# AI/ML for Climate Workshop

International Livestock Research Institute (ILRI)

hide: - toc

# 🌦 Python Basics for Climate and Meteorology

Welcome ❕

This is a practical introduction to the Python programming language, with climate and meteorology-flavored examples.

## 🚀 Interactive Learning

launch binder

**Click the Binder button above to launch an interactive Jupyter notebook environment where you can run all the code examples in this lesson!**

> 💡 ***Tip**: The Binder environment includes all necessary packages and sample climate data. It may take 1-2 minutes to start up the first time.*

Python is:
- An interpreted, high-level language
- Designed for readability and simplicity
- Very popular in data science, machine learning, and climate analytics
- Backed by strong scientific libraries (NumPy, Pandas, Xarray, etc.)

Python emphasizes:
- Simple, readable syntax
- Built-in powerful data types

- Large ecosystem of libraries
- Strong community support

---

## Contents

Exercises

---

# 0. Basic Syntax and Structure

```
# Import necessary modules
import sys
import keyword
import operator
from datetime import datetime
import os
```

---

## Keywords

Keywords are the reserved words in Python and can't be used as an identifier

```
print(keyword.kwlist) # List all Python Keywords
```

*Output:*

```
['False', 'None', 'True', 'and', 'as', 'assert', 'async', 'await', 'break', 'class', 'c
```

```
import = 125 # Keywords can't be used as identifiers
```

---

# Comments in Python

Comments can be used to explain the code for more readabilty.

```
# Single line comment
val1  = 10
```

```
# Convert 2 m temperature from Celsius to Kelvin (K = C + 273.15)
t2m_c = 27.3   # daily mean 2 m air temp in C
t2m_k = t2m_c + 273.15  # result in Kelvin
t2m_k
```

```
# Multiple line comment
'''
```

```
Multiple line comment
Multiple line comment
Multiple line comment
'''
```

```
"""
Multiple line comment
Multiple line comment
Multiple line comment
"""
```

---

# Statements

Instructions that a Python interpreter can execute

### Single line statement

```
p1 = 10 + 20
p1
```

### Multiple line statement

```
p1 = 20 + 30 \
 + 40 + 50 +\
 + 70 + 80
p1
```

### Multiple line statement

```
p2 = ['a' ,
    'b' ,
    'c' ,
    'd'
     ]
p2
```

---

# Indentation

- Indentation refers to the spaces at the beginning of a code line.

- It is very important as Python uses indentation to indicate a block of code.
- If the indentation is not correct we will endup with **IndentationError error**.

```
p = 10
if p == 10:
    print ('P is equal to 10') # correct indentation
```

```
# if indentation is skipped we will encounter "IndentationError: expected an inde
p = 10
if p == 10:
print ('P is equal to 10')
```

```
for i in range(0,5):
    print(i)
```

```
# Count wet days (> 1 mm) using proper indentation
rain = [0.0, 0.8, 1.2, 5.0, 0.0, 2.0]
wet = 0
for x in rain:
    if x > 1.0:
        wet += 1  # wet = wet + 1
wet
```

---

# Docstrings

1) Docstrings provide a convenient way of associating documentation with functions, classes, methods or modules.

2) They appear right after the definition of a function, method, class, or module.

```
def square(num):
    '''Square Function :- This function will return the square of a number'''
    return num**2
```

```
square(2)
```

```
square.__doc__  # We can access the Docstring using __doc__ method
```

```
def c_to_k(t_c: float) -> float:
    """Convert Celsius to Kelvin.

    Parameters
    ----------
    t_c : float
        Temperature in C.
    Returns
    -------
    float
        Temperature in Kelvin.
    """
    return t_c + 273.15

c_to_k(30.0)
```

```
c_to_k.__doc__
```

---

# 1. Variables and Data Types

## 1.1 Variables

- `Variable` is a reserved memory location to store values.
- A variable is created the moment you first assign a value to it.
- `A variable` in Python can either be of 3 data types: `integer` , `float` , or a `string` .
- `Data type` specifies the category of the variables.
- We can use `type(variable_name)` to find the type of given `variable_name` .
- Comments do not change anything and are not `compiled` .
- The lines inside triple quotes are ignore during runtime.
- We also use `=` to assign a value to the name of variable.
- Example: `var_name = 1` . Note that it's different to comparison operator of equal to ( `==` ).
- We can use `print()` to display the value of variable or the results of any expression.
- Be aware of indentations. Python is serious about them.

```
p =30
```

```
'''
id() function returns the "identity" of the object.
The identity of an object - Is an integer
                          - Guaranteed to be unique
                          - Constant for this object during its lifetime.
'''
id(p)
```

**Memory address of the variable**

```
# Get the memory address in hexadecimal format using hex() function
hex(id(p))
```

```
p = 10
q = 10
r = p

# Checking the memory address of the variables
print(
    id(p) , id(q) , id(r),
    hex(id(p)) , hex(id(q)) , hex(id(r))
    )
```

```
station_id = "ADD001"     # Addis Ababa example code
lat, lon = 9.03, 38.74    # degrees
elev_m = 2355
print(station_id, lat, lon, elev_m)
```

## Variable Assigment

- Variable names in Python can contain alphanumerical characters
- a-z, A-Z, 0-9 and some special characters such as _. Variable names must start with a letter.
- By convention, variable names start with a lower-case letter
- There are a number of Python keywords that cannot be used as variable names.

```
intvar = 10 # Integer variable
floatvar = 2.57 # Float Variable
strvar = "Python Language" # String variable
print(intvar)
print(floatvar)
print(strvar)
```

```
intvar, floatvar, strvar = 10, 2.57, "Python Language"  # Using commas to separate
print(intvar)
print(floatvar)
print(strvar)
```

```
p1 = p2 = p3 = p4 = 44 # All variables pointing to same value
print(p1,p2,p3,p4)
```

---

## Data Types

## 1.2 Numbers

- Numbers in Python can either be integers `int` or floats `float`.
- Integer are real, finite, natural or whole numbers.
- Take an example: `1`, `2`, `3`, `4` are integers.
- Floats are numbers that have decimal points such as `4.6`, `6.0`, `7.7`.
- Note that `4.0` is considered as a float data type too.
- We can perform operations on numbers. The operations that we can perform include addition, multiplication, division, modular, etc...

```
int_var = 10
float_var = 12.8


print(type(int_var))
print(sys.getsizeof(val1)) # size of integer object in bytes
```

```
print(type(float_var))
print(sys.getsizeof(float_var)) # size of float object in bytes
```

```
# Unit conversions common in climate work
wind_ms = 6.0
wind_kmh = wind_ms * 3.6     # m/s -> km/h
precip_mm_day = 12.0
precip_m_day = precip_mm_day / 1000.0
print("Wind (km/h):", wind_kmh, "| Precip (m/day):", precip_m_day)
```

## Boolean

- Boolean data type can have only two possible values true or false.

```
bool1 = True

bool2 = False

print(bool1)
print(bool2)
print(type(bool1))
print(type(bool2))
```

## Numeric Operations

```
# Addition

1 + 100
```

```
# Multiplication

1 * 100
```

```
# Division

1 / 100
```

```
# Floor division

## Floor division is a type of division that rounds the result down to the nearest whol

7 // 2
```

```
# Modular (%)
# This is the remainder or a value remaining after dividing two numbers
# 100 / 1 = 100, remainder is 0

10 % 2
```

```
# Powers
# 1 power any number is 1 always

1 ** 100
```

```
2 ** 2
```

## 1.3 Strings

- String is a sequence of characters.
- Strings are one of the commonly used and important data types.
- Strings are expressed in either `"..."` or `'...'`.

We can manipulate strings in many ways. A simple example is to concat the strings.

```
str_var = 'One'
str_var2 = 'Hundred'
```

```
str_var + str_var2
```

```
str_var + ' ' + 'and' + ' '+ str_var2 + '.'
```

```
# We can use print() to display a string

print(" This is a string")
```

```
# We can also compare strings to check whether they are similar.
# If they are similar, case by case, comparison operator returns true. Else false

"A string" == "a string"
```

```
"A string" == "A string"
```

### Strings Methods

- Python provides many built-in methods for manipulating strings.

As a programmer, knowing typical string methods and how to use them will give you a real leverage when working with strings.

```
sentence = 'this IS A String'
```

```
# Case capitalization
# It return the string with first letter capitalized and the rest being lower cases.

sentence.capitalize()
```

```
# Given a string, convert it into title (each word is capitalized)

sentence_2 = 'this is a string to be titled'
sentence_2.title()
```

```
# Converting the string to upper case

sentence.upper()
```

```
# Converting the string to upper case

sentence.lower()
```

```
# Splitting the string

sentence.split()
```

- You can use `replace()` method to replace some characters in string with another characters.
- Replace method takes two inputs: characters to be replaced, and new characters to be inserted in string, `replace('characters to be replaced', 'new characters')`.

Example, given the string "This movie was awesome", replace the world `movie` with `project`.

```
stri = "This movie was awesome"
stri.replace('movie', 'project')
```

```
# In the following string, replace all spaces with `%20'

stri_2 = "The future is great"
stri_2.replace(' ', '%20')
```

```
stri_3 = "Climate "
stri_3 = stri_3*5
stri_3
```

```
len(stri_3)
```

```
mystr = "Hello World"
```

```
mystr[0] # First character in string
```

## String Indexing

```
mystr[-1]  # Last character in string
```

## String Slicing

```
mystr[0:5]  # First five characters
# mystr[:5]  # From start to index 5
```

```
mystr[6:11] # From index 6 to index 10
# mystr[6:]  # From index 6 to end
```

```
mystr[-5:]  # Last five characters
```

```
del mystr # Delete a string
print(mystr)
```

```
mystr2 = "   Hello Everyone  "
mystr2
```

```
mystr2.strip() # Remove leading and trailing whitespace
```

```
mystr2.rstrip() # Remove trailing whitespace
```

```
mystr2.lstrip() # Remove leading whitespace
```

```
mylist =mystr2.split()
mylist
```

```
station_name = "Addis Ababa Observatory"
msg = f"Station {station_id} ({station_name}) at {lat:.2f} deg, {lon:.2f} deg"
msg
```

---

# 2. Data Structures

- Data structures are used to organize and store the data.
- Python has 4 main data structures: `Lists`, `Dictionaries`, `Tuples` and `Sets`.

## 2.1 List

- A list is a set of ordered values.
- Each value in a list is called an `element` or `item` and can be identified by an index.
- A list supports different data types, we can have a list of integers, strings, and floats.

What we will see with Python lists:

- Creating a list
- Accessing elements in a list
- Slicing a list
- Changing elements in a list
- Traversing a list
- Operations on list
- Nested lists
- List methods
- List and strings

### Creating a List

A python list can be created by enclosing elements of similar or different data type in square brackets `[...]`, or with `range()` function.

```
# Creating a list

week_days = ['Mon', 'Tue', 'Wed', 'Thur','Fri']
even_numbers = [2, 4, 6, 8, 10]
mixed_list = ['Mon', 1, 'Tue', 2, 'Wed', 3]

# Displaying elements of a list
print(week_days)
```

```
print(even_numbers)
print(mixed_list)
```

### Accessing the elements of the list

We can access the a given element of the list by providing the index of the element in a bracket. The index starts at 0 in Python.

```
# Accessing the first elements of the list
week_days = ['Mon', 'Tue', 'Wed', 'Thur','Fri']
week_days[0]
```

```
# Get the third element of the list
even_numbers = [2, 4, 6, 8, 10]
even_numbers[2]
```

```
# Getting the last element of the list
even_numbers = [2, 4, 6, 8, 10]
print(even_numbers[-1])
```

### Slicing a list

Given a list, we can slice it to get any parts or combination of its elements forming another list.

```
# Get the elements from index 0 to 2. Index 2 is not included.

week_days = ['Mon', 'Tue', 'Wed', 'Thur','Fri']
week_days[0:2]
```

```
# Get elements from the last fourth elements to the first
# -1 starts at the last element 'Fri', -2 second last element `Thur'..... -4 to 'Tue'

week_days[-4:]
```

```
# Get all elements up to the fourth index

week_days[:4]
```

```
# Get all elements from the second to the last index

week_days[2:]
```

You can use `[:]` to copy the entire list.

```
# Get all elements in the list
week_days[:]
```

### Changing elements in a list

Python lists are mutable. We can delete or change the elements of the list.

```
names = ['James', 'Jean', 'Sebastian', 'Prit']
names
```

```
# Change 'Jean' to 'Nyandwi' and 'Sebastian' to 'Ras'

names[1:3] = ['Nyandwi', 'Ras']
names
```

```
# Change 'Prit' to Sun

names[-1] = 'Sun'
names
```

```
# Change `James` to `yonas`

names[0] = 'yonas'
names
```

In order to delete a given element in a list, we can empty slice it but the better way to delete element is to use `del` keyword.

```
# Delete Nyandwi in names list

del names[1]
names
```

- If you know the index of the element you want to remove, you can use `pop()`.
- If you don't provide the index in pop(), the last element will be deleted.

```
names = ['James', 'Jean', 'Sebastian', 'Prit']
names.pop(2)
names
```

Also, we can use `remove()` to delete the element provided inside the remove() method.

```
names = ['James', 'Jean', 'Sebastian', 'Prit']
names.remove('James')
names
```

We can also use `append()` to add element to the list.

```
# Adding the new elements in list

names = ['James', 'Jean', 'Sebastian', 'Prit']
names.append('Jac')
names.append('Jess')
names
```

### Operations on list

```
# Concatenating two lists

a = [1,2,3]
b = [4,5,6]

c = a + b

c
```

```
# We can also use * operator to repeat a list a number of times

[None] * 5
```

```
# Creating a list with repeated elements
[True] * 4
```

```
# Creating a list with repeated elements
[1,2,4,5] * 2
```

### Nested lists

```
# Creating a list in other list

nested_list = [1,2,3, ['a', 'b', 'c']]
```

```
# Get the ['a', 'b', 'c'] from the nested_list

nested_list[3]
```

```
nested_list[3][0]  # Get 'a' from the nested list
```

```
# Indexing and slicing a nested list is quite similar to normal list

nested_list[1]
```

### List Methods

- Python also offers methods which make it easy to work with lists.
- We already have been using some list methods such as `pop()` and `append()` but let's review more other methods.

```
# Sorting a list with sort()

even_numbers = [2,14,16,12,20,8,10]

even_numbers.sort()

even_numbers
```

```
# Reversing a string with reverse()

even_numbers.reverse()
even_numbers
```

```
# Adding other elements to a list with append()

even_numbers = [2,14,16,12,20,8,10]

even_numbers.append(40)
even_numbers
```

```
# Removing the first element of a list

even_numbers.remove(2)
even_numbers
```

```
## Return the element of the list at index x

even_numbers = [2,14,16,12,20,8,10]

## Return the item at the 1st index

even_numbers.pop(1)

week_days = ['Mon', 'Tue', 'Wed', 'Thur','Fri']

even_numbers
```

```
# pop() without index specified will return the last element of the list

even_numbers = [2,14,16,12,20,8,10]
even_numbers.pop()
even_numbers
```

```
# Count a number of times an element appear in a list

even_numbers = [2,2,4,6,8,2]
even_numbers.count(2)
```

### List and strings

We previously have learned about strings. Strings are sequence of characters. List is a sequence of values.

```
# We can convert a string into list

stri = 'Apple'

list(stri)
```

```
# Splitting a string produces a list of individual words

stri_2 = "List and Strings"
stri_2.split()
```

The split() string method allows to specify the character to use a a boundary while splitting the string. It's called delimiter.

```
stri_3 = "state-of-the-art"
```

```
stri_3.split('-')
```

```

```

---

## 2.2 Dictionaries

- Dictionaries are powerful Python data structure that are used to store data of `key and `values`.
- A dictionary is a collection of key and values. A dictionary stores a mapping of keys and values. A key is what we can refer to index.
- It is unordered, changeable, and does not allow duplicate keys.

What we will see:

- Creating a dictionary
- Accessing values and keys in dictionary
- Solving counting problems with dictionary
- Traversing a dictionary
- The setdefault() method

**Creating a dictionary**

- We can create with a `dict()` function and add items later
- or insert keys and values right away in the curly brackets { }.

Let's start with dict() function to create an empty dictionary.

```
# Creating a dictionary

countries_code = dict()
print(countries_code)
```

You can verify it's a dictionary by passing it through `type()`.

```
type(countries_code)
```

Let's add items to the empty dictionary that we just created.

```
# Adding items to the empty dictionary.
```

```
countries_code["Kenya"] = 254
```

```
countries_code
```

Let's create a dictionary with {}. It's the common way to create a dictionary.

```
countries_code = {
        "Ethiopia": 251,
        "Kenya": 254,
        "Rwanda":250,
        "Uganda": 49,
        "Tanzania": 91,
}

countries_code
```

To add key and values to a dictionary, we just add the new key to [ ] and set its new value. See below for example...

```
countries_code['Djibouti'] = 253
countries_code
```

**Accessing the values and keys in a dictionary**

Just like how we get values in a list by using their indices, in dictionary, we can use a key to get its corresponding value.

```
# Getting the code of Rwanda

countries_code["Rwanda"]
```

We can also check if a key exists in a dictionary by using a classic `in` operator.

```
"India" in countries_code
```

```
# Should be False

"Kenya" in countries_code
```

To get all the keys, value, and items of the dictionary, we can respectively use `keys()`, `values()`, and `items()` methods.

```
# Getting the keys and the values and items of the dictionary

dict_keys = countries_code.keys() # Get all keys of the dictionary
dict_values = countries_code.values() # Get all values of the dictionary
dict_items = countries_code.items() # Get all items (key-value pairs) of the dictionary

print(f"Keys: {dict_keys}\n Values:{dict_values}\n Items:{dict_items}")
```

Lastly, we can use `get()` method to return the value of a specified key. Get method allows to also provide a value that will be returned in case the key doesn't exists. This is a cool feature!!

```
# Get the value of the Kenya

countries_code.get('Kenya')
```

```
countries_code
```

**Traversing a dictionary**

We previously used for loop in dictionary to iterate through the values. Let's review it again.

```
countries_code
```

```
for country in countries_code:
    print(country)
```

We can also iterate through the items(key, values) of the dictionary.

```
for country, code in countries_code.items():
    print(country, code)
```

**The setdefault() Method**

- The setdefault() method allows you to set a value of a given key that does not already have a key.
- This can be helpful when you want to update the dictionary with a new value in case the key you are looking for does not exist.

```
countries_code.setdefault("Somalia", 252)
```

```
countries_code
```

```
# Remove a key-value pair
countries_code.pop("UK")
```

```
countries_code
```

```
station = {
    "id": "ADD001",
    "name": "Addis Ababa Observatory",
    "coords": (9.03, 38.74),
    "daily_precip_mm": [0.0, 2.1, 0.0, 5.4, 1.2],
}
station["name"], station["coords"]
```

***Summarizing dictionary***

- Dictionaries are not ordered and they can not be sorted - list are ordered (and unordered) and can be sorted.
- Dictionary can store data of different types: floats, integers and strings and can also store lists.

---

## 2.3 Tuples

- Tuple is similar to list but the difference is that you can't change the values once it is defined (termed as `immutability`).
- Due to this property it can be used to keep things that you do not want to change in your program.
- Example can be a country codes, zipcodes, etc...

```
coords = (9.03, 38.74)
lat, lon = coords    # tuple unpacking
lat, lon
```

```
# Empty tuple
tup1 = ()
```

```
 # tuple of integers numbers
tup2 = (10,30,60)

print(tup2)
```

```
# tuple of float numbers
tup3 = (10.77,30.66,60.89)
print(tup3)
```

```
 # tuple of strings
tup4 = ('apple', 'banana', 'cherry')
print(tup4)
```

```
 # Nested tuples
tup5 = ('Asif', 25 ,(50, 100),(150, 90))

print(tup5)
```

```
# accessing elements in nested tuples

tup5[2]  # Accessing the third element which is a tuple (50, 100)
```

```
tup5[2][0]  # Accessing the first element of the third element which is 50
```

```
len(tup5) #Length of list
```

```
tup = (1,4,5,6,7,8,1)
```

```
# Indexing

tup[4]
```

```
tup[-1]
```

```
## Tuples are not changeable. Running below code will cause an error

tup[2] = 10
```

```
# You can not also add other values to the tuple. This will be error
tup.append(12)
```

```
# Get the length of a tuple
print(len(tup))
```

```
# Count occurrences of an element
print(tup.count(1))
```

## Tuple Slicing

```
# tuple slicing
tup = (1,4,5,6,7,8,1)

tup[-1]  # Accessing the last element which is 1
tup[0]   # Accessing the first element which is 1
tup[0:3] # Accessing elements from index 0 to 2
tup[-1:3] # Accessing elements from last to index 3
```

## Sorting a tuple with sorted() function

```
tup = (5,2,8,1,4)

sorted(tup)  # Returns a sorted list of the tuple elements
```

```
sorted(tup, reverse=True)  # Returns a sorted list of the tuple elements in descending
```

---

## 2.4 Sets

Sets are used to store the unique elements. They are not ordered like list.

```
codes = ["RA","RA","TS","BR","TS","NSW","DZ"]
unique_codes = set(codes)
unique_codes
```

```
set_1 = {1,2,3,4,5,6,7,8}

set_1
```

```
set_2 = {1,1,2,3,5,3,2,2,4,5,7,8,8,5}
set_2
```

```
len(set_2)
```

As you can see, set only keep unique values. There can't be a repetition of values.

```
# List Vs Set

odd_numbers = [1,1,3,7,9,3,5,7,9,9]

print("List:{}".format(odd_numbers))

print("********")

set_odd_numbers = {1,1,3,7,9,3,5,7,9,9}

print("Set:{}".format(set_odd_numbers))
```

---

# 3. Comparison and Logic operators

`Comparison` operators are used to compare values. It will either return true or false.

```
## Greater than
100 > 1
```

```
## Equal to

100 == 1
```

```
## Less than

100 < 1
```

```
## Greater or equal to

100 >= 1
```

```
## Less or equal to
```

```
100 <= 1
```

```
'Intro to Python' == 'intro to python'
```

```
'Intro to Python' == 'Intro to Python'
```

`Logic` operators are used to compare two expressions made by comparison operators.

- Logic `and` returns true only when both expressions are true, otherwise false.
- Logic `or` returns true when either any of both expressions is true. Only false if both expressions are false.
- Logic `not` as you can guess, it will return false when given expression is true, vice versa.

```
100 == 100 and 100 == 100
```

```
100 <= 10 and 100 == 100
```

```
100 == 10 or 100 == 100
```

```
100 == 10 or 100 == 10
```

```
not 1 == 2
```

```
not 1 == 1
```

---

# 4. Control Flow

We will cover:

- If statement
- For loop
- While loop

## 4.1 If, Elif, Else

Structure of If condition:

```
if condition:
  do something

else:
  do this
```

```
if 100 < 2:

  print("This will not be printed")
```

```
if 100 > 2:

  print("This will be printed out")
```

```
if 100 < 2:

  print("As expected, no thing will be displayed")

else:
  print('Printed')
```

```
# Let's assign a number to a variable name 'jean_age' and 'yannick_age'

john_age = 30
luck_age = 20

if john_age > luck_age:
  print("John is older than Luck")

else:
  print(" John is younger than Luck")
```

```
# Let's use multiple conditions

john_age = 30
luck_age = 20
yan_age = 30

if john_age < luck_age:
  print("John is older than Luck")

elif yan_age == luck_age:
  print(" Yan's Age is same as Luck")
```

```
elif luck_age > john_age:
  print("Luck is older than John")

else:
  print("John's age is same as Yan")
```

We can also put if condition into one line of code. This can be useful when you want to make a quick decision between few choices.

Here is the structure:

`'value_to_return_if_true' if condition else 'value_to_return_if_false'`

Let's take some examples...

```
# Example 1: Return 'Even' if below num is 'Even' and `Odd` if not.

num = 45

'Even' if num % 2 == 0 else 'Odd'
```

```
# Example 2: Return True if a given element is in a list and False if not

nums = [1,2,3,4,5,6]

True if 3 in nums else False
```

```
pr = 8.0  # mm/day
if pr == 0:
    print("Dry day")
elif pr < 2.5:
    print("Light rain")
else:
    print("Rainy day")
```

## 4.2 For Loop

For loop is used to iterate over list, string, tuples, or dictionary.

Structure of for loop:

```
for item in items:
  do something
```

```
even_nums = [2,4,6,8,10]

for num in even_nums:
  print(num)
```

```
week_days = ['Mon', 'Tue', 'Wed', 'Thur','Fri']

for day in week_days:
  print(day)
```

```
sentence = "It's been a long time learning Python. I am revisiting the basics!!"

for letter in sentence:
  print(letter)
```

```
sentence = "It's been a long time learning Python. I am revisiting the basics!!"

# split is a string method to split the words making the string

for letter in sentence.split():
  print(letter)
```

```
# For loop in dictionary

countries_code = { "United States": 1,
                   "India": 91,
                   "Germany": 49,
                   "China": 86,
                   "Rwanda":250
               }

for country in countries_code:
  print(country)
```

```
for code in countries_code.values():
  print(code)
```

For can also be used to iterate over an sequence of numbers.

- `Range` is used to generate the sequence of numbers.

```
for number in range:
  do something
```

```
for number in range(10):
  print(number)
```

```
for number in range(10, 20):
  print(number)
```

One last thing about `for loop` is that we can use it to make a list. This is called list comprehension.

```
letters = []

for letter in 'MachineLearning':
  letters.append(letter)

letters
```

The above code can be simplified to the following code:

```
letters = [letter for letter in 'MachineLearning']

letters
```

```
# Accumulate rain until we reach 10 mm
rain = [0.0, 0.5, 2.1, 7.8, 0.0, 0.0, 3.0]
total = 0.0
for r in rain:
    total += r # total = total + r
total
```

## 4.3 While loop

While loop will executes the statement(s) as long as the condition is true.

Structure of while loop

```
while condition:
  statement(s)
```

```
a = 10
while a < 20:
    print('a is: {}'.format(a)) # Display the value of a
    a = a + 1
```

```
# Walk forward until cumulative rain >= 10 mm or we run out of days
i, cumulative = 0, 0.0
while i < len(rain) and cumulative < 10.0:
    cumulative += rain[i]
    i += 1 # i = i + 1
i, cumulative
```

---

# 5. Functions

- Functions are used to write codes or statements that can be used multiple times with different parameters.
- One fundamental rule in programming is "DRY" or Do not Repeat Yourself.

This is how you define a function in Python:

```
def function_name(parameters):

  """
  This is Doc String
  You can use it to notes about the functions
  """
  statements

  return results
```

- `function_name` is the name of the function. It must not be similar to any built in functions. We will see built in functions later.
- `parameters` are the values that are passed to the function.
- `Doc String` is used to add notes about the function. It is not a must to use it but it is a `good practice`.
- `return` specify something or value that you want to return everytime you call or run your function.

```
# Function to add two numbers and return a sum

def add_nums(a,b):

  """
  Function to add two numbers given as inputs
  It will return a sum of these two numbers
  """
```

```
  sum = a+b

  return sum
```

```
add_nums(2,4)
```

```
add_nums(4,5)
```

```
# Displaying the doc string noted early

print(add_nums.__doc__)
```

```
def activity(name_1, name_2):

  print("{} and {} are playing basketball!".format(name_1, name_2))
```

```
activity("yonas", "Tamirat")
```

```
def c_to_k(t_c: float) -> float:
    """Convert Celsius to Kelvin.

    Parameters
    ----------
    t_c : float
        Temperature in C.
    Returns
    -------
    float
        Temperature in Kelvin.
    """
    return t_c + 273.15

c_to_k(30.0)
```

---

# 6. Lamdba Functions

- There are times that you want to create anonymous functions.
- These types of functions will only need to have one expressions.

```
## Sum of two numbers
```

```
def add_nums(a,b):

  sum = a+b

  return sum

add_nums(1,3)
```

We can use lambda to make the same function in just one line of code! Let's do it!!

```
sum_of_two_nums = lambda c,d: c + d

sum_of_two_nums(4,5)
```

```
c_to_f = lambda c: c * 9/5 + 32
list(map(c_to_f, [20.0, 25.0, 30.0]))
```

---

# 7. Built in Functions

Python being a high level programming language, it has bunch of built in functions which make it easy to get quick computations done.

```
# using len() to count the length of the string

message = 'Do not give up!'
len(message)
```

```
# Using max() to find the maximum number in a list

odd_numbers = [1,3,5,7]
max(odd_numbers)
```

```
# Using min() to find the minimum number in a list

min(odd_numbers)
```

```
# Sorting the list with sorted()

odd_numbers = [9,7,3,5,11,13,15,1]

sorted(odd_numbers)
```

Let's learn two more useful built functions: they are `map` and `filter`.

## 7.1 Map function

- Map gives you the ability to apply a function to an iterable structures such as list.
- When used with a list for example, you can apply the function to every element of the list.

Let's see how it works.

```
def cubic(number):
  return number ** 3
```

```
num_list = [0,1,2,3,4]
```

```
# Applying `map` to the num_list to just return the list where each element is cubed...

list(map(cubic, num_list))
```

```
# Convert a list of C to K using map
temps_c = [24.0, 26.5, 29.0]
temps_k = list(map(lambda c: c + 273.15, temps_c))
temps_k
```

## 7.2 Filter function

```
def odd_check(number):

  return number % 2 != 0

# != is not equal to operation
```

```
# Create a list of numbers
num_list = [1,2,4,5,6,7,8,9,10,11]

# Applying `filter` to the num_list to just return the odd numbers in the list
list(filter(odd_check, num_list))
```

```
# Keep rainy days >= 1 mm
daily_mm = [0.0, 0.6, 1.2, 3.5, 0.0, 5.0]
```

```
wet_days = list(filter(lambda mm: mm >= 1.0, daily_mm))
wet_days
```

---

# 8. More Useful Python Stuff

Python is an awesome programming language that has a lot of useful functions.

Let's see more useful things you may need beyond what's we just saw already.

## 8.1 Enumerate Function

Enumerate function convert iterable objects into enumerate object. It basically returns a tuple that also contain a counter.

That's sounds hard, but with examples, you can see how powerful this function is...

```
seasons = ['Spring', 'Summer', 'Fall', 'Winter']

list(enumerate(seasons))
```

As you can see, each element came with index counter automatically. The counter initially start at 0, but we can change it.

```
list(enumerate(seasons, start=1))
```

Here is another example:

```
class_names = ['Spring', 'Summer', 'Fall', 'Winter']

for index, class_name in enumerate(class_names, start=0):
  print(index,'-',class_name)
```

```
# Find first day exceeding 35C
tmax = [30.0, 31.5, 34.9, 35.1, 33.0]
idx = None
for i, val in enumerate(tmax):
    if val > 35.0:
        idx = i
        break
idx
```

## 8.2 Zip Function

Zip is an incredible function that takes two iterators and returns a pair of corresponsing elements as a tuple.

```
name = ['Jessy', 'Joe', 'Jeannette']
role = ['ML Engineer', 'Web Developer', 'Data Engineer']

zipped_name_role = zip(name, role)
zipped_name_role
```

The zip object return nothing. In order to show the zipped elements, we can use a list. It's also same thing for enumerate you saw above.

```
list(zipped_name_role)
```

# 9. Python Modules

- A Python module is a file containing `Python code`, such as `functions`, `variables`, or `classes`, that can be imported and used in another Python program.
- Modules are reusable code libraries in Python.
- You can import entire modules or specific functions/variables.
- Built-in modules are ready to use, while external modules need installation with pip.

Types of Loading Modules - Import the entire module: You can import an entire module and access its functions or variables using the module name.

```
# Importing the math module
import math

# Use the sqrt function from the math module
result = math.sqrt(16)
print(result)
```

## Importing Specific Functions or Variables

```
# Importing only the sqrt function from math

from math import sqrt

# Directly use the sqrt function
```

```
result = sqrt(25)
print(result)
```

## Using an Alias ‘

- You can give a module or function an alias (short name) for convenience.

```
# Using an alias for math
import math as m

# Use the alias to call functions
result = m.pow(2, 3)  # 2 raised to the power of 3
print(result)
```

## Importing All Contents of a Module

- You can import everything from a module, but this is not recommended as it may lead to naming conflicts.

```
# Importing all contents of math
from math import *

# Use functions without the module name
result = factorial(5)
print(result)
```

## Built-in vs. External Modules

Built-in Modules: Python comes with several modules built-in. Examples include:

- math (mathematical functions)
- os (interacting with the operating system)
- sys (system-specific parameters and functions)
- random (random number generation)

External Modules: are not built into Python and need to be installed using a package manager like pip.

- SciPy
- Scikit-learn
- TensorFlow
- PyTorch
- Keras

## Getting Help with Modules

- Using the help() Function: provides detailed documentation about a module.

```
# Getting help on the math module
import math
help(math)
```

- Using the dir() Function: lists all attributes and functions available in a module.

```
# Listing all functions in the math module
import math
print(dir(math))
```

## Creating Your Own Module

- Create a Python file (e.g., my_module.py) with functions or variables.

```
def greet(name):
    return f"Hello, {name}!"
```

- Using the custom module: import your module in another Python file or script.

```
import my_module

message = my_module.greet("Yonas")
print(message)
```

```
module_src = '''\
def heat_index_c(t_c: float, rh: float) -> float:
    """Approximate heat index in C (demo only)."""
    t_f = t_c * 9/5 + 32
    hi_f = (-42.379 + 2.04901523*t_f + 10.14333127*rh - 0.22475541*t_f*rh
            - 6.83783e-3*t_f*t_f - 5.481717e-2*rh*rh
            + 1.22874e-3*t_f*t_f*rh + 8.5282e-4*t_f*rh*rh
            - 1.99e-6*t_f*t_f*rh*rh)
    return (hi_f - 32) * 5/9
'''
open("climate_utils.py","w").write(module_src)
```

```
from climate_utils import heat_index_c
heat_index_c(32.0, 60.0)
```

# 10. Common Python Errors

- Python errors, also called **exceptions**, occur when something goes wrong during the execution of your program.
- Understanding these errors is key to debugging.

## SyntaxError

- SyntaxError occurs when the Python code violates the syntax rules.

```
print("Hello
```

```
print("Hello")  # Corrected version
```

## NameError

- happens when you use a variable or function that hasn’t been defined.

```
print(value)  # Trying to print a variable that hasn't been defined
```

```
# Corrected version:
value = 10
print(value)  # Output: 10
```

## TypeError

- TypeError occurs when an operation is performed on incompatible data types.

```
number = 5
text = "hello"

print(number + text)  # Trying to add a number to a string
```

```
# Corrected version:
print(f"{number} {text}")  # Output: 5 hello
```

## IndexError

- occurs when you try to access an index that’s out of range in a list or similar data structure.

```
numbers = [1, 2, 3]

print(numbers[3])  # Accessing an index that doesn't exist
```

```
# Corrected version:
print(numbers[2])
```

## Debugging Basics

- Debugging helps `identify` and `fix` errors in your program.
- Python provides tools like `try-except` for error handling preventing the program from crashing abruptly when an exception occurs.
- `Graceful Recovery` : can provide `informative error messages` or take `alternative actions` to recover from the error.

```
10 / 0
```

```
try:
    result = 10 / 0  # Division by zero causes ZeroDivisionError

except ZeroDivisionError: # except block catches specific errors and prevents the progr
    print("Error: Cannot divide by zero!")
```

## Catch different types of errors

```
text
```

```
value = int("text")
```

```
try:
    value = int("text")  # This will cause a ValueError
except ValueError:
    print("Error: Cannot convert text to an integer!")
except TypeError:
    print("Error: There was a type mismatch!")
```

## Catch any error

- Using `Exception` catches any type of error.
- The variable `e` contains information about the error.

```
try:
    result = 10 / 0  # This will cause a ZeroDivisionError
except Exception as e:
    print(f"An error occurred: {e}")
```

## Exercise

1) Create a loop that prints all the odd numbers between 1 and 20? 2) Write a function that takes a number and returns its square? 3) Create a function that takes two numbers and returns their average? 4) Write a function with a default argument that prints a farewell message (e.g., “Goodbye, Guest!”)? 5) Write a program that handles a ValueError when converting user input to an integer? 6) Use a try-except block to handle a FileNotFoundError when opening a non-existent file? 7) Debug a program by printing variable values at different points?

---