characters as you wish. The only limit is your machine's memory resources. A string can also be empty:

Escape Sequences in Strings

- Sometimes, you want Python to interpret a character or sequence of characters within a string differently.
- This point discussed in session 01.

\"	Print the next character as a double quote, not a string closer
\'	Print the next character as a single quote, not a string closer
\n	Print a new line character (remember our print statements?)
\t	Print a tab character
\r	Print a carriage return (not used very often)
\$	Print the next character as a dollar, not as part of a variable
\\	Print the next character as a backslash, not an escape character

Raw Strings

- A raw string literal is preceded by **r** or **R**, which specifies that escape sequences in the associated string are not translated. The backslash character is left in the string:

```
print('foo\nbar')
```

```
foo
bar
```

```
print(r'foo\nbar')
```

```
foo\nbar
```

```
print('foo\\bar')
```

```
foo\bar
```

```
print(R'foo\\bar')
```

```
foo\\bar
```

Triple-Quoted Strings

- There is yet another way of delimiting strings in Python. Triple-quoted strings are delimited by matching groups of three single quotes or three double quotes.
- Escape sequences still work in triple-quoted strings, but single quotes, double quotes, and newlines can be included without escaping them.
- This provides a convenient way to create a string with both single and double quotes in it:

Triple-Quoted Strings (cont.)

```
print('''This string has a single (') and a double (") quote.''')
```

```
This string has a single (') and a double (") quote.
```

```
print("""This is a
string that spans
across several lines""")
```

```
This is a
string that spans
across several lines
```

Check Variable Type Using isinstance()

checks if the variable type is true or false

isinstance()

```
message = "welcome"
print(isinstance(message, int))
```

```
False
```

```
x=3
print(isinstance(x,float))
```

```
False
```

```
print(isinstance(x,int))
```

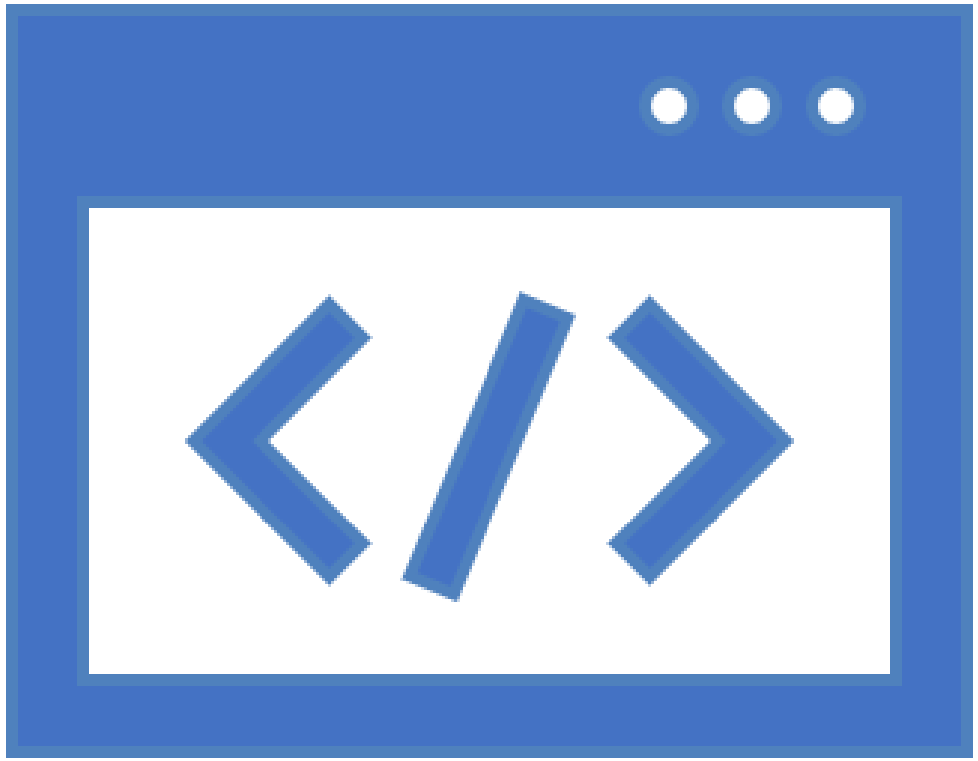
```
True
```

Dynamic Typing

- **Recall type: int, float, str, etc**
  - **Can we change the variable's type from a line to another?**
  - **Does the language check that: before or during running the code?**
- **Python is a dynamically typed language.**
  - **The types of variables are checked during running the program,**
  - **Pros: Easy development**
  - **Cons: More errors at runtime and in shipped code**
  - **Future reading**
- **Other languages are called: statically typed language**
  - **Every variable has a specific type that CAN'T be changed (checked before running)**
  - **Example: C++ and Java**

Input function

- Similar to the print function
- There is an input function()
  - It always reads the input as a string
- 2 ways
  - input() which reads directly
  - input(msg) which prints a message first
- **Let’s demo**



Reading: no printed message

```
2   str = input()              # enter hello world
3   print(str)                 # hello world
4   print(type(str))           # <class 'str'>
5
6   str = input()              # enter 12
7   print(str, type(str))      # 12 <class 'str'>
8
9
```

Reading: with a printed message

```
2
3       str = input('Enter your name: ')     # mostafa
4       print('Hello ' + str)                # print('Hello ' +
5
6       print('Hello ' + input('Enter your name: '))
7
```

Observe

- **Input() function reads a complete line, and return as string**
- **Let’s say you want to read 2 numbers**
  - **Then don’t enter both of them on the same line**
  - **Use 2 input() and input twice (2 lines)**
  - **a = input()**
  - **b = input()**

More convenient way

- Let’s say you want to read 3 strings from a single line, you can use this line of code (later you will understand)
  - **a, b, c = input().split()**
  - **Then: a, b, c are 3 strings**
- Let’s read 4 integers:
  - **a, b, c, d = map(int, input().split())**
  - **But you must enter really 4 integers**
- Let’s read 5 floats
  - **a, b, c, d, e = map(float, input('Enter 5 numbers: ').split())**
- Follow this syntax style for now

Type casting

1. Type Conversion is the conversion of object from one data type to another data type.
2. Implicit Type Conversion is automatically performed by the Python interpreter.
3. Python avoids the loss of data in Implicit Type Conversion.
4. Explicit Type Conversion is also called Type Casting, the data types of objects are converted using predefined functions by the user.
5. In Type Casting, loss of data may occur as we enforce the object to specific data type.

Implicit Type Conversion

- In Implicit type conversion, Python automatically converts one data type to another data type. This process doesn't need any user involvement.

```
num_int = 123
num_flo = 1.23

num_new = num_int + num_flo

print("datatype of num_int:",type(num_int))
print("datatype of num_flo:",type(num_flo))

print("Value of num_new:",num_new)
print("datatype of num_new:",type(num_new))
```

Explicit Type Conversion

- **Example 2: Addition of string(higher) data type and integer(lower) datatype**

```
num_int = 123
num_str = "456"

print("Data type of num_int:",type(num_int))
print("Data type of num_str before Type Casting:",type(num_str))

num_str = int(num_str)
print("Data type of num_str after Type Casting:",type(num_str))

num_sum = num_int + num_str

print("Sum of num_int and num_str:",num_sum)
print("Data type of the sum:",type(num_sum))
```

- **Example 2: Addition of string(higher) data type and integer(lower) datatype**

```
num_int = 123
num_str = "456"

print("Data type of num_int:",type(num_int))
print("Data type of num_str before Type Casting:",type(num_str))

num_str = int(num_str)
print("Data type of num_str after Type Casting:",type(num_str))

num_sum = num_int + num_str

print("Sum of num_int and num_str:",num_sum)
print("Data type of the sum:",type(num_sum))
```

Explicit Type Conversion

- **Converting Integers to Floats**

THANKS!

Session 2

Session Content

- What’s a Data Structure ?
- Lists
- Tuples
- Sets
- Dictionaries
- Homework
- References & Tools

Session Objective

Learn what is and how to use Data Structures

What’s a Data Structure ?

- A data organization, management, and storage format that enables efficient access and modification
- A collection of data values, the relationships among them, and the functions or operations that can be applied to the data

List

```
mylist=["apple","banana", "cherry"]
```

```
mylist
```

```
['apple', 'banana', 'cherry']
```

Tuple

```
mytuple=("apple","banana", "cherry")
```

```
mytuple
```

```
('apple', 'banana', 'cherry')
```

Set

```
myset={"apple","banana", "cherry"}
```

```
myset
```

```
{'apple', 'banana', 'cherry'}
```


Lists

First type of Data Structure

Let's create a variable called numbers; it's assigned with not just one number but is filled with a list consisting of five values

(note: the list starts with an open square bracket and ends with a closed square bracket; the space between the brackets is filled with five numbers separated by commas).

A list is one of the most common and basic data structures in Python.

```
mylist=["apple","banana", "cherry"]
```

```
mylist
```

```
['apple', 'banana', 'cherry']
```

Lists

Lists can contain any mix and match of the data types you have seen so far

All ordered containers (like lists) are indexed in python using a starting index of 0

```
list1 = [1,2,3.6,'Mostafa' ,False]
```

```
print(list1)
```

```
[1, 2, 3.6, 'Mostafa', False]
```

```
list1[0]
```

```
1
```

```
list1[2]
```

```
3.6
```

```
list1[3]
```

```
'Mostafa'
```

Retrieving Element

You can retrieve the last element by reducing the index by 1

```
list1 = [1,2,3.6,'Mostafa' ,False]
```

```
list1[len(list1)-1]
```

```
False
```

You can index from the end of a list by using negative values, where -1 is the last element, -2 is the second to last element and so on.

```
list1[len(list1)-1]
```

```
False
```

```
list1[len(list1)-2]
```

```
'Mostafa'
```

```
list1[len(list1)-3]
```

```
3.6
```

Slicing

You saw that we can pull more than one value from a list at a time by using slicing

```
list1 = [1,2,3.6,'Mostafa' ,False]
```

```
list1[1:4]
```

```
[2, 3.6, 'Mostafa']
```

```
list1[:4]
```

```
[1, 2, 3.6, 'Mostafa']
```

If you know that you want to start at the beginning

```
list1[2:]
```

```
[3.6, 'Mostafa', False]
```

Return all the elements to the end of the list

Slicing (cont.)

You saw that we can pull all elements in the list

```
list1 = [1,2,3.6,'Mostafa' ,False]
```

```
list1[:]
```

```
[1, 2, 3.6, 'Mostafa', False]
```

in / not in

You saw that we can also use in and not in to return a bool of whether an element exists within our list

```
list1 = [1,2,3.6,'Mostafa' ,False]
```

```
3.6 in list1
```

```
True
```

```
'Mostafa' in list1
```

```
True
```

```
'st' in list1      #consider 'st' is an element
```

```
False
```

```
'st' not in list1
```

```
True
```

```
100 not in list1
```

```
True
```

Order

Order is about whether the position of an element in the object can be used to access the element. **Both strings and lists are ordered,** We can use the order to access parts of a list and string.

```
greeting = "Hello there"
```

```
print(greeting[0])
```

```
H
```

```
list1 = [1,2,3.6,'Mostafa' ,False]
```

```
print(list1[0])
```

```
1
```

Methods – len()

len() returns how many elements are in a list

```
list=[0,2,3,4,7]
```

```
len(list)
```

```
5
```

Methods – max()

max() returns the greatest element of the list

```
list=[0,2,3,4,7,1]
```

```
max(list)
```

```
7
```

Methods – min()

min() returns the smallest element in a list

```
list=[-1,-5,0,2,3,4,7]
```

```
min(list)
```

```
-5
```

Methods – append()

append() method : A helpful method called append adds an element to the end of a list <<list.append(item/element)>>

```
list=[-1,-5,0,2,3,2,4,7]
```

```
list.append(55)
```

```
list
```

```
[-1, -5, 0, 2, 3, 2, 4, 7, 55]
```

Append **one element**

```
list1=[-1,-5,0,2,3,2,4,7]
list2=[20,30]
```

```
list1.append(list2)
```

```
list1
```

```
[-1, -5, 0, 2, 3, 2, 4, 7, [20, 30]]
```

Append **list** as an element

Methods – extend()

extend() adding all items of a list (passed as an argument) to the end of the list <<list1.extend(list2)>>

```
list1=[-1,-5,0,2,3,2,4,7]
list2=[20,30]

list1.append(list2)

list1
```

```
[-1, -5, 0, 2, 3, 2, 4, 7, [20, 30]]
```

append()

```
list1=[-1,-5,0,2,3,2,4,7]
list2=[20,30]

list1.extend(list2)

list1
```

```
[-1, -5, 0, 2, 3, 2, 4, 7, 20, 30]
```

extend()

Methods – sorted()

sorted() returns a copy of a list in order from smallest to largest, leaving the list unchanged

```
list=[-1,-5,0,2,3,2,4,7]
```

```
sorted(list)
```

```
[-5, -1, 0, 2, 2, 3, 4, 7]
```

Methods – sort()

By default, Python sorts

- **strings in alphabetical order**
- **numbers in ascending numerical order**

```
ranks = ["kingdom","phylum", "class", "order", "family"]
print("at the start : " + str(ranks))
ranks.sort()
print("after sorting : " + str(ranks))
```

```
at the start : ['kingdom', 'phylum', 'class', 'order', 'family']
after sorting : ['class', 'family', 'kingdom', 'order', 'phylum']
```

Methods – reverse()

reverse() reverses objects of list in place.

```
ranks = ["kingdom","phylum", "class", "order", "family"]
print("at the start : " + str(ranks))
ranks.sort()
print("after sorting : " + str(ranks))
ranks.reverse()
print("sorting descending : " + str(ranks))
```

```
at the start : ['kingdom', 'phylum', 'class', 'order', 'family']
after sorting : ['class', 'family', 'kingdom', 'order', 'phylum']
sorting descending : ['phylum', 'order', 'kingdom', 'family', 'class']
```

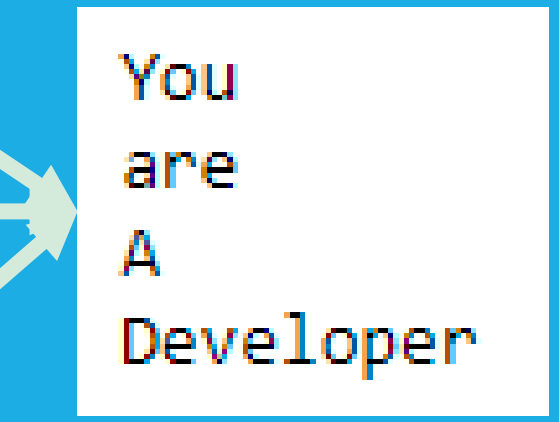
Methods – join()

join() method : Join is a **string method** that takes a list of strings as an argument, and returns a string consisting of the list elements joined by a separator string

```
print("You", "\nare", "\nA", "\nDeveloper")
```

```
new_str="\n".join(["You", "are", "A", "Developer"])
print(new_str)
```

```
list=["You", "are", "A", "Developer"]
new_str="\n".join(list)
print(new_str)
```



Methods – join() (cont.)

```
list=["You", "are", "A", "Developer"]
new_str="-".join(list)
print(new_str)
```

```
You-are-A-Developer
```

Methods – split()

split() method which works on strings.
split() takes a single argument, called the **delimiter**, and splits the original string wherever it sees the delimiter, producing a list.

```
names = "melanogaster,simulans,yakuba,ananassae"
species = names.split(",")
print (species)
```

```
['melanogaster', 'simulans', 'yakuba', 'ananassae']
```

Note

- When we treat a **string** as a list, **each character** becomes an individual element,

But

- When we treat a **file** as a list, **each line** becomes an individual element.

NEGATIVE INDEXING

- Use negative indexes to start the slice from the end of the sequence

```
        #  -7  -6 -5 -4 -3 -2 -1    # 7 + neg_pos
my_list = [0, 1, 2, 3, 4, 5, 6]

ln = len(my_list)

print(my_list[ln-1])    # 6 = last number
print(my_list[ln-2])    # 5 = 2nd last number

# Negative indexing
print(my_list[-1])    # 6 = last number
print(my_list[-2])    # 5 = 2nd last number

print(my_list.pop(-1))  # 6
print(my_list.pop(-1))  # 5

#my_list: [0, 1, 2, 3, 4]

```

SLICING WITH -VE INDEXING

```
3           #-9 -8 -7 -6 -5 -4 -3 -2 -1    # 9 + neg_pos
4   my_list = [0, 1, 2, 3, 4, 5, 6, 7, 8]
5
6
7   sub_list = my_list[3:7]      # 3 4 5 6
8   # we can rewrite by finding the matched -ve indices
9   sub_list = my_list[-6:-2]    # 3 4 5 6
10  sub_list = my_list[-6:7]     # 3 4 5 6
11  sub_list = my_list[3:-2]     # 3 4 5 6
12
13  # observe: -6 < -2
14  #sub_list = my_list[-2:-6]   # Empty list!
15
16
```

SLICING WITH -VE INDEXING AND -VE STEP

```
2
3           #-9 -8 -7 -6 -5 -4 -3 -2 -1    # 9 + neg_pos
4   my_list = [0, 1, 2, 3, 4, 5, 6, 7, 8]
5
6   sub_list = my_list[1:8:1]     # 1 2 3 4 5 6 7
7
8   sub_list = my_list[-8:-2:1]    # 1 2 3 4 5 6
9   sub_list = my_list[-2:-8:-1]   # 7 6 5 4 3 2
10
11
```

LIST COMPREHENSION

LIST COMPREHENSION

- Elegant syntax to make more readable / short code
- It doesn’t solve a new programming problem!
- Just new syntax for what we can already solve with available tools!
- new_list = [expression **for** member **in** iterable]

From old to new syntax

```
2
3   lst1 = [2, 3, 4, 1]
4
5   # Old syntax
6   lst2 = []
7   for i in lst1:
8       lst2.append(i *i + 1)
9
10  print(lst2)      # [5, 10, 17, 2]
11
12  # new syntax
13  lst2 = [i*i+1    for i in lst1]
14  print(lst2)      # [5, 10, 17, 2]
15
```

List comprehension

```
2
3   # new_list = [expression  for member in iterable  ]
4
5   lst1 = [2, 3, 4, 1]
6
7   lst2 = [i*i+1   for i in lst1]
8   print(lst2)     # [5, 10, 17, 2]
9
10  lst3 = [n+1 for n in range(5, 9)]
11  print(lst3)     # [6, 7, 8, 9]
12
13  lst4 = [3*char for char in 'Hey']
14  print(lst4)     # ['HHH', 'eee', 'yyy']
15
```

From old to new syntax: conditional

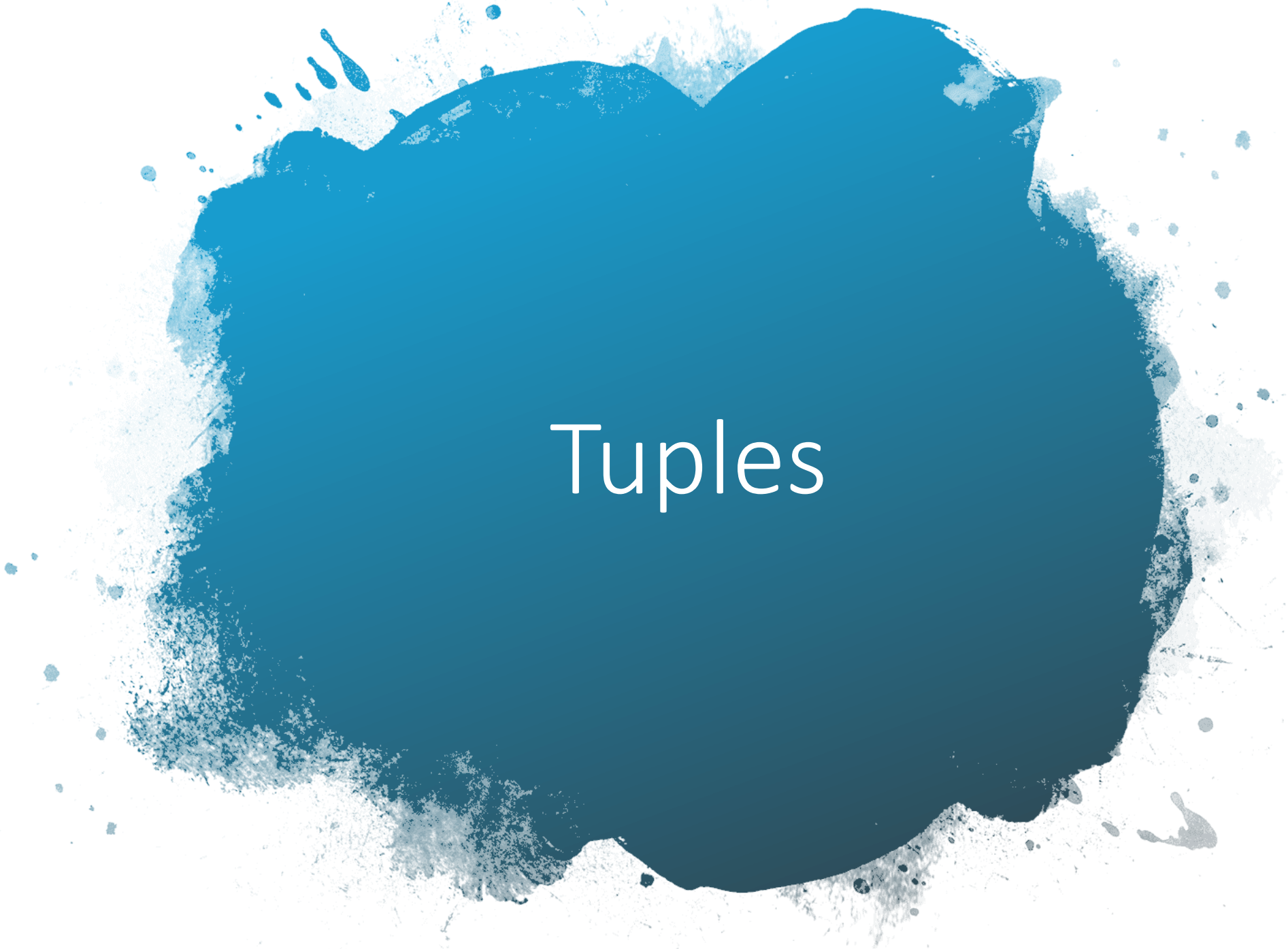
```
1
2   lst1 = [1, -2, 6, -3, 2, -6]
3
4   # Old syntax
5   lst2 = []
6   for n in lst1:
7       if n > 0:
8           lst2.append(n)
9
10  print(lst2)      # [1, 6, 2]
11
12  # New syntax
13  lst3 = [n for n in lst1   if n > 0]
14  print(lst3)      # [1, 6, 2]
15
16
```

Conditional list comprehension

```
4
5   lst = [1, -2, 6, -3, 2, -6]
6   lst3 = [n for n in lst   if n > 0]
7   print(lst3)     # [1, 6, 2]
8
9   lst4 = [n for n in lst   if n % 2 == 0]
10  print(lst4)     # [-2, 6, 2, -6]
11
12  lst5 = [n for n in lst   if n % 2 == 0 and n % 3 == 0]
13  print(lst5)     # [6, -6]
14
15  lst6 = [n for n in lst   if n % 2 == 0 if n % 3 == 0]
16  print(lst6)     # [6, -6]
17
18  sentence = 'Glad that you took this course!'
19  vowels = [i for i in sentence if i in 'aeiou']
20  print(vowels)   # ['a', 'a', 'o', 'u', 'o', 'o', 'i', 'o', 'u', 'e']
21
22
```

List comprehension: filter and transform

```
2
3   lst = [1, -2, 6, -3, 2, -6]
4
5   def sq(i):
6       return i * i
7
8   def is_even(i):
9       return i % 2 == 0
10
11  lst5 = [sq(n) for n in lst   if is_even(n)]
12  print(lst5)      # [4, 36, 4, 36]
13
14
15  # we call if is_even(n) : filter
16  # we call sq(n)         : transform
17
```


Tuples

Second type of Data Structure

Tuples are used to store multiple items in a *single variable*.

A tuple is a collection which is ordered and **unchangeable**.

Tuples are written with *round brackets*.

```
mytuple=("apple","banana", "cherry")
```

```
mytuple
```

```
('apple', 'banana', 'cherry')
```

Tuples

A tuple can contain different data types

```
tuple1=("abc", 34, True, 40, "male")
```

Tuple item are indexed

the first item has index [0],

the second item has index [1] etc.

```
tuple1[0]
```

```
'abc'
```

```
tuple1[3]
```

```
40
```

Tuples

You can retrieve the last element by reducing the index by 1

```
tuple1[len(tuple1)-1]
```

```
'male'
```

You can index from the end of a tuple by using negative values, where -1 is the last element, -2 is the second to last element and so on.

```
tuple1[-1]
```

```
'male'
```

```
tuple1[-2]
```

```
40
```

Tuples - Slicing

You saw that we can pull more than one value from a tuple at a time by using slicing

```
tuple1[1:3]
```

```
(34, True)
```

```
tuple1[:2]
```

```
('abc', 34)
```

If you know that you want to start at the beginning

```
tuple1[1:]
```

```
(34, True, 40, 'male')
```

Return all the elements to the end of the list

in / not in

You saw that we can also use in and not in to return a bool of whether an element exists within our tuple

```
tuple1=("abc", 34, True, 40, "male")
```

```
34 in tuple1
```

```
True
```

```
54 in tuple1
```

```
False
```

```
54 not in tuple1
```

```
True
```

Mutability

Mutability is about whether we can change an object once it has been created. If an object (like a list or string) can be changed (like a list can), then it is called **mutable**. However, if an object cannot be changed with creating a completely new object (like strings), then the object is considered **immutable**.

```
list1=["abc", 34, True, 40, "male"]
```

```
list1[0]="one"
list1
```

```
['one', 34, True, 40, 'male']
```

you can replace 1 with 'one' in the above list. This is because lists are mutable.

```
tuple1=("abc", 34, True, 40, "male")
```

```
tuple1[0]="one"
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-22-f3f8871a37c5> in <module>()
----> 1 tuple1[0]="one"

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

This is because strings are immutable. This means to change this string; you will need to create a completely new string.

Order

Order is about whether the position of an element in the object can be used to access the element. **Both strings and lists are ordered,** We can use the order to access parts of a list and string.

```
tuple1=("abc", 34, True, 40, "male")
```

```
tuple1[0]
```

```
'abc'
```

Tuple

```
greeting="Hello there"
```

```
greeting[0]
```

```
'H'
```

String

Methods – len()

len() returns **how many** elements are in a tuple

```
tuple1=(4,5,6,2,3,7)
```

```
len(tuple1)
```

```
6
```

Methods – max()

max() returns the **greatest** element of the tuple

```
tuple1=(4,5,6,2,3,7)
```

```
max(tuple1)
```

```
7
```

Methods – min()

min() returns the **smallest** element in a tuple

```
tuple1=(4,5,6,2,3,7)
```

```
min(tuple1)
```

```
2
```

Methods – append()

tuples in Python are **immutable**, so you **cannot append** variables to a tuple once it is created.

```
tuple1=(4,5,6,2,3,7)
tuple1.append(10)
```

```
---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)
<ipython-input-8-17155099b063> in <module>()
      1 tuple1=(4,5,6,2,3,7)
----> 2 tuple1.append(10)

AttributeError: 'tuple' object has no attribute 'append'
```

Methods – append() (cont.)

Append to tuples in Python

workaround

```
tuple1=(4,5,6,2,3,7)
list1=list(tuple1)
```

```
list1.append(10)
```

```
tuple1=tuple(list1)
```

```
tuple1
```

```
(4, 5, 6, 2, 3, 7, 10)
```

Methods – extend()

tuples in Python are **immutable**, so you **cannot extend** variables to a tuple once it is created.

```
tuple1=(4,5,6,2,3,7)
```

```
tuple2=(10,12)
```

```
tuple1.extend(tuple2)
```

```
---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)
<ipython-input-15-6c9810f40d34> in <module>()
----> 1 tuple1.extend(tuple2)

AttributeError: 'tuple' object has no attribute 'extend'
```

Methods – extend() (cont.)

extend tuple in Python

workaround

```
tuple1=(4,5,6,2,3,7)
```

```
tuple2=(10,12)
```

```
list1=list(tuple1)
list2=list(tuple2)
```

```
list1
```

```
[4, 5, 6, 2, 3, 7]
```

```
list2
```

```
[10, 12]
```

```
list1.extend(list2)
```

```
list1
```

```
[4, 5, 6, 2, 3, 7, 10, 12]
```

```
tuple1=tuple(list1)
```

```
tuple1
```

```
(4, 5, 6, 2, 3, 7, 10, 12)
```

Methods – sorted()

sorted() returns a copy of a tuple in order from smallest to largest, leaving the tuple unchanged

```
tuple1=(4,5,6,2,3,7)
```

```
sorted(tuple1)
```

```
[2, 3, 4, 5, 6, 7]
```

```
tuple1
```

```
(4, 5, 6, 2, 3, 7)
```

Methods – sort()

A tuple is a data type for immutable ordered sequences of elements. To sort elements of a tuple, we can use the sorted function, providing the tuple as the first argument. This function returns a sorted list from the given iterable, and we can easily convert this list into a tuple using the built-in function tuple

```
ranks = ("kingdom","phylum", "class", "order", "family")
print("at the start : " + str(ranks))
ranks.sort()
print("after sorting : " + str(ranks))
```

```
at the start : ('kingdom', 'phylum', 'class', 'order', 'family')

---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)
<ipython-input-37-8e69149b5a9a> in <module>()
      1 ranks = ("kingdom","phylum", "class", "order", "family")
      2 print("at the start : " + str(ranks))
----> 3 ranks.sort()
      4 print("after sorting : " + str(ranks))

AttributeError: 'tuple' object has no attribute 'sort'
```

Methods – sort() (cont.)

Sort tuple in Python **workaround**

```
ranks = ("kingdom","phylum", "class", "order", "family")
sorted(ranks)
```

```
['class', 'family', 'kingdom', 'order', 'phylum']
```

```
ranks_s=sorted(ranks)
```

```
ranks
```

```
('kingdom', 'phylum', 'class', 'order', 'family')
```

```
ranks_s
```

```
['class', 'family', 'kingdom', 'order', 'phylum']
```

Methods – reverse()

Since tuples are immutable, there is no way to reverse a tuple in-place. Creating a copy requires more space to hold all of the existing elements.

```
ranks = ("kingdom","phylum", "class", "order", "family")
print("at the start : " + str(ranks))
ranks_s=sorted(ranks)
print("after sorting : " + str(ranks))
ranks.reverse()
print("sorting descending : " + str(ranks))
```

```
at the start : ('kingdom', 'phylum', 'class', 'order', 'family')
after sorting : ('kingdom', 'phylum', 'class', 'order', 'family')

---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)
<ipython-input-63-ed1a9605f776> in <module>()
      3 ranks_s=sorted(ranks)
      4 print("after sorting : " + str(ranks))
----> 5 ranks.reverse()
      6 print("sorting descending : " + str(ranks))

AttributeError: 'tuple' object has no attribute 'reverse'
```

Methods – reverse() (cont.)

reverse tuple in Python

workaround

```
ranks = ("kingdom","phylum", "class", "order", "family")
print("at the start : " + str(ranks))
ranks_s=sorted(ranks)
print("after sorting : " + str(ranks_s))
ranks_s_list=list(ranks_s)
ranks_s_list.reverse()
#ranks_s_list
ranks_s_list_t=tuple(ranks_s_list)
ranks_s_list_t
```

```
at the start : ('kingdom', 'phylum', 'class', 'order', 'family')
after sorting : ['class', 'family', 'kingdom', 'order', 'phylum']

('phylum', 'order', 'kingdom', 'family', 'class')
```

Methods – join()

join() method : Join is a **string method** that takes a tuple of strings as an argument, and returns a string consisting of the list elements joined by a separator string

```
tuple=("You", "are", "A", "Developer")
new_str="-".join(tuple)
print(new_str)
```

```
You-are-A-Developer
```

Methods – split()

split() method which works on strings.
split() takes a single argument, called the **delimiter**, and splits the original string wherever it sees the delimiter, producing a list.

```
names = "melanogaster,simulans,yakuba,ananassae"
species = names.split(",")
print (species)
```

```
['melanogaster', 'simulans', 'yakuba', 'ananassae']
```


Sets

Third type of Data Structure

Sets are used to store multiple items in a single variable.

A set is a collection which is both unordered and unindexed.

Sets are written with curly brackets.

```
myset={"apple","banana", "cherry"}
```

```
myset
```

```
{'apple', 'banana', 'cherry'}
```

Sets

A set can contain different data types

```
set1={"abc",34,True,40,"male"}
```

Set items cannot be referred to by index or key

```
set2={"apple","banana","cherry"}
set2[1]
```

```
---------------------------------------------------------------------------
TypeError
<ipython-input-9-8076e5a92284> in <module>()
      1 set2={"apple","banana","cherry"}
----> 2 set2[1]

TypeError: 'set' object does not support indexing
```

in / not in

in and not in used to return a bool of whether an element exists within our set

```
myset={"apple","banana", "cherry"}
```

```
"banana" in myset
```

```
True
```

```
"orange" in myset
```

```
False
```

```
"orange" not in myset
```

```
True
```

Methods – len()

len() returns **how many** elements are in a set

```
myset={"apple","banana", "cherry"}
```

```
len(myset)
```

```
3
```

Methods – max()

max() returns the **greatest** element of the set

```
set1={"avocado","apple","banana", "orange", "cherry"}
```

```
max(set1)
```

```
'orange'
```

```
set2={3,7,23,1,6,8}
```

```
max(set2)
```

```
23
```

Methods – min()

min() returns the **smallest** element in a set

```
set1={"avocado","apple","banana", "orange", "cherry"}
```

```
min(set1)
```

```
'apple'
```

```
set2={3,7,23,1,6,8}
```

```
min(set2)
```

```
1
```

Methods – sorted()

sorted() returns a copy of a set in order from smallest to largest, leaving the set unchanged

```
set1={"avocado","apple","banana", "orange", "cherry"}
```

```
sorted(set1)
```

```
['apple', 'avocado', 'banana', 'cherry', 'orange']
```

```
set2={3,7,23,1,6,8}
```

```
sorted(set2)
```

```
[1, 3, 6, 7, 8, 23]
```

Dictionaries

https://realpython.com/python-dicts/

Fourth type of Data Structure

- Dictionaries are Python's implementation of a data structure that is more generally known as an associative array.
- A dictionary consists of a collection of key-value pairs. Each key-value pair maps the key to its associated value.
- You can define a dictionary by enclosing a comma-separated list of key-value pairs in curly braces ({}). A colon (:) separates each key from its associated value:

```
d = {
    <key>: <value>,
    <key>: <value>,
      .
      .
      .
    <key>: <value>
}
```

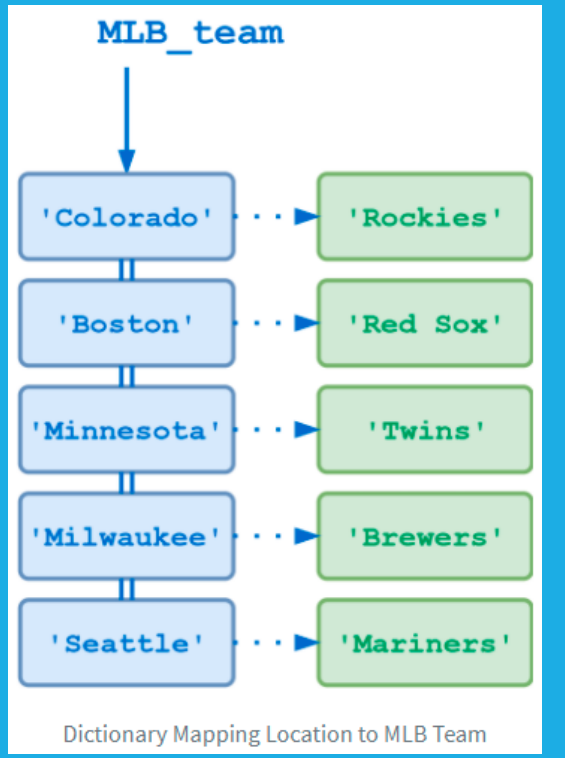
Dictionary Example

The following defines a dictionary that maps a location to the name of its corresponding Major League Baseball team:

```
MLB_team = {
      'Colorado' : 'Rockies',
      'Boston'   : 'Red Sox',
      'Minnesota': 'Twins',
      'Milwaukee': 'Brewers',
      'Seattle'  : 'Mariners'
}
```

```
MLB_team
```

```
{'Colorado': 'Rockies',
 'Boston': 'Red Sox',
 'Minnesota': 'Twins',
 'Milwaukee': 'Brewers',
 'Seattle': 'Mariners'}
```



Dictionary Mapping Location to MLB Team

Construct a dictionary with the built-in dict() function

The argument to **dict()** should be a sequence of key-value pairs.

A list of tuples works well for this:

```
MLB_team = dict([
      ('Colorado', 'Rockies'),
      ('Boston', 'Red Sox'),
      ('Minnesota', 'Twins'),
      ('Milwaukee', 'Brewers'),
      ('Seattle', 'Mariners')
 ])
```

```
MLB_team
```

```
{'Colorado': 'Rockies',
 'Boston': 'Red Sox',
 'Minnesota': 'Twins',
 'Milwaukee': 'Brewers',
 'Seattle': 'Mariners'}
```

Accessing Dictionary Values

The entries in the dictionary display in the order they were defined. But that is irrelevant when it comes to retrieving them.

Dictionary elements **are not accessed by numerical index**:

```
MLB_team = dict([
      ('Colorado', 'Rockies'),
      ('Boston', 'Red Sox'),
      ('Minnesota', 'Twins'),
      ('Milwaukee', 'Brewers'),
      ('Seattle', 'Mariners')
 ])
```

```
MLB_team[1]
```

```
---------------------------------------------------------------------------
KeyError
<ipython-input-90-38b5ba4ee407> in <module>()
----> 1 MLB_team[1]

KeyError: 1
```

Accessing Dictionary Values (cont.)

- A value is retrieved from a dictionary by specifying its corresponding key in square brackets ([ ]):

- If you refer to a key that is not in the dictionary, Python raises an exception:

```
MLB_team = dict([
     ('Colorado', 'Rockies'),
     ('Boston', 'Red Sox'),
     ('Minnesota', 'Twins'),
     ('Milwaukee', 'Brewers'),
     ('Seattle', 'Mariners')
 ])
```

```
MLB_team['Minnesota']
```

```
'Twins'
```

```
MLB_team['Colorado']
```

```
'Rockies'
```

```
MLB_team['Toronto']
```

```
---------------------------------------------------------------
KeyError
<ipython-input-100-0cf457503e5d> in <module>()
----> 1 MLB_team['Toronto']

KeyError: 'Toronto'
```

Adding an entry to an existing dictionary

- Adding an entry to an existing dictionary is simply a matter of assigning a new key and value:

```
MLB_team = dict([
     ('Colorado', 'Rockies'),
     ('Boston', 'Red Sox'),
     ('Minnesota', 'Twins'),
     ('Milwaukee', 'Brewers'),
     ('Seattle', 'Mariners')
 ])
```

```
MLB_team['Kansas City'] = 'Royals'
MLB_team
```

```
{'Colorado': 'Rockies',
 'Boston': 'Red Sox',
 'Minnesota': 'Twins',
 'Milwaukee': 'Brewers',
 'Seattle': 'Mariners',
 'Kansas City': 'Royals'}
```

Delete from dictionary

- To delete an entry, use the **del** statement, specifying the key to delete:

```
MLB_team = dict([
      ('Colorado', 'Rockies'),
      ('Boston', 'Red Sox'),
      ('Minnesota', 'Twins'),
      ('Milwaukee', 'Brewers'),
      ('Seattle', 'Mariners')
 ])
```

```
del MLB_team['Seattle']
MLB_team
```

```
{'Colorado': 'Rockies',
 'Boston': 'Red Sox',
 'Minnesota': 'Twins',
 'Milwaukee': 'Brewers'}
```

Building a Dictionary Incrementally

- Defining a dictionary using curly braces and a list of key-value pairs, as shown above, is fine if you know all the keys and values in advance. But what if you want to build a dictionary on the fly?
- You can start by creating an empty dictionary, which is specified by empty curly braces. Then you can add new keys and values one at a time:

```
person = {}
type(person)
```

```
dict
```

```
person['fname'] = 'Mostafa'
person['lname'] = 'Ali'
person['age'] = 45
person['spouse'] = 'Mariam'
person['children'] = ['Ahmed', 'Adel', 'Khalid']
person['pets'] = {'dog': 'Fido', 'cat': 'Sox'}
```

```
person
```

```
{'fname': 'Mostafa',
 'lname': 'Ali',
 'age': 45,
 'spouse': 'Mariam',
 'children': ['Ahmed', 'Adel', 'Khalid'],
 'pets': {'dog': 'Fido', 'cat': 'Sox'}}
```

Building a Dictionary Incrementally (cont.)

- Once the dictionary is created in this way, its values are accessed the same way as any other dictionary:

```
person
```

```
{'fname': 'Mostafa',
 'lname': 'Ali',
 'age': 45,
 'spouse': 'Mariam',
 'children': ['Ahmed', 'Adel', 'Khalid'],
 'pets': {'dog': 'Fido', 'cat': 'Sox'}}
```

```
person['fname']
```

```
'Mostafa'
```

```
person['children']
```

```
['Ahmed', 'Adel', 'Khalid']
```

Building a Dictionary Incrementally (cont.)

- **Retrieving** the values in the sub-list or sub-dictionary requires an additional index or key:

```
person
```

```
{'fname': 'Mostafa',
 'lname': 'Ali',
 'age': 45,
 'spouse': 'Mariam',
 'children': ['Ahmed', 'Adel', 'Khalid'],
 'pets': {'dog': 'Fido', 'cat': 'Sox'}}
```

```
person['children'][-1]
```

```
'Khalid'
```

```
person['pets']['cat']
```

```
'Sox'
```

Restrictions on Dictionary

Keys

- Duplicate keys are not allowed.
- A dictionary maps each key to a corresponding value, so it doesn't make sense to map a particular key more than once.

- A dictionary key must be of a type that is immutable.
- A tuple can also be a dictionary key, because tuples are immutable:

Values

- Literally none at all.
- A dictionary value can be any type of object Python supports, including mutable types like lists and dictionaries, and user-defined objects.

Restrictions on Dictionary Keys (1)

- Duplicate keys are not allowed.
- A dictionary maps each key to a corresponding value, so it doesn't make sense to map a particular key more than once.

```
MLB_team = {
     'Colorado' : 'Rockies',
     'Boston'   : 'Red Sox',
     'Minnesota': 'Timberwolves',
     'Milwaukee': 'Brewers',
     'Seattle'  : 'Mariners',
     'Minnesota': 'Twins'
}
MLB_team
```

```
{'Colorado': 'Rockies',
 'Boston': 'Red Sox',
 'Minnesota': 'Twins',
 'Milwaukee': 'Brewers',
 'Seattle': 'Mariners'}
```

```
MLB_team = {
     'Colorado' : 'Rockies',
     'Boston'   : 'Red Sox',
     'Minnesota': 'Twins',
     'Milwaukee': 'Brewers',
      'Seattle'  : 'Mariners'
}

MLB_team['Minnesota'] = 'Timberwolves'
MLB_team
```

```
{'Colorado': 'Rockies',
 'Boston': 'Red Sox',
 'Minnesota': 'Timberwolves',
 'Milwaukee': 'Brewers',
 'Seattle': 'Mariners'}
```

- Similarly, if you specify a key a second time during the initial creation of a dictionary, the second occurrence will override the first:

Restrictions on Dictionary Keys (2)

- A dictionary key must be of a type that is immutable.
- A tuple can also be a dictionary key, because tuples are immutable:

```
d = {(1, 1): 'a', (1, 2): 'b', (2, 1): 'c', (2, 2): 'd'}
```

```
d[(1,1)]
```

```
'a'
```

```
d[(2,1)]
```

```
'c'
```

Restrictions on Dictionary Values

- Literally none at all.
- A dictionary value can be any type of object Python supports, including mutable types like lists and dictionaries, and user-defined objects.

- There is also no restriction against a particular value appearing in a dictionary multiple times:

```
d = {0: 'a', 1: 'a', 2: 'a', 3: 'a'}
d
```

```
{0: 'a', 1: 'a', 2: 'a', 3: 'a'}
```

```
d[0] == d[1] == d[2]
```

```
True
```

in / not in

- in / not in operators return True or False according to whether the specified operand occurs as a key in the dictionary:

```
MLB_team = {
     'Colorado' : 'Rockies',
     'Boston'   : 'Red Sox',
     'Minnesota': 'Twins',
     'Milwaukee': 'Brewers',
     'Seattle'  : 'Mariners'
 }
```

```
'Milwaukee' in MLB_team
```

```
True
```

```
'Toronto' in MLB_team
```

```
False
```

```
'Toronto' not in MLB_team
```

```
True
```

Methods – len()

- The len() function returns the number of key-value pairs in a dictionary:

```
MLB_team = {
...      'Colorado' : 'Rockies',
...      'Boston'   : 'Red Sox',
...      'Minnesota': 'Twins',
...      'Milwaukee': 'Brewers',
...      'Seattle'  : 'Mariners'
... }
```

```
len(MLB_team)
```

```
5
```

Methods – .get()

- Returns the value for a key if it exists in the dictionary.
- d.get(<key>) searches dictionary d for <key> and returns the associated value if it is found. If <key> is not found, it returns **None** (error is not raise)

```
d = {'a': 10, 'b': 20, 'c': 30}
```

```
print(d.get('b'))
```

```
20
```

```
print(d.get('z'))
```

```
None
```

Methods – d.items()

- Returns a list of key-value pairs in a dictionary.
- d.items() returns a list of tuples containing the key-value pairs in d. The first item in each tuple is the key, and the second item is the key's value:

```
d = {'a': 10, 'b': 20, 'c': 30}
d
```

```
{'a': 10, 'b': 20, 'c': 30}
```

```
list(d.items())
```

```
[('a', 10), ('b', 20), ('c', 30)]
```

```
list(d.items())[1][0]
```

```
'b'
```

```
list(d.items())[1][1]
```

```
20
```

Methods – d.keys() / d.values()

```
d = {'a': 10, 'b': 20, 'c': 30}
d
```

```
{'a': 10, 'b': 20, 'c': 30}
```

- **d.keys()** returns a list of all keys in d:

```
list(d.keys())
```

```
['a', 'b', 'c']
```

- **d.values()** returns a list of all values in d:

```
list(d.values())
```

```
[10, 20, 30]
```

Methods – d.pop(<key>[, <default>])

- Removes a key from a dictionary, if it is present, and returns its value.
- If <key> is present in d, d.pop(<key>) removes <key> and returns its associated value:

```
d = {'a': 10, 'b': 20, 'c': 30}
d.pop('b')
```

```
20
```

```
d
```

```
{'a': 10, 'c': 30}
```

Methods – d.pop(<key>[, <default>]) (cont.)

- d.pop(<key>) raises a KeyError exception if <key> is not in d:

```
d = {'a': 10, 'b': 20, 'c': 30}
d.pop('z')
```

```
---------------------------------------------------------------------------
KeyError
<ipython-input-3-c5c02d6cff06> in <module>()
      1 d = {'a': 10, 'b': 20, 'c': 30}
----> 2 d.pop('z')

KeyError: 'z'
```

Methods – d.pop(<key>[, <default>]) (cont.)

- If <key> is not in d, and the optional <default> argument is specified, then that value is returned, and **no exception is raised:**

```
d = {'a': 10, 'b': 20, 'c': 30}
d.pop('z', -1)
```

```
-1
```

```
d
```

```
{'a': 10, 'b': 20, 'c': 30}
```

Methods – d.popitem()

- Removes a key-value pair from a dictionary.
- d.popitem() removes the last key-value pair added from d and returns it as a tuple:

```
d = {'a': 10, 'b': 20, 'c': 30}
d.popitem()
```

```
('c', 30)
```

```
d
```

```
{'a': 10, 'b': 20}
```

```
d.popitem()
```

```
('b', 20)
```

```
d
```

```
{'a': 10}
```

Methods – d.popitem() (cont.)

- If d is empty, d.popitem() raises a KeyError exception:

```
d={}
d
```

```
{}
```

```
d.popitem()
```

```
---------------------------------------------------------------------------
KeyError
<ipython-input-39-83c64cff336b> in <module>()
----> 1 d.popitem()

KeyError: 'popitem(): dictionary is empty'
```

Methods – d.update(<obj>)

- Merges a dictionary with another dictionary or with an iterable of key-value pairs.
- If <obj> is a dictionary, d.update(<obj>) merges the entries from <obj> into d. For each key in <obj>:
- If the key is not present in d, the key-value pair from <obj> is added to d.
- If the key is already present in d, the corresponding value in d for that key is updated to the value from <obj>.
- Here is an example showing two dictionaries merged together:

Methods – d.update(<obj>) - Example

- Here is an example showing two dictionaries merged together:

```
d1 = {'a': 10, 'b': 20, 'c': 30}
d2 = {'b': 200, 'd': 400}
```

```
d1.update(d2)
```

```
d1
```

```
{'a': 10, 'b': 200, 'c': 30, 'd': 400}
```

```
d2
```

```
{'b': 200, 'd': 400}
```

References & Tools

References & Tools

- https://www.w3schools.com/
- https://www.wikipedia.org/


thank you

Python

Session-04

Ali Heikal
Fatma El-Zahraa
Mohamed Fakhar El-Deen
Ramadan Babers

Objectives

- What is a loop
- For loop
- While loop
- Infinite loop
- If/Elif/Else
- Loop keywords (break/continue)
- Exercises on loops and conditions

1-Why do we need lists and loops?

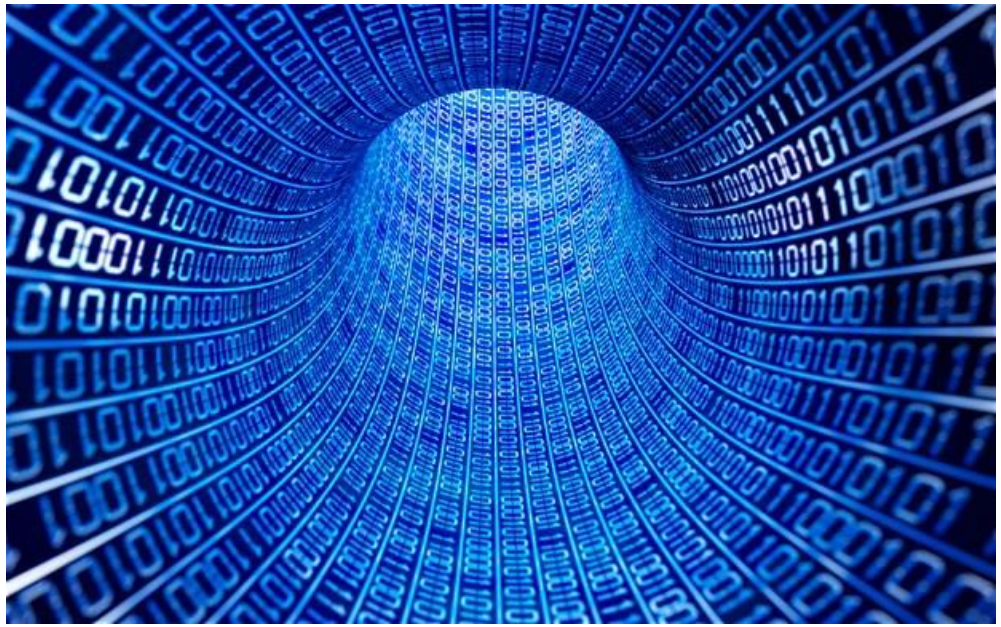
In programming:

- lists used to store sequences of related data and to perform the same operation on every element in a list, like displaying each element or manipulating them mathematically.

- The loop used to iterate over each element we stored in a list, repeating the same code for each element.

1-1 Deal with large datasets

A very common situation in biological research is to have a large collection of data (DNA sequences, SNP positions, gene expression measurements) that all need to be processed in the same way.



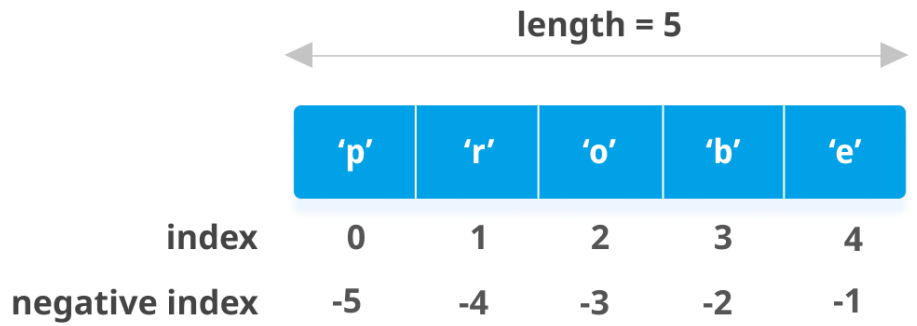
1-2 When we've needed to store multiple bits of information

for example, the three DNA sequences as below

```
seq_1 = "ATCGTACGATCGATCGATCGCTAGACGTATCG"
seq_2 = "actgatcgacgatcgatcgatcacgact"
seq_3 = "ACTGAC-ACTGT-ACTGTA----CATGTG"
```

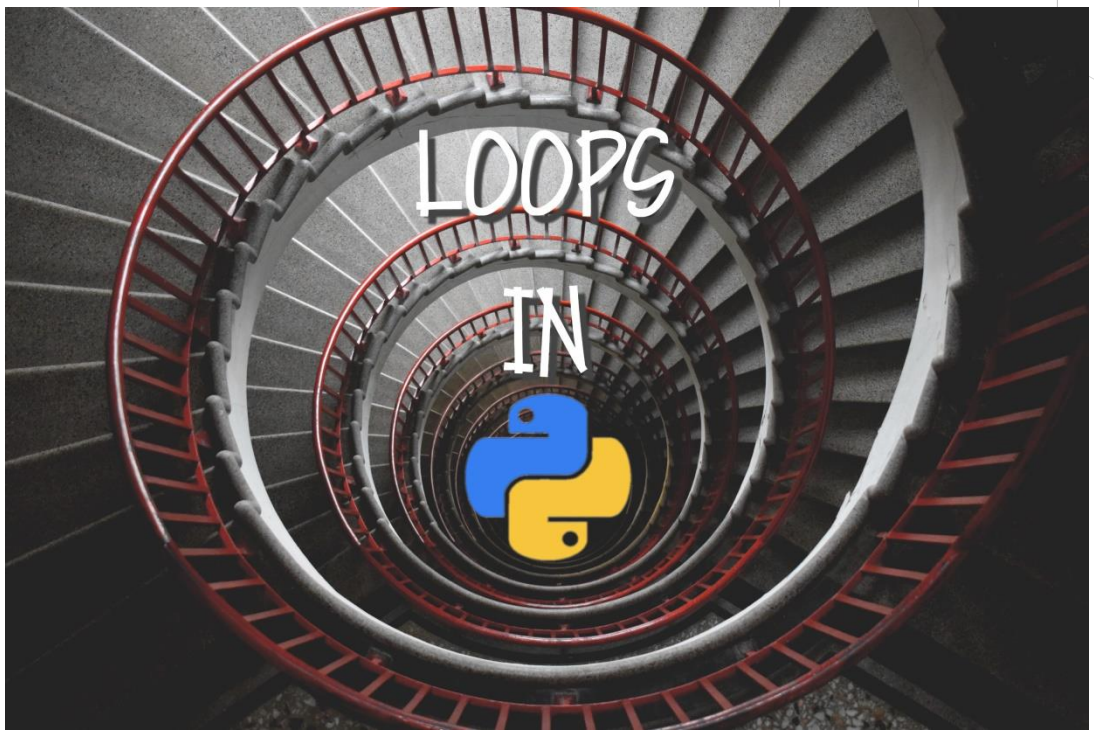
1-3 list

- A list (also called an array in other programming languages) is a tool that can be used to store multiple pieces of information at once.
- It can also be defined as a variable containing multiple other variables.



1-4 loop

- Is a sequence of instructions that is continually repeated until a certain condition is reached.



2- Creating lists and retrieving elements

2-1 To make a new list

- Put several strings or numbers inside square brackets, separated by commas. Two examples below. (Each individual item in a list is called an element)

```
apes = ["Homo sapiens", "Pan troglodytes", "Gorilla gorilla"]
conserved_sites = [24, 56, 132]
```

2- Creating lists and retrieving elements

2-2 To retrieve data from a list

- The index is known

```
apes = ["Homo sapiens", "Pan troglodytes", "Gorilla gorilla"]
conserved_sites = [24, 56, 132]
print(apes[0])
first_site = conserved_sites[2]
print(first_site)
```

```
Homo sapiens
132
```

2- Creating lists and retrieving elements

2-2 To retrieve data from a list

- The index is unknown

```
apes = ["Homo sapiens", "Pan troglodytes", "Gorilla gorilla"]
chimp_index = apes.index("Pan troglodytes")
print(chimp_index)
```

```
1
```

3- Writing a loop

- We have the following list

```
apes = ["Homo sapiens", "Pan troglodytes", "Gorilla gorilla"]
```

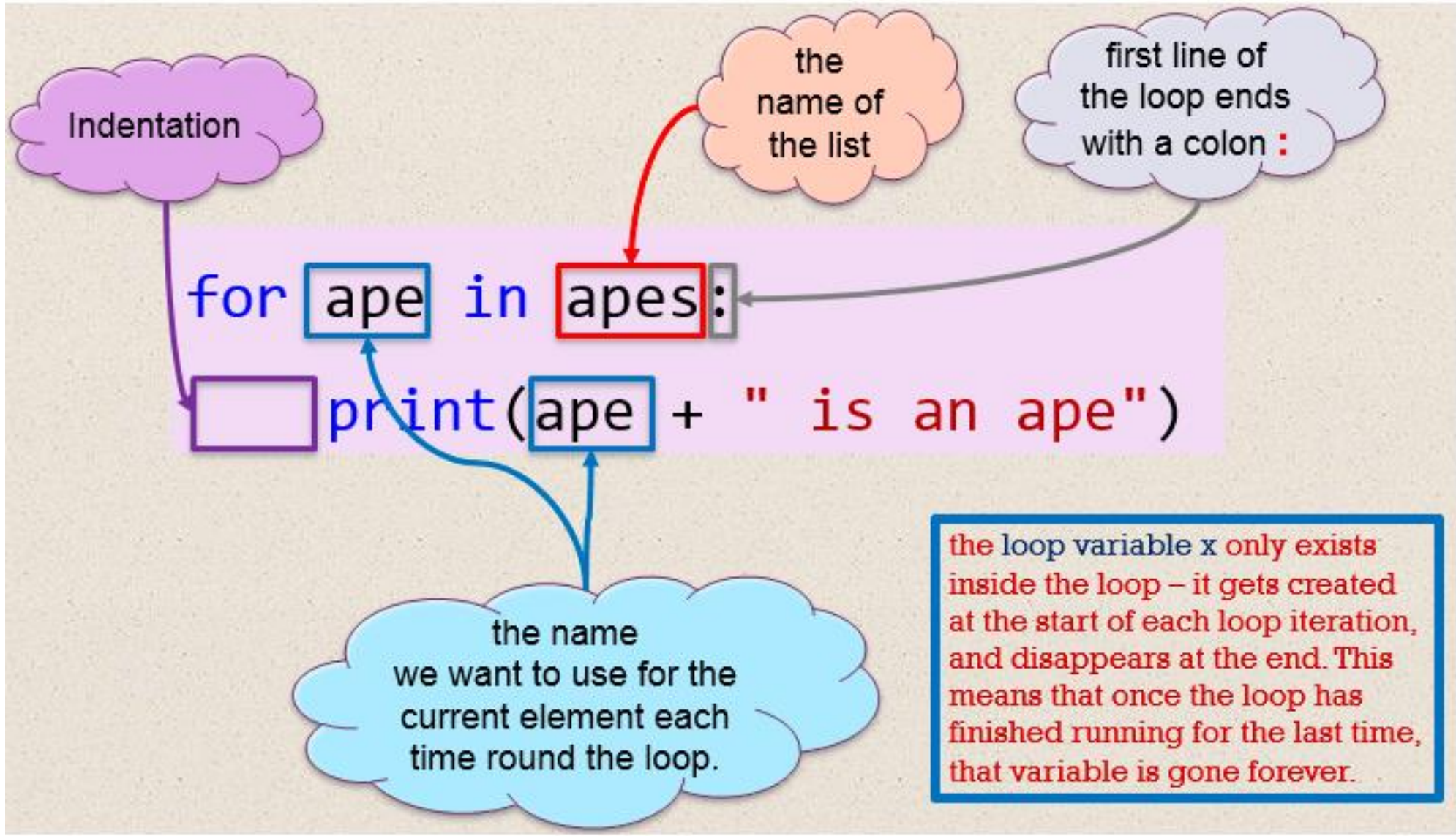
- We need to print it as:

Homo sapiens is an ape
Pan troglodytes is an ape
Gorilla gorilla is an ape

- Solution

```
apes = ["Homo sapiens", "Pan troglodytes", "Gorilla gorilla"]
for ape in apes:
    print(ape + " is an ape")
```

```
Homo sapiens is an ape
Pan troglodytes is an ape
Gorilla gorilla is an ape
```


Indentation
the name of the list
first line of the loop ends with a colon :
for ape in apes:
print(ape + " is an ape")
the name we want to use for the current element each time round the loop.
the loop variable x only exists inside the loop – it gets created at the start of each loop iteration, and disappears at the end. This means that once the loop has finished running for the last time, that variable is gone forever.

3-1 Iterating over lines in a file

- When we treat a string as a list, each character becomes an individual element,

But

- When we treat a file as a list, each line becomes an individual element.

3-1 Iterating over lines in a file - Example

- we have the protein = "vlspadktnv" and we need to print it as

vls vlsp vlspa vlspad ...etc...

```
vls
vlsp
vlspa
vlspad
vlspadk
vlspadkt
vlspadktn
vlspadktnv
```

Solution

```
protein = "vlspadktnv"
stop_positions = [3,4,5,6,7,8,9,10]
for stop in stop_positions:
    substring = protein[0:stop]
    print(substring)
```

```
vls
vlsp
vlspa
vlspad
vlspadk
vlspadkt
vlspadktn
vlspadktnv
```

3-2 Looping with ranges (Example 1)

```
for number in range(6):
    print (number)
```

```
0
1
2
3
4
5
```

3-2 Looping with ranges (Example 2)

```
for number in range(3,8):
    print(number)
```

```
3
4
5
6
7
```

3-2 Looping with ranges (Example 3)

```
for number in range(2,14,4):
    print(number)
```

```
2
6
10
```

4- Conditions, True and False

```
print(3 == 5)
print(3 > 5)
print(3 <=5)
print(len("ATGC") > 5)
print("GAATTC".count("T") > 1)
print("ATGCTT".startswith("ATG"))
print("ATGCTT".endswith("TTT"))
print("ATGCTT".isupper())
print("ATGCTT".islower())
print("V" in ["V", "W", "L"])
```

```
print(3 == 5)                          False
print(3 > 5)                           False
print(3 <=5)                           True
print(len("ATGC") > 5)                 False
print("GAATTC".count("T") > 1)         True
print("ATGCTT".startswith("ATG"))      True
print("ATGCTT".endswith("TTT"))        False
print("ATGCTT".isupper())              True
print("ATGCTT".islower())              False
print("V" in ["V", "W", "L"])          True
```

5- If Statement

If statement

- if statement: Executes a group of statements only if a certain condition is true. Otherwise, the statements are skipped.

Syntax:

```
if condition:
    statements
```

If statement (Example -1)

```
gpa = 3.4
if gpa > 2.0:
    print ("Your Application is accepted")
```

```
Your Application is accepted
```

If statement (Example -2)

```
expression_level = 125
if expression_level > 100:
    print("gene is highly expressed")
```

```
gene is highly expressed
```

If statement (Example -3)

List - Loop - If statement example

```
accs = ['ab56', 'bh84', 'hv76', 'ay93', 'ap97', 'bd72']
for accession in accs:
    if accession.startswith('a'):
        print(accession)
```

```
ab56
ay93
ap97
```

If/Else Statement

- if/else statement: Executes one block of statements if a certain condition is True, and a second block of statements if it is False.

Syntax:

```
if condition:
        statements
else:
        statements
```

If/Else Statement (Example)

```
gpa = 1.4
if gpa > 1.0:
    print ("Welcome to Helwan University")
else:
    print ("Your Application is Denied")
```

```
Welcome to Helwan University
```

Exercise

Ask about the temperature degree. If it is greater than 35 degree then wear shorts if not wear long pants

Exercise (solution)

Ask about the temperature degree. If it is greater than 35 degree then wear shorts if not wear long pants

```
temperature = float(input('What is the temperature? '))
if temperature > 35:
        print('Wear shorts.')
else:
        print('Wear long pants.')
print('Get some exercise outside.')
```

```
What is the temperature? 36
Wear shorts.
Get some exercise outside.
```

Multiple conditions elif ("else if")

if/else statement: Executes one block of statements if a certain condition is True, and a second block of statements if it is False.

Syntax:

```
if condition:
    statements
elif condition:
    statements
else:
    statements
```

Multiple conditions elif ("else if") - Example

```
gpa = 1.4
if gpa > 3.0:
    print ("Welcome to Helwan University")
elif gpa > 1.2:
    print ("Welcome to Faculty of Science")
else:
    print ("Your Application is Denied")
```

```
Welcome to Faculty of Science
```

6- While loop

While loop

while loop: repeatedly executes a target statement as long as a given condition is true.

Syntax:

```
while expression:
    statement(s)
```

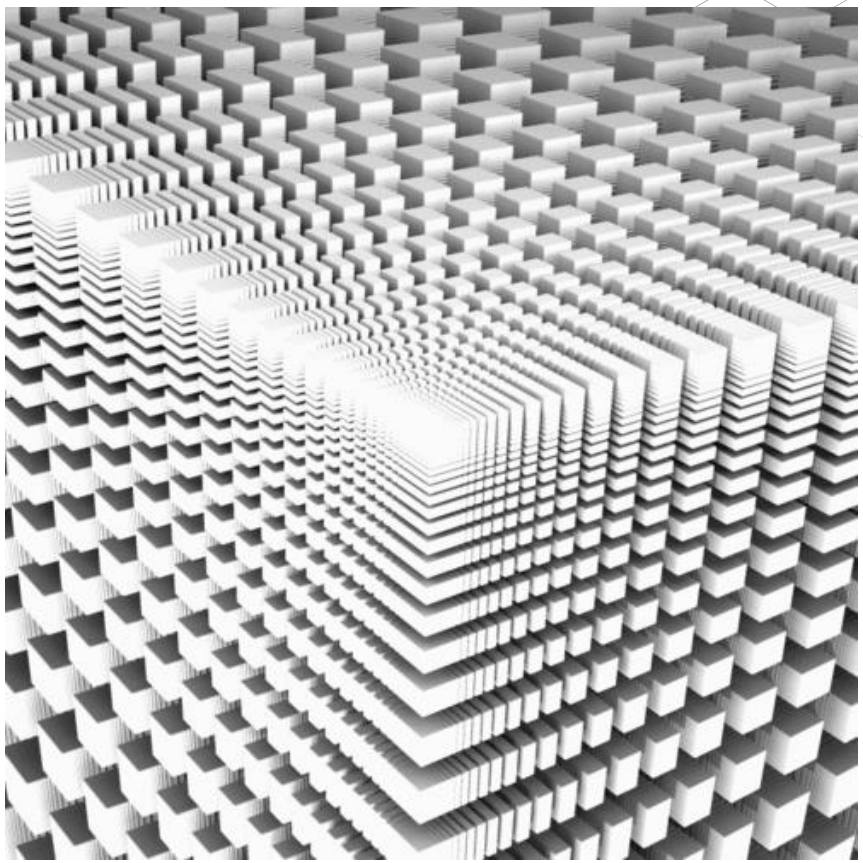
While loop (Example)

```
i=0
while i !=10:
    i=i+1
    print (i)
```

```
1
2
3
4
5
6
7
8
9
10
```

Infinite Loops

- Every while loop needs a way to stop running so it won't continue to run forever.



Infinite Loops (Example)

- this counting loop should count from 1 to 5:

```
x = 1
while x <= 5:
    print(x)
    x += 1
```

```
1
2
3
4
5
```

Infinite Loops (Example)

- But if you accidentally omit the line x += 1 (as shown in pervious example), the loop will run forever:

```
x = 1
while x <= 5:
    print(x)
    x += 1
```

```
x = 1
while x <= 5:
    print(x)
```

7- Exit a Loop with break keyword

The break Statement

- The break statement, when it occurs within a for loop, or while loop, causes the loop to terminate immediately and program execution continues with the statement after the loop

The break Statement (Example - 1)

```
prompt = "\nPlease enter the name of a city you have visited:"
prompt += "\n(Enter 'quit' when you are finished.) "
while True:
    city = input(prompt)
    if city == 'quit':
        break
    else:
        print("I'd love to go to " + city.title() + "!")
```

The break Statement (Example - 1)

```
prompt = "\nPlease enter the name of a city you have visited:"
prompt += "\n(Enter 'quit' when you are finished.) "
while True:
    city = input(prompt)
    if city == 'quit':
        break
    else:
        print("I'd love to go to " + city.title() + "!")
```

```
Please enter the name of a city you have visited:
(Enter 'quit' when you are finished.) Cairo
I'd love to go to Cairo!

Please enter the name of a city you have visited:
(Enter 'quit' when you are finished.) Paris
I'd love to go to Paris!

Please enter the name of a city you have visited:
(Enter 'quit' when you are finished.) quit
```

The break Statement (Example - 2)

```
n = 5
while n > 0:
    n -= 1
    if n == 2:
        break
    print(n)
print('Loop ended.')
```

```
4
3
Loop ended.
```

The break Statement (Example - 3)

```
prompt = "\nTell me something, and I will repeat it back to you:"
prompt += "\nEnter 'quit' to end the program. "
message = ""
while message != "quit":
    message = input(prompt)
    if message != 'quit':
        print(message)
```

```
Tell me something, and I will repeat it back to you:
Enter 'quit' to end the program. I will graduate from ITI soon
I will graduate from ITI soon

Tell me something, and I will repeat it back to you:
Enter 'quit' to end the program. quit
```

The continue Statement

- The continue statement, when it occurs within a for loop, a while loop, or a do-while loop, causes the program execution to jump to the bottom of the loop.

- None of the statements between the continue statement and the bottom of the loop are executed. The loop then continues as normal.

The continue Statement (Example)

consider a loop that counts from 1 to 10 but prints only the odd numbers in that range

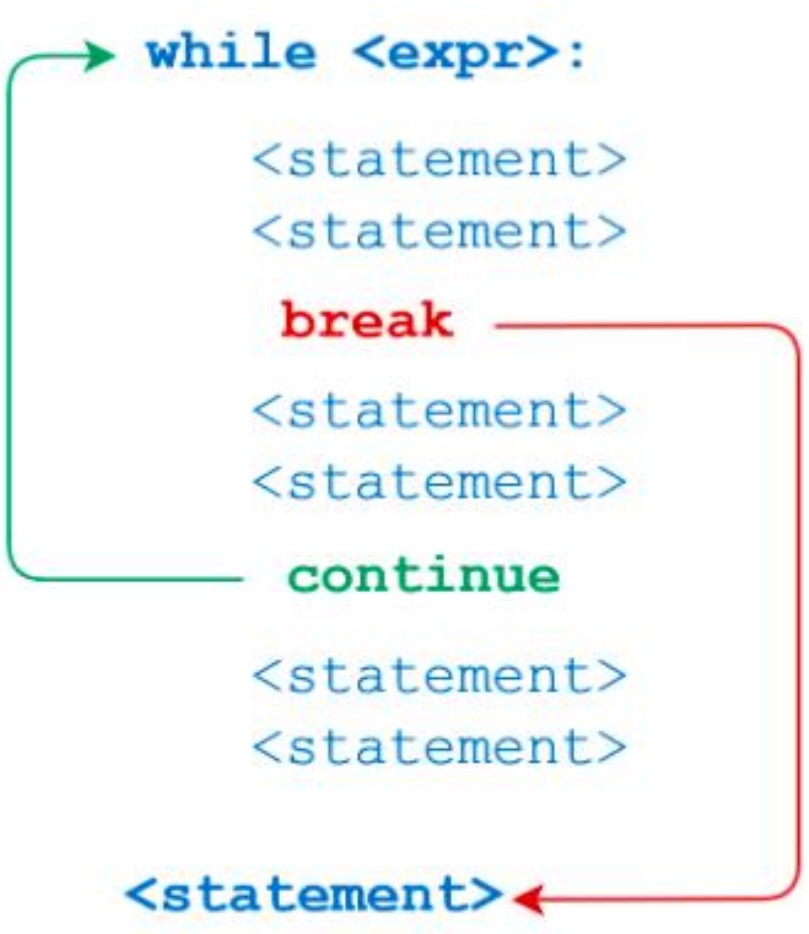
The continue Statement (Example - Soln)

```
current_number = 0
while current_number < 10:
    current_number += 1
    if current_number % 2 == 0:
        continue
    print(current_number)
```

```
1
3
5
7
9
```

Break

immediately terminates a loop entirely. Program execution proceeds to the first statement following the loop body.



Continue

immediately terminates the current loop iteration. Execution jumps to the top of the loop, and the controlling expression is re-evaluated to determine whether the loop will execute again or terminate.

The difference between break and continue in a Loop

One-Line while Loops

- A while loop can be specified on one line. If there are multiple statements in the block that makes up the loop body, they can be separated by semicolons (;):

One-Line while Loops (Example)

```
n = 5
while n > 0: n -= 1; print(n)
```

```
4
3
2
1
0
```

THANKS!

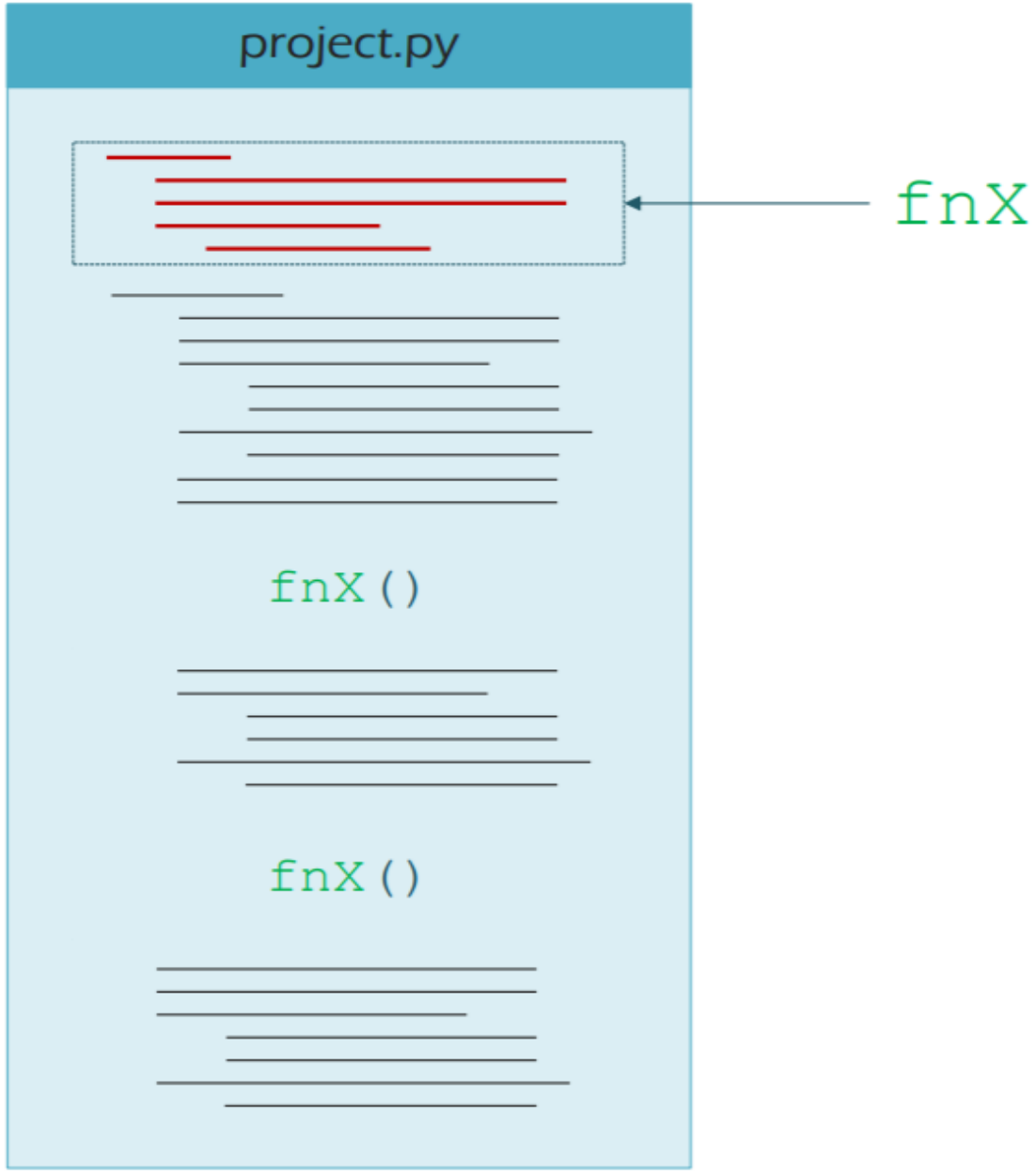
ANY QUESTIONS?

Python

Session-05

Objectives

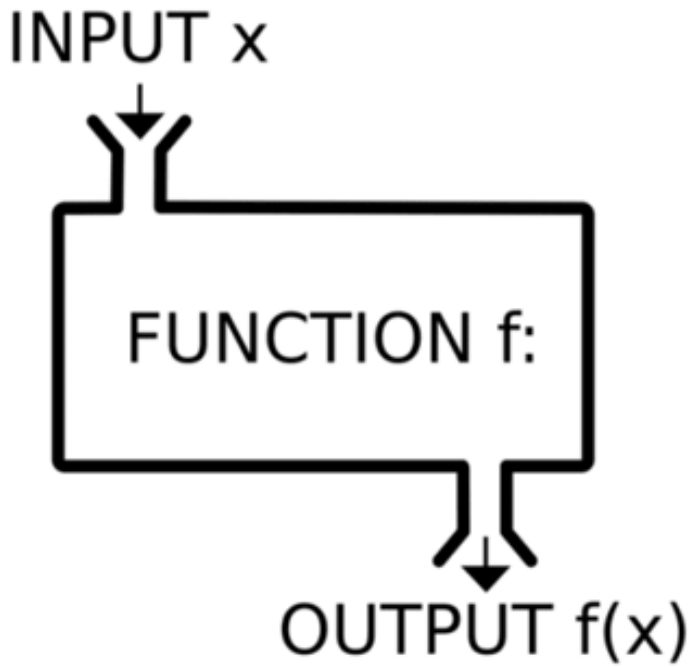
- Defining a function
- Arguments
- Variable-length arguments
- Return statement
- Recursion
- Lambda Function
- Tips and tricks
- Exercises on functions


project.py
fnX
fnX()
fnX()

Defining a function

Introduction to functions in python

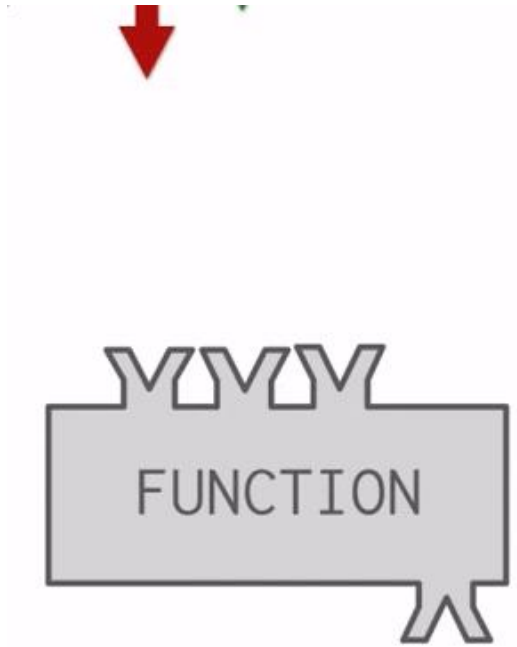
- Function in any programming language is a sequence of statements in a certain order, given a name.
- When called, those statements (function) are executed.
- So, we don’t have to write the code again and again for each [type of] data that we want to apply it to.



Introduction to functions in python (cont.)

In another words

- A function is a block of code which only runs when it is called.
- You can pass data, known as arguments, into a function.
- A function can return data as a result.



Types of Function

Built-in functions

```
print("Enter your name: ")
```

```
Enter your name:
```

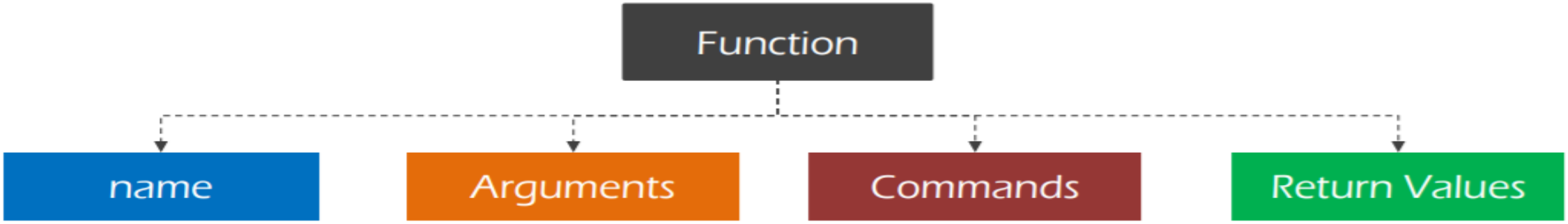
```
var=18
type(var)
```

```
int
```

User-define function

- Will discussed in this session

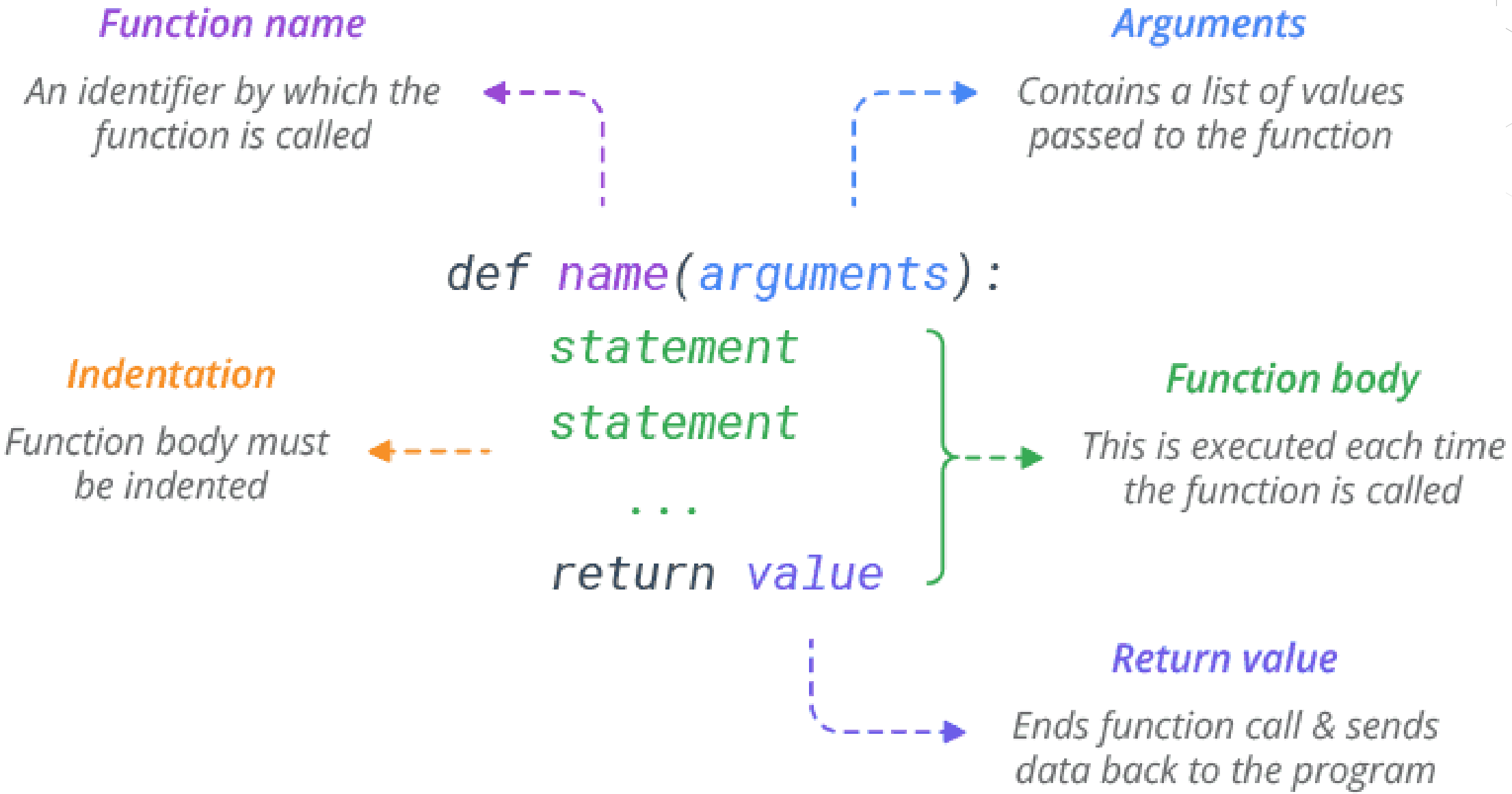
```
def fnName:
    pass
```




```
def measureTemp ( temp ):
    if temp < 37:
        return "Too Cold"
    elif temp > 37:
        return "Too Hot"
    return "Normal"
```

```
measureTemp(37)
# "Normal"
```

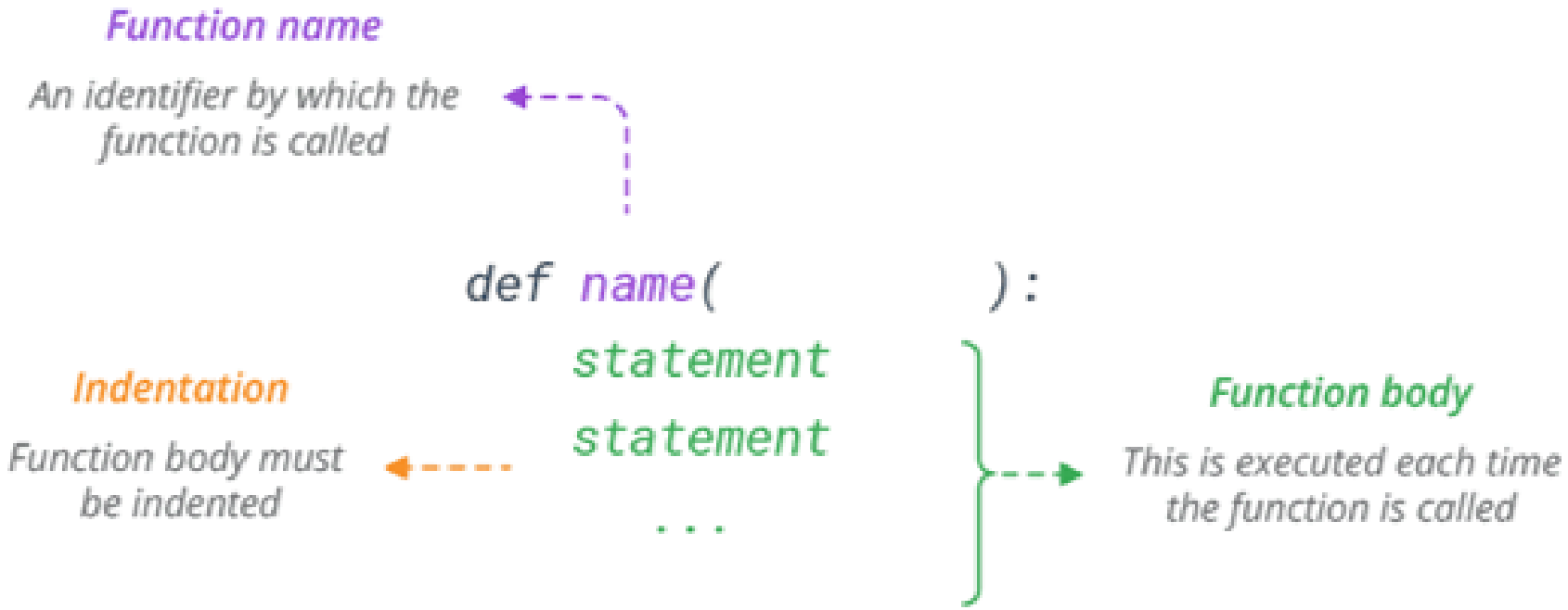
Function syntax in python



Function syntax in python

Define function

- In Python a function is defined using the **def** keyword:



Call a Function

- The def statement only creates a function but does not call it.
- After the def has run, you can call (run) the function by adding parentheses after the function's name.

Function definition

```
def my_function():
```

Function call

```
my_function()
```

Example

Function definition

```
def my_function():
    print("Welcome to Function session")
```

Function call

```
my_function()
```

Output

```
Welcome to Function session
```

Workshop 01

- Create a function that print your full name, your age and your collage.
- Then call a function at the body of your code

Workshop 01 - Solution

```
def personal_details():
    print("Name: Ramadan Babers\n","Age: "+str(45) +"\n" ,"Faculty: Faculty of Science")
```

```
personal_details()
```

```
Name: Ramadan Babers
 Age: 45
 Faculty: Faculty of Science
```

Arguments

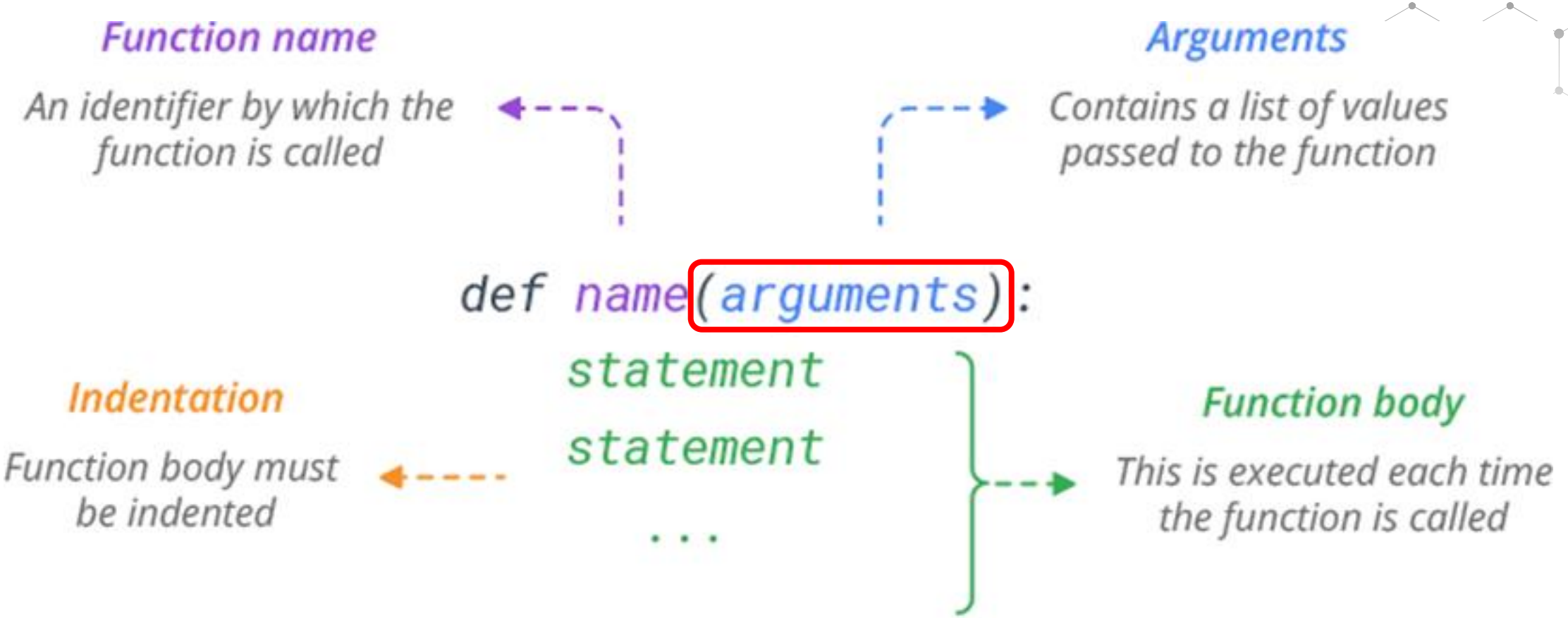
Arguments

Pass Arguments

- You can send information to a function by passing values, known as arguments. Arguments are declared after the function name in parentheses.

- When you call a function with arguments, the values of those arguments are copied to their corresponding parameters inside the function.

Arguments (cont.)



Example

Function definition

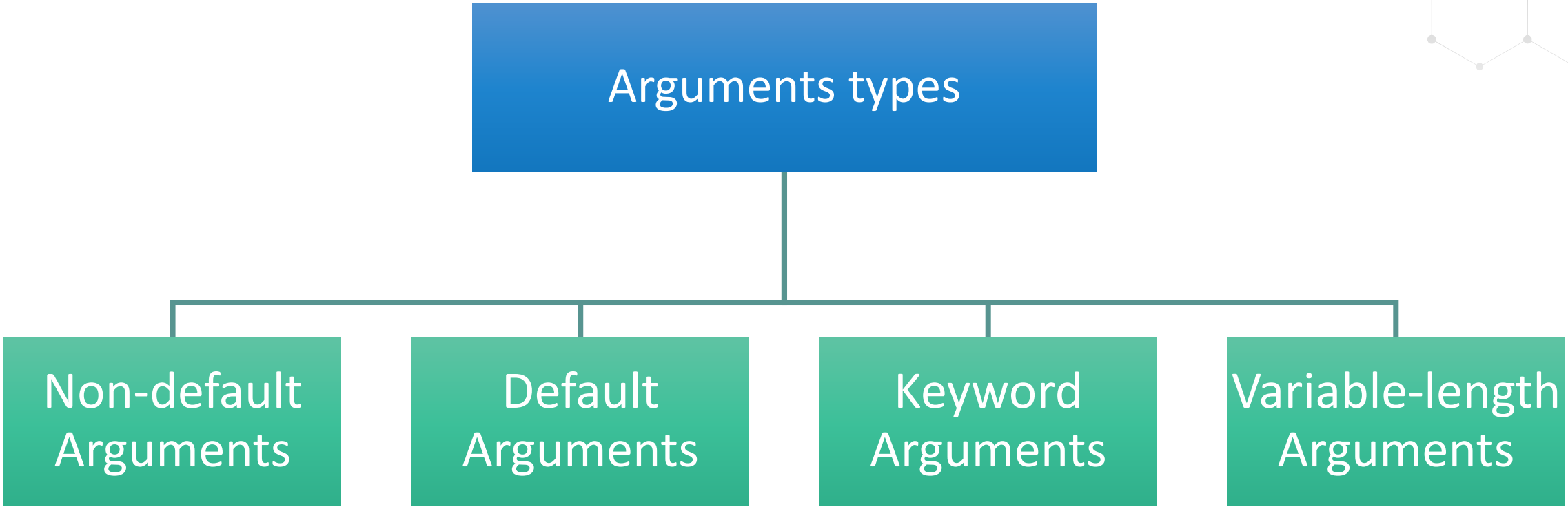
```
def hello(name):
    print('Hello,', name)
```

Function call

```
hello('Bob')
```

Output

```
Hello, Bob
```


Arguments types
Non-default Arguments
Default Arguments
Keyword Arguments
Variable-length Arguments

1. Non-default Arguments

- Function arguments can have no default values in function define section.

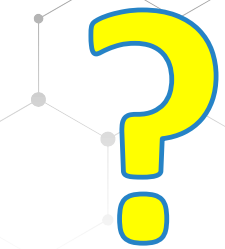
```
def greet(name, msg):
        print("Hello", name + ', ' + msg)
```

```
greet("Nabil", "How do you do?")
```

```
Hello Nabil, How do you do?
```

1. Non-default Arguments (cont.)

```
greet("Nabil")
```



```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-4-ae8fdcb08841> in <module>()
----> 1 greet("Nabil")

TypeError: greet() missing 1 required positional argument: 'msg'
```

Workshop 02

- Create a function that print your full name, your age and your collage by using Non-default arguments

Workshop 02 - Solution

```
def personal(name,age, faculty):
    print("Name: "+name, "\nAge: "+ str(age), "\nFaculty: "+faculty)
```

```
personal("Mostafa",25, "Arts")
```

```
Name: Mostafa
Age: 25
Faculty: Arts
```

Workshop 03

- Write a Python program to print the even numbers from a given list

  Sample List : [1, 2, 3, 4, 5, 6, 7, 8, 9]
  Expected Result : [2, 4, 6, 8]

Workshop 03 - Solution

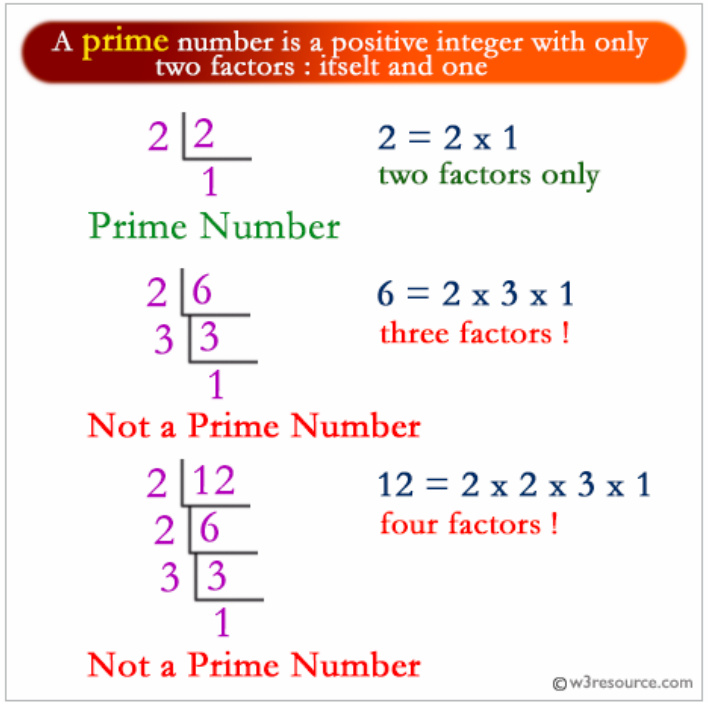
```
def is_even_num(l):
    enum = []
    for n in l:
        if n % 2 == 0:
            enum.append(n)
    return enum
```

```
print("Even Number list: " , is_even_num([1, 2, 3, 4, 5, 6, 7, 8, 9]))
```

```
Even Number list:  [2, 4, 6, 8]
```

Exercise 01

- Write a Python function that takes a number as a parameter and check the number is prime or not



Prime number between 1 to 100:

1	2	3	4	5	6	7	8	9	10
11	12	13	14	15	16	17	18	19	20
21	22	23	24	25	26	27	28	29	30
31	32	33	34	35	36	37	38	39	40
41	42	43	44	45	46	47	48	49	50
51	52	53	54	55	56	57	58	59	60
61	62	63	64	65	66	67	68	69	70
71	72	73	74	75	76	77	78	79	80
81	82	83	84	85	86	87	88	89	90
91	92	93	94	95	96	97	98	99	100

Exercise 01 - Solution

```
def test_prime(n):
    if (n==1):
        return False
    elif (n==2):
        return True;
    else:
        for x in range(2,n):
            if(n % x==0):
                return False
        return True
```

```
print("The number is Prime: ",test_prime(11))
```

```
The number is Prime:  True
```

Exercise 02

- Write a Python function to calculate the factorial of a number (a non-negative integer). The function accepts the number as an argument

Exercise 02 - Solution

```
def factorial(n):
    fact = 1
    for num in range(2, n + 1):
        fact = fact * num
    return(fact)
```

```
factorial(4)
```

```
24
```

2. Default Arguments

- Function arguments can have default values in function define section.
- We can provide a default value to an argument by using the assignment operator (=).

```
def greet(name, msg="Good morning!"):
          print("Hello", name + ', ' + msg)
```

```
greet("Ahmed")
```

```
Hello Ahmed, Good morning!
```

2. Default Arguments (cont.)

```
greet("Nabil", "How do you do?")
```

```
Hello Nabil, How do you do?
```

2. Default Arguments (cont.)

- Is it possible to do that ?

```
def greet(msg="Good morning!" , name):
          print("Hello", name + ', ' + msg)
```

```
  File "<ipython-input-20-29a138191420>", line 1
    def greet(msg="Good morning!" , name):
                  ^
SyntaxError: non-default argument follows default argument
```

Workshop 04

- Create a function showEmployee() in such a way that it should accept employee name, and it’s salary and display both, and if the salary is missing in function call it should show it as 9000

Workshop 04 - Solution

```
def showEmployee(name, salary=9000):
    print("Employee", name, "salary is:", salary)
```

```
showEmployee("Ben", 18000)
showEmployee("Ben")
```

```
Employee Ben salary is: 18000
Employee Ben salary is: 9000
```

3. Keyword Arguments / named Arguments

- Keyword arguments are passed by it’s name instead of their position as opposed to positional arguments in the function call.

- As a result we don’t have to mind about position of arguments when calling a function.

3. Keyword Arguments / named Arguments (cont.)

```
def greet(name, msg="Good morning!"):
        print("Hello", name + ', ' + msg)
```

```
greet(name="Ahmed", msg= "How do you do?")
```

```
Hello Ahmed, How do you do?
```

```
greet(msg= "How do you do?" , name="Ahmed")
```

```
Hello Ahmed, How do you do?
```

3. Keyword Arguments / named Arguments (cont.)

- 1positional, 1 keyword argument

```
def greet(name, msg="Good morning!"):
        print("Hello", name + ', ' + msg)
```

```
greet("Ahmed", msg = "How do you do?")
```

```
Hello Ahmed, How do you do?
```

```
greet(name= "Bruce" , "How do you do?")
```

```
  File "<ipython-input-24-7b3725c21fda>", line 1
    greet(name= "Bruce" , "How do you do?")
                        ^
SyntaxError: positional argument follows keyword argument
```

4. Variable Length Arguments (*args and **kwargs)

- Variable length arguments are useful when you want to create functions that take unlimited number of arguments.
- Unlimited in the sense that you do not know beforehand how many arguments can be passed to your function by the user.

4. Variable Length Arguments (*args and **kwargs) (cont.)

*args

- When you prefix a parameter with an asterisk * , it collects all the unmatched positional arguments into a tuple.
- Because it is a normal tuple object, you can perform any operation that a tuple supports, like indexing, iteration etc.

4. Variable Length Arguments (*args and **kwargs) (cont.)

- Following function prints all the arguments passed to the function as a tuple.

```
def print_arguments(*args):
        print(args)

print_arguments(1, 54, 60, 8, 98, 12)
```

```
(1, 54, 60, 8, 98, 12)
```

4. Variable Length Arguments (*args and **kwargs) (cont.)

**kwargs

- The ** syntax is similar, but it only works for keyword arguments.
- It collects them into a new dictionary, where the argument names are the keys, and their values are the corresponding dictionary values.

4. Variable Length Arguments (*args and **kwargs) (cont.)

```
def print_arguments(**kwargs):
        print(kwargs)


print_arguments(name='Bob', age=25, job='dev')
```

```
{'name': 'Bob', 'age': 25, 'job': 'dev'}
```

Workshop 05

- Write a function func1() such that it can accept a variable length of argument and print all arguments value

```
func1(20, 40, 60)
func1(80, 100)
```

Expected Output:

After `func1(20, 40, 60)`:

20
40
60

After `func1(80, 100)`:

80
100

Workshop 05 - Solution

```
def func1(*args):
    for i in args:
        print(i)
```

```
func1(20, 40, 60)
```

```
20
40
60
```

```
func1(80, 100)
```

```
80
100
```

Return Statements

Return Value

- To return a value from a function, simply use a **return** statement.
- Once a return statement is executed, nothing else in the function body is executed.

Return Value (cont.)

```
def sum(a, b):
    return a + b
```

```
x = sum(3, 4)
print(x)
```

```
7
```

Return Value (cont.)

- Note: a python function always returns a value. So, if you do not include any return statement, it automatically returns **None**.

```
def sum(a, b):
    return a + b

x = sum(3, 4)
print(x)
```

```
7
```

```
def sum(a, b):
    a + b

x = sum(3, 4)
print(x)
```

```
None
```

Return Multiple Values

- Python can return multiple values, something missing from many other languages. You can do this by separating return values with a **comma**.

```
def func(a, b):
        return a+b, a-b
```

```
result = func(3, 2)
print(result)
```

```
(5, 1)
```

Return Multiple Values (cont.)

- When you return multiple values, Python packs them in a single tuple and returns it.
- You can then use multiple assignment to unpack the parts of the returned tuple.

```
def func(a, b):
      return a+b, a-b
```

```
add, sub = func(3, 2)
print(add)
print(sub)
```

```
5
1
```

Workshop 06

- Write a function calculation() such that it can accept two variables and calculate the addition and subtraction of it. And also it must return both addition and subtraction in a single return call

For example:

```
def calculation(a, b):
    # Your Code

res = calculation(40, 10)
print(res)
```

A res should produce result 50, 30

Workshop 06 – Solution

Solution 01

```
def calculation(a, b):
    return a+b, a-b
```

```
res = calculation(40, 10)
print(res)
```

```
(50, 30)
```

Solution 02

```
def calculation(a, b):
    return a+b, a-b
```

```
add, sub = calculation(40, 10)
print("add: ", add)
print("sub: ", sub)
```

```
add:  50
sub:  30
```

Recursion

Recursion

- A recursive function is a function that calls itself and repeats its behavior until some condition is met to return a result.

Recursion - Example

- countdown() is a recursive function that calls itself (recurse) to countdown.
- If num is 0 or negative, it prints the word “Stop”.
- Otherwise, it prints num and then calls itself, passing num-1 as an argument.

```
def countdown(num):
        if num <= 0:
            print('Stop')
        else:
            print(num)
            countdown(num-1)
```

```
countdown(4)
```

```
4
3
2
1
Stop
```

Workshop 07

- Write a recursive function to calculate the sum of numbers from 0 to 10

Workshop 07 - Solution

```
def recur_sum(n):
    if n <= 1:
        return n
    else:
        return n + recur_sum(n-1)
```

```
recur_sum(3)
```

```
6
```

```
recur_sum(5)
```

```
15
```

```
recur_sum(1)
```

```
1
```

```
recur_sum(0)
```

```
0
```

```
recur_sum(-2)
```

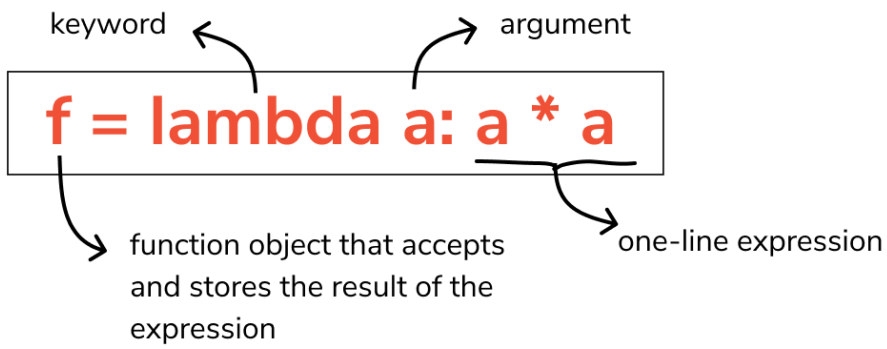
```
-2
```

Lambda Functions

Lambda Functions

- In Python, an anonymous function means that a function is without a name.
- As we already know that the **def** keyword is used to define a normal function in Python.
- Similarly, the **lambda** keyword is used to define an anonymous function in Python.

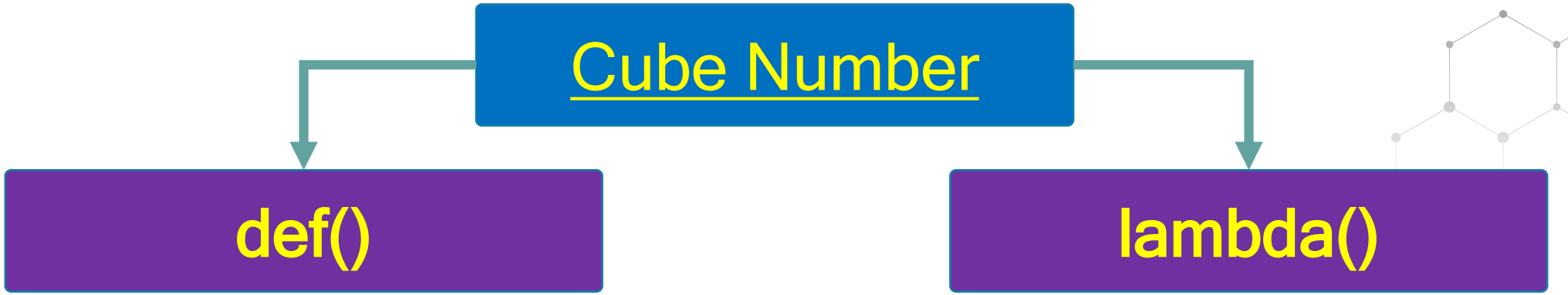
Syntax: lambda arguments: expression



Lambda Functions (cont.)

- This function can have any number of arguments but only one expression, which is evaluated and returned.
- One is free to use lambda functions wherever function objects are required.
- You need to keep in your knowledge that lambda functions are syntactically restricted to a single expression.
- It has various uses in particular fields of programming besides other types of expressions in functions.

Lambda Functions (Example)




```
def cube(y):
    return y*y*y
```

```
print(cube(5))
```

```
125
```

```
lambda_cube = lambda y: y*y*y
```

```
print(lambda_cube(5))
```

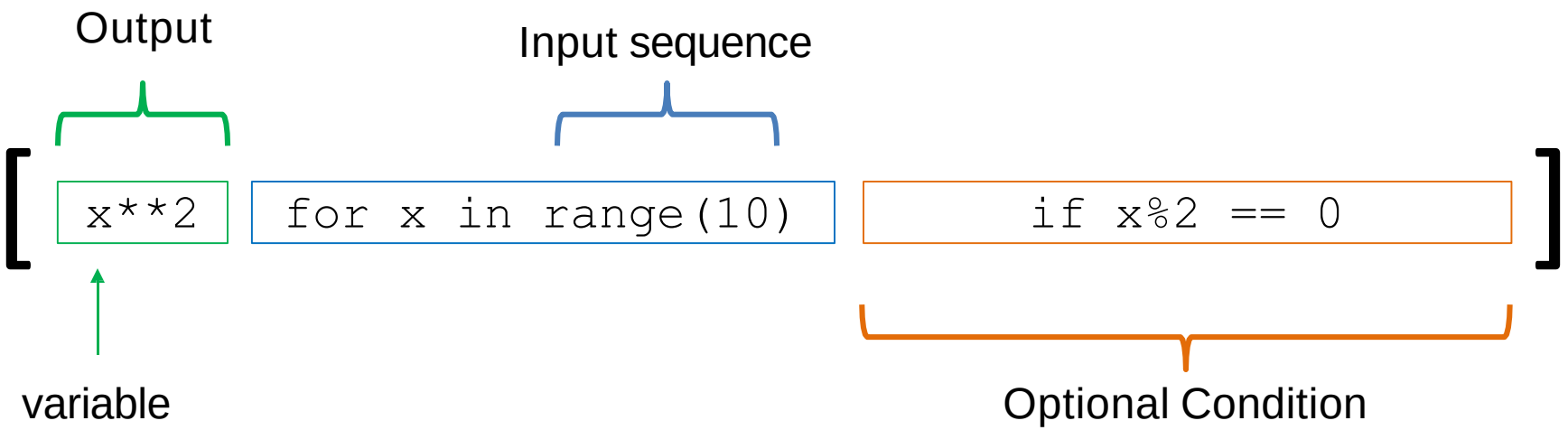
```
125
```

Sequence Unpacking

```
l = [1,13,3,7]

a,b,c,d = l

# a=1,b=13,c=3,d=7

a,*b,c = l

# a=1,b=[13,3],c=7
```

List Comprehension

It is an easy method to construct a list




```
L = [ x**2 for x in range(10) if x%2 == 0 ]
#output: [0, 4, 16, 36, 64]
```

Enumerate Function

```
languages = [“JavaScript”, “Python”, “Java”]
for i , l in enumerate(languages):
      print(“Element Value: ” , l, end=“, ”)
      print(“Element Index: ” , i)
```

```
Output:
Element Value: JavaScript, Element index:
               0
Element Value: Python, Element index: 1
Element Value: Java, Element index: 2
```

all & any

all check if all items in an iterable are truthy value. **any** check if one item at least in an iterable is truthy value.

```
L = [0,5,9,7,8]
all(L)     #False
any(L)     #True
```

THANKS!