Python for Data Analysis: Methods & Tools

Python for everyday people

Written by Felipe Dominguez - Professor Adjunct
Hult International Business School

Chapter 02 - Python Fundamentals.

Numbers, Strings, Lists, Dictionaries, and more!

1. Type of Data

It was mentioned in on *Chapter 01 - Help, Print, Dynamic Strings, and User Input*, that input from the user is always stored as a string. As you can imagine by now, strings are one of the data types you can use in Python. Furthermore, there are 7 fundamental data types in Python: **numerics, strings, sequence types, binary, mapping, boolean, and sets**.

All of these data types will be covered in depth throughout the course. For a brief overview of each type, refer to Table 1.1.

Data Type	|	Description
Numeric	|	Holds numeric values (int, float, complex)
String	|	Sequence of characters wrapped by "" or '' that can be read as text
Boolean	|	Two constant values that represent truth (True and False)
Sequence types	|	Allows you to store multiple values in an organized and efficient way (Lists, tuples, and range)
Binary	|	Allows to manipulate binary data in Python (bytes, bytearray, and memoryview)
Set	|	Unordered collection of distinct hashable objects
Mapping	|	A mapping object maps hashable values to arbitrary objects. Mappings are mutable objects (dictionaries)

Table 1.1: Python data types.

2. Numbers & Strings

2.1. Numbers & Maths

Let's start with numbers. Python store numeric values into three types: **integer, float, and complex**

```
a. Integer: Whole number, positive or negative, without
decimals, of unlimited length.
b. Float: Number, positive or negative, containing one or more
decimals.
c. Complex: Number of the form a + bj, where a and b are real
numbers.
```

You can convert a number to a specific numeric type using the built-in functions **int()**, **float()**, and **complex()** by wrapping the number in the function. Furthermore, it is important to be mindful of the data types when working with numbers, as the type of the result may vary depending on the types of the operands. If both operands are of the same type (e.g., both integers), the result will also be of that type. If the operands are of different types (e.g., integer and float), the result will be of the more general data type.

For examples, see code samples 2.1.1 and 2.1.2.

Note: *If you use the **int()** function on a float number, it will return the whole part of the number. I.e., **int(5.6)** will return **5**.*

In [1]:
```
# Code 2.1.1.

# This is a integer
integer_1 = int(5.667)
integer_2 = 2
print("These numbers are integers:", integer_1, ",", integer_2)

# This is a float
float_1 = float(5.667)
print("This is a float:", float_1)

# this is a complex
# complext(real, imagine)
complex_1 = complex(2, 3)
print("This is a complex:", complex_1)

print(f"""
{"-"*40}
Let's confirmt the type of the variables using \
the "type" function:
- integer_1 is {type(integer_1)}
- integer_2 is {type(integer_2)}
- float_1   is {type(float_1)}
- complex_1 is {type(complex_1)}
""")
```

```
These numbers are integers: 5 , 2
This is a float: 5.667
This is a complex: (2+3j)

----------------------------------------
Let's confirmt the type of the variables using the "type" function:
- integer_1 is <class 'int'>
- integer_2 is <class 'int'>
- float_1   is <class 'float'>
- complex_1 is <class 'complex'>
```

In [2]:

```
# Code 2.1.2.

# Let's mix variables and check there types

# integer * integer
mult_integer = integer_1 * integer_2

# integer + float
sum_int_float = integer_1 + float_1

# float + complex
sum_float_complex = float_1 + complex_1

print(f"""
{"-"*50}
- The multiplication of integers ({integer_1} * {integer_2}) gives:
({mult_integer}, {type(mult_integer)}
- The sum of integer and float ({integer_1} + {float_1}) gives:
{type(sum_int_float)}
- The sum of float and complex gives:
{type(sum_float_complex)}
""")
```

```
--------------------------------------------------
- The multiplication of integers (5 * 2) gives:
(10, <class 'int'>
- The sum of integer and float (5 + 5.667) gives:
<class 'float'>
- The sum of float and complex gives:
<class 'complex'>
```

What are classes?

As you can see in the output of code samples 2.1.1 and 2.1.2, Python describes each variable using the word **"class"** and the **name of the data type**. In Python, a class is a template for creating objects. It serves as a blueprint that defines the attributes and methods common to all objects of a certain type.

Since everything is an **object** in Python programming, when you create a variable, you are creating an instance (or object) of a particular class (data type). For example, when you use the int() function to convert a variable to an integer, you are creating an object of the int class. ***Note:*** *Classes help us to ensure that objects behave as they are supposed to behave.*

2.1.1. More mathematical operations

Python provides several built-in mathematical operations, such as addition, subtraction, multiplication, or division, which can be used directly in your code. However, to access more advanced mathematical operations such as logarithms and trigonometry, you will need to import the **math package**.

To get started, let's first ask for help (see code 2.1.1.1) and then implement the desired operation (see code 2.1.1.2).

In [4]:
```
# Code 2.1.1.1
import math
# Math functions
# Check all the functions we can use!
help(math)
```

```
Help on module math:

NAME
    math

MODULE REFERENCE
    https://docs.python.org/3.9/library/math

    The following documentation is automatically generated from the Python
    source files.  It may be incomplete, incorrect or include features that
    are considered implementation detail and may vary between Python
    implementations.  When in doubt, consult the module reference at the
    location listed above.

DESCRIPTION
    This module provides access to the mathematical functions
    defined by the C standard.

FUNCTIONS
    acos(x, /)
        Return the arc cosine (measured in radians) of x.

        The result is between 0 and pi.

    acosh(x, /)
        Return the inverse hyperbolic cosine of x.

    asin(x, /)
        Return the arc sine (measured in radians) of x.

        The result is between -pi/2 and pi/2.

    asinh(x, /)
        Return the inverse hyperbolic sine of x.

    atan(x, /)
        Return the arc tangent (measured in radians) of x.

        The result is between -pi/2 and pi/2.

    atan2(y, x, /)
        Return the arc tangent (measured in radians) of y/x.

        Unlike atan(y/x), the signs of both x and y are considered.

    atanh(x, /)
        Return the inverse hyperbolic tangent of x.

    ceil(x, /)
        Return the ceiling of x as an Integral.

        This is the smallest integer >= x.

    comb(n, k, /)
        Number of ways to choose k items from n items without repetition and w
ithout order.

        Evaluates to n! / (k! * (n - k)!) when k <= n and evaluates
        to zero when k > n.
        Also called the binomial coefficient because it is equivalent
        to the coefficient of k-th term in polynomial expansion of the
        expression (1 + x)**n.

        Raises TypeError if either of the arguments are not integers.
        Raises ValueError if either of the arguments are negative.

    copysign(x, y, /)
        Return a float with the magnitude (absolute value) of x but the sign o
f y.

        On platforms that support signed zeros, copysign(1.0, -0.0)
        returns -1.0.

    cos(x, /)
        Return the cosine of x (measured in radians).

    cosh(x, /)
        Return the hyperbolic cosine of x.

    degrees(x, /)
        Convert angle x from radians to degrees.

    dist(p, q, /)
        Return the Euclidean distance between two points p and q.

        The points should be specified as sequences (or iterables) of
        coordinates.  Both inputs must have the same dimension.

        Roughly equivalent to:
            sqrt(sum((px - qx) ** 2.0 for px, qx in zip(p, q)))

    erf(x, /)
        Error function at x.

    erfc(x, /)
        Complementary error function at x.

    exp(x, /)
        Return e raised to the power of x.

    expm1(x, /)
        Return exp(x)-1.

        This function avoids the loss of precision involved in the direct eval
uation of exp(x)-1 for small x.

    fabs(x, /)
        Return the absolute value of the float x.

    factorial(x, /)
        Find x!.

        Raise a ValueError if x is negative or non-integral.

    floor(x, /)
        Return the floor of x as an Integral.

        This is the largest integer <= x.
    fmod(x, y, /)
        Return fmod(x, y), according to platform C.

        x % y may differ.

    frexp(x, /)
        Return the mantissa and exponent of x, as pair (m, e).

        m is a float and e is an int, such that x = m * 2.**e.
        If x is 0, m and e are both 0.  Else 0.5 <= abs(m) < 1.0.

    fsum(seq, /)
        Return an accurate floating point sum of values in the iterable seq.

        Assumes IEEE-754 floating point arithmetic.

    gamma(x, /)
        Gamma function at x.

    gcd(*integers)
        Greatest Common Divisor.

    hypot(...)
        hypot(*coordinates) -> value

        Multidimensional Euclidean distance from the origin to a point.

        Roughly equivalent to:
            sqrt(sum(x**2 for x in coordinates))

        For a two dimensional point (x, y), gives the hypotenuse
        using the Pythagorean theorem:  sqrt(x*x + y*y).

        For example, the hypotenuse of a 3/4/5 right triangle is:

            >>> hypot(3.0, 4.0)
            5.0

    isclose(a, b, *, rel_tol=1e-09, abs_tol=0.0)
        Determine whether two floating point numbers are close in value.

          rel_tol
            maximum difference for being considered "close", relative to the
            magnitude of the input values
          abs_tol
            maximum difference for being considered "close", regardless of the
            magnitude of the input values

        Return True if a is close in value to b, and False otherwise.

        For the values to be considered close, the difference between them
        must be smaller than at least one of the tolerances.

        -inf, inf and NaN behave similarly to the IEEE 754 Standard.  That
        is, NaN is not close to anything, even itself.  inf and -inf are
        only close to themselves.

    isfinite(x, /)
        Return True if x is neither an infinity nor a NaN, and False otherwis
e.
    isinf(x, /)
        Return True if x is a positive or negative infinity, and False otherwi
se.

    isnan(x, /)
        Return True if x is a NaN (not a number), and False otherwise.

    isqrt(n, /)
        Return the integer part of the square root of the input.

    lcm(*integers)
        Least Common Multiple.

    ldexp(x, i, /)
        Return x * (2**i).

        This is essentially the inverse of frexp().

    lgamma(x, /)
        Natural logarithm of absolute value of Gamma function at x.

    log(...)
        log(x, [base=math.e])
        Return the logarithm of x to the given base.

        If the base not specified, returns the natural logarithm (base e) of
x.

    log10(x, /)
        Return the base 10 logarithm of x.

    log1p(x, /)
        Return the natural logarithm of 1+x (base e).

        The result is computed in a way which is accurate for x near zero.

    log2(x, /)
        Return the base 2 logarithm of x.

    modf(x, /)
        Return the fractional and integer parts of x.

        Both results carry the sign of x and are floats.

    nextafter(x, y, /)
        Return the next floating-point value after x towards y.

    perm(n, k=None, /)
        Number of ways to choose k items from n items without repetition and w
ith order.

        Evaluates to n! / (n - k)! when k <= n and evaluates
        to zero when k > n.

        If k is not specified or is None, then k defaults to n
        and the function returns n!.

        Raises TypeError if either of the arguments are not integers.
        Raises ValueError if either of the arguments are negative.
    pow(x, y, /)
        Return x**y (x to the power of y).
    
    prod(iterable, /, *, start=1)
        Calculate the product of all the elements in the input iterable.
        
        The default start value for the product is 1.
        
        When the iterable is empty, return the start value.  This function is
        intended specifically for use with numeric values and may reject
        non-numeric types.
    
    radians(x, /)
        Convert angle x from degrees to radians.
    
    remainder(x, y, /)
        Difference between x and the closest integer multiple of y.
        
        Return x - n*y where n*y is the closest integer multiple of y.
        In the case where x is exactly halfway between two multiples of
        y, the nearest even value of n is used. The result is always exact.
    
    sin(x, /)
        Return the sine of x (measured in radians).
    
    sinh(x, /)
        Return the hyperbolic sine of x.
    
    sqrt(x, /)
        Return the square root of x.
    
    tan(x, /)
        Return the tangent of x (measured in radians).
    
    tanh(x, /)
        Return the hyperbolic tangent of x.
    
    trunc(x, /)
        Truncates the Real x to the nearest Integral toward 0.
        
        Uses the __trunc__ magic method.
    
    ulp(x, /)
        Return the value of the least significant bit of the float x.

DATA
    e = 2.718281828459045
    inf = inf
    nan = nan
    pi = 3.141592653589793
    tau = 6.283185307179586

FILE
    /Users/fadominguez/opt/anaconda3/lib/python3.9/lib-dynload/math.cpython-39
-darwin.so
```

In [5]:
```
# Code 2.1.1.2.
```

```python
# Import math functions
import math

# Numeric Operations
a = 2
b = 5

# Sum / Difference
print(f"""
a + b = {a + b}
""")

# Multiplication / Division
print(f"""
a * b = {a * b}
""")

# a to the power of b
print(f"""
a^b = {math.pow(a,b)}
or
a^b = {a**b}
""")

# Logarithm
print(f"""
log_10(a) = {math.log10(a)}
""")

# sin(alpha). Input value in radians.
print(f"""
sin(pi/2) = {math.sin(math.pi/2)}
""")
```

```
a + b = 7


a * b = 10


a^b = 32.0
or
a^b = 32


log_10(a) = 0.3010299956639812


sin(pi/2) = 1.0
```

2.1.1. Overwriting numbers

Often you will find yourself with the need of overwriting numbers multiple times. Let's say for example, you want to count from 1 to 100. Instead of declaring 100 separate variables, you can simply overwrite a single variable 100 times. This technique is often used when you want to repeat a mathematical operation multiple times.

```
Declare and overwrite n variables
a = 1, a = 2, ..., a = n

Use math operations to overwrite a variable
x = x + 1
```

Given the frequent use of math operations in programming, Python has developed a built-in function to shorten the syntax for these operations. It is similar to the use of an apostrophe in English to denote possession, such as in the phrase "I'm" instead of "I am".

Table 2.1.1 lists the most common math abbreviations. To practice using these abbreviations, take a look at code sample 2.1.1.

Operand	Description
+=	addition and overwrite
-=	subtraction and overwrite
*=	multiplication and overwrite
/=	division and overwrite
%=	modulus and overwrite
**=	exponentiation and overwrite

Table 2.1.1. Common math abbreviations

In [4]:
```python
# Code 2.1.1.

# Declare x,y
x = 10
y = 10

# Add normal way
x = x + 10

# Abbreviate add
y +=10

print(f"""
x: {x}
y: {y}
x = y : {x==y}

""")
```

```
x: 20
y: 20
x = y : True
```

2.1.2. Randomness

Let's take a moment to discuss randomness. By definition, **randomness** refers to the lack of order, pattern, or purpose. Examples of random events include tossing a coin, rolling a dice, or selecting lottery numbers. For example, when starting a soccer match, the referee may toss a coin to randomly determine which team will start the game. There is a certain beauty in the unpredictability of random events, as you cannot know in advance what the outcome will be and therefore, you cannot control or manipulate it.

Randomness is often used in statistical sampling, scientific experiments, and program simulations to **avoid bias and create a more realistic representation of the world**. However, it can be challenging to replicate a truly random process. Random numbers can be divided into two categories: **pseudo-random** and **true-random**. Take a look at the following two sets of numbers:

```
i. 1/3, +1/5, −1/7, +1/9, −1/11, +1/13, −1/15,...
ii. 7, 8, 5, 3, 9, 8, 1, 6, 3, 3, 9,...
```

If I ask you to identify the next number in sequences i. and ii., you would probably realize that the next number in i. is +1/17, as there is a clear pattern present. However, you might conclude that sequence ii. is random, as there is no apparent pattern. In reality, **sequence ii. follows an algorithm**, each number represents one of the decimal parts of the operation π/4. This is a great example of a pseudo-random sequence, as there is a pattern or algorithm that determines the sequence of numbers. For most programming tasks (including this course), pseudo-random sequences are sufficient. However, for security encryption, true randomness is often required to prevent predictability.

Python has a built-in random library that allows you to generate pseudo-random numbers and use them in your code. The **randint()** function within this library returns a random integer within a specified rang

```
randint(a, b) method of random.Random instance
    Return random integer in range [a, b], including both start
and end points.
```

Finally, the random package also allows you to set a **seed** for a random sequence of numbers. The seed determines the starting point of the algorithm that generates the pseudo-random numbers. As previously mentioned, pseudo-random sequences are based on a specific algorithm, so setting the seed allows you to generate the same sequence of random numbers every time.

The ability to replicate results is crucial in data analysis and machine learning, as it allows you to test your code and share your findings with others. Imagine if every time you run a code you get a different result, how could you confirm you hypothesis? Without the ability to replicate results, it would not be possible to determine if your code is correct or to share your insights with others.

To see the effect of setting a seed, try running code samples 2.1.2.a and 2.1.2.b and observe the difference.

In [7]:

```
# Code 2.1.2.a.

# Import random package
import random

# Asking user number input
user_number = input(f"""
Hey!
Can you guess between 1 and 100
what's my favorite number?
""")

# Transform user input into int
user_number = int(user_number)

# python pseudo-randomly select a number
python_number = random.randint(1,100)

# print the result
print(f"""
Let's see!
You selected: {user_number}
I selected: {python_number}
Did you guess correctly? [True or False]
{user_number == python_number}
""")
```

```
Hey!
Can you guess between 1 and 100
what's my favorite number?
1

Let's see!
You selected: 1
I selected: 1
Did you guess correctly? [True or False]
True
```

In [8]:

```
# Code 2.1.2.b.

# Import random package
import random

# Set random seed
random.seed(1)

# Asking user number input
user_number = input(f"""
Hey!
Can you guess between 1 and 100
what's my favorite number?
""")

# Transform user input into int
user_number = int(user_number)

# python pseudo-randomly select a number
python_number = random.randint(1,100)

# print the result
print(f"""
Let's see!
You selected: {user_number}
I selected: {python_number}
Did you guess correctly? [True or False]
{user_number == python_number}
""")
```

```
Hey!
Can you guess between 1 and 100
what's my favorite number?
1

Let's see!
You selected: 1
I selected: 18
Did you guess correctly? [True or False]
False
```

2.2 Strings & more strings

You have previously worked with strings on Chapter 01. As mentioned in **section 4.2.1. of Chapter 01**, there are three ways to wrap or enclose string objects:

```
a. ""
b. ''
c. """"""""
```

Strings can encompass all data types, meaning that you can convert any object into a string. To do this, you can use the **str()** function. See code 2.2.1 for an example.

It is important to note that whenever you prompt the user for input in Python, **the value will be stored as a string**. Therefore, you need to be careful and remember **to convert the value to the desired data type if necessary**.

Observe the difference in code samples 2.2.2 and 2.2.3. ***Note:*** *Pay attention to the error message in this example. This is one of the most common errors when coding.*

In [9]:
```
# Code 2.2.1

# wrapping int numbers into string
```

```
this_is_a_number = 2
this_is_a_string = "2"

# Checking the type of the variables
print(f"""
{"-"*40}
- The first variables is a: {type(this_is_a_number)}
- The second variable is a: {type(this_is_a_string)}
""")
```

```
----------------------------------------
- The first variables is a: <class 'int'>
- The second variable is a: <class 'str'>
```

In [10]:

```
# Code 2.2.2.

# importing os path and sys functions
import sys
import os

# adding os path to import function
sys.path.append(os.path.abspath("./__resources"))

# import think function
from thinking import think

# Saving input user into variable "number"
number = input(prompt = """Let's play a game! \
I bet you, it doesn't matter which number you    \
choose, after step 4, you wil get the number 5.
Select a whole number: """)

# Calling think function
think()
print("\n------- The code below will throw an error -------")


# Adding the next number to number
number_2 = number + (number + 1)

# Dividing number_2 by 9
number_3 = number_2 / 9

# Dividing number_3 by2
number_4 = number_3 / 2

# Substracting number to number_4
number_5 = number_4 - number

print(f"""
I know the result number is five.
But don't take it for granted, let's check!
Is 5 = number_5? {5 == number_5} """)
```

```
Let's play a game! I bet you, doesn't matter which number you    choose, after
step 4, you wil get the number 5.
Select a whole number: 2
Thinking...Thinking...Thinking...
------- The code below will throw an error -------
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Input In [10], in <cell line: 25>()
     21 print("\n------- The code below will throw an error -------")
     24 # Adding the next number to number
---> 25 number_2 = number + (number + 1)
     27 # Dividing number_2 by 9
     28 number_3 = number_2 / 9

TypeError: can only concatenate str (not "int") to str
```

In [15]:

```
# Code 2.2.4.

# importing os path and sys functions
import sys
import os

# adding os path to import function
sys.path.append(os.path.abspath("./__resources"))

# import think function
from thinking import think

# Saving input user into variable "number"
number = input(prompt = """Let's play a game! \
I bet you, it doesn't matter which number you    \
choose, after step 4, you wil get the number 5.
Select a whole number: """)

number = int(number)

# Step 1: Create number_2 variable
# Adding the subsequent number
number_2 = number + (number + 1)

# Step 2: Adding 9 to number_2
number_3 = number_2 + 9

# Step 3: Dividing number_3 by 2
number_4 = number_3 / 2

# Step 4: Substracting number to number_4
number_5 = number_4 - number



print(f"""

- Step 1: Add the subsequent number      | {number} + {number + 1}
- Step 2: Add 9                          | {number_2 + 9}
- Step 3: Divide the result by 2         | {number_3 / 2}
- Step 4: substract the original number | {number_4} - {number}
""")

# Calling think function
think()

print(f"""

I know the result is number five.
But don't take it for granted, let's check!
Is 5 = number_5? {5 == number_5} """)
```

```
Let's play a game! I bet you, it doesn't matter which number you    choose, af
ter step 4, you wil get the number 5.
Select a whole number: 9


- Step 1: Add the subsequent number      | 9 + 10
- Step 2: Add 9                          | 28
- Step 3: Divide the result by 2         | 14.0
- Step 4: substract the original number  | 14.0 - 9

Thinking...Thinking...Thinking...

I know the result is number five.
But don't take it for granted, let's check!
Is 5 = number_5? True
```

3. Logically is it True or False?

3.1. To be or not to be...?

Logic is defined as the study of correct reasoning. Most of the time, when someone mentions the term logic, you might you might fly back to the ancient Greece and picture Aristotle delivering a philosophical speech on an amphitheater. This is not far from the truth, as one of the first thinkers of logic and one of its major contributors was, indeed, Aristotle. Furthermore, in his Organon collection, he introduced the term logic for the first time.

You might be thinking, but what does this have to do with Python? Well, formal logic plays a central role in both philosophy and mathematics, and thus, in programming as well. **Booleans** are a data type that represents the two fundamental logical values: True and False. Combining these values allows you to create multiple scenarios and trigger different code or programs. Booleans are also the foundation of conditional statements, which were at the beginning of artificial intelligence (along with probability and statistics). ***Note:*** *Conditional statements will be cover in **Chapter 04**.*

3.2. The Truth Terms

The boolean values **True** and **False** can be used directly in your code. However, in most cases, this will not be sufficient. You will need to create and combine logical statements that represent your problem and the desired outcome in order to trigger an action. Therefore, it is essential to understand (and ideally memorize) the possible combination you can use in order to effectively use boolean logic in your code.

Refer to Table 3.2 for a list of **"truth terms"** and how they can be used to produce a boolean result (True or False). Practice using these truth terms in code samples 3.2.1 and 3.2.2.

Term	|	Description	|	Example	|	Result
and	|	Intersection of arguments. Returns True if both statements are true	|	2 == 2 **and** 1 == 1	|	True
or	|	Disjunction of arguments. Returns True if one of the statements is true	|	2 == 2 **or** 1 == 0	|	True
not	|	Reverse the result, returns False if the result is true	|	**not** 1 == 1	|	False
!=	|	Returns True when two arguments are different.	|	1 **!=** 0	|	True
==	|	Returns True when two arguments are equal	|	1 **==** 1	|	True
>=	|	Returns True when the left argument is greater or equal than the right argument	|	5 **>=** 2	|	True
<=	|	Returns True when the right argument is greater or equal than the left argument	|	2 **<=** 5	|	True
True	|	Boolean value of truth	|	-	|	True
False	|	Boolean value of untruth	|	-	|	False

Table 3.2: Truth Terms.

In [16]:
```
# Code 3.2.1.

# Let's do some exercises just with True and False

# define a as True and b as False
a = True
b = False

print(f"""{"*" * 40}
Considering:
a = {a}
b = {b}

Let's check logical combinations with booleans
- a and b = {a and b}
- a or b = {a or b}
- a and (a or b) = {a and (a or b)}
- a and not(a or b) = {a and not(a or b)}
- not a or b = {not a or b}
- a and not b = {a and not b}
{"*" * 40}
""")

# define c and d as numbers:
c = 100
d = 50

print(f"""{"*" * 40}
Cosidering:
c = {c}
d = {d}
Let's check logical combinations with numbers
- c != d = {c != d}
- (c - 50) == d = {c - 50 == d}
- c / d >= 10 = {c / d >= 10}
- c^2 - 2*c*d + d^2 == 2500 = {c**2 - 2*c*d + d**2 == 2500}
- int(False) = {int(False)}
- int(True) = {int(True)}
{"*" * 40}
""")
```

```
****************************************
Considering:
a = True
b = False

Let's check logical combinations with booleans
- a and b = False
- a or b = True
- a and (a or b) = True
- a and not(a or b) = False
- not a or b = False
- a and not b = True
****************************************

****************************************
Cosidering:
c = 100
d = 50

Let's check logical combinations with numbers
- c != d = True
- (c - 50) == d = True
- c / d >= 10 = False
- c^2 - 2*c*d + d^2 == 2500 = True
- int(False) = 0
- int(True) = 1
****************************************
```

In [17]:

```
# Code 3.2.2.
# Let's build some logic statements
# The following statements are true
# p: "Aldo is Italian"
p = True

# q: "Bob is English"
q = True

# Provide the boolean value
# of the next statements

# 1. Aldo isn't Italian
print(f"""{"-" * 40}
1. Aldo isn't Italian: not p = {not p}
{"-" * 40}
""")

# 2. "Aldo is Italian while Bob is English"
print(f"""{"-" * 40}
2. Aldo is Italian while Bob is English:
```

```
p and q = {p and q}
{"-" * 40}
""")

# 3. “Either Aldo is Italian and Bob is English,
#    or neither Aldo is Italian nor Bob is English”
print(f"""{"-" * 40}
3. Either Aldo is Italian and Bob is English \
or neither Aldo is Italian nor Bob is English:
(p and q) or (not p and not q) = \
{(p and q) or (not p and not q)}
{"-" * 40}
""")
```

```
----------------------------------------
1. Aldo isn't Italian: not p = False
----------------------------------------

----------------------------------------
2. Aldo is Italian while Bob is English:
p and q = True
----------------------------------------

----------------------------------------
3. Either Aldo is Italian and Bob is English or neither Aldo is Italian nor Bo
b is English:
(p and q) or (not p and not q) = True
----------------------------------------
```

In [85]:
```
bool("")
```

Out[85]:
```
False
```

4. Binaries

Binary sequence types refer to **immutable** sequences of single bytes. While you will not cover or work with binary data types in this course, it is important to note that binary data types are used for manipulating binary data. There are two types of binary data types:

```
i. bytes
ii. bytearray
```

You can declare a byte data type in a similar way to strings ('' or "") but you must add the prefix **b** at the beginning of the declaration. You can also declare a bytearray by using the **bytearray( )** function.

See the example in code sample 4.1 for more details.

In [18]:
```
# Code 4.1.

# Declare my_byte_1 variable. Use b prefix and ''
```

```
my_byte = b'char_data'

# Declare my_bytearray variable
my_bytearray = bytearray(5)

# print my_byte andmy_bytearray
print(f"""my_byte = {my_byte}""")  # b'char_data'
print(f"""my_bytearray = {my_bytearray}""")  # bytearray(b'\x00\x00\x00\x00\x0
```

```
my_byte = b'char_data'
my_bytearray = bytearray(b'\x00\x00\x00\x00\x00')
```

5. Sets

Sets are a data type in Python that allows you to store various **unordered** items. Sets **do not allow for duplicate elements**, so they will only store the unique values of the items declared. Since sets are unordered collections, they do not record the position or index of elements. As a result, you cannot access elements using indexing, slicing, or other sequence-like behaviors. You can only **loop** through the elements of the set *(e.g., for x in set)*.

There are currently two built-in set types: **set( )** and **frozenset**. The main difference between these types is that **sets** are **mutable** (you can add or remove items), whereas **frozensets** are **immutable** (they cannot be altered after they are created).

```
Format to create a set types:
my_set = {item_1, item_2, ..., item_n}
my_frozenset = frozenset({item_1, item_2, ..., item_n})
```

Common uses of sets includes removing duplicates from sequences, computing mathematical operations like intersections, unions, difference, or symmetric differences, among others.

In [89]:
```
my_set = set({"1", 2, 3 ,4})
my_set
```

Out[89]:
```
{'1', 2, 3, 4}
```

In [19]:
```
# Code 5.1.

# Declare a set
my_set = {1, 2, 3, 4}
print(f"""my_set = {my_set}""")

# Add new value to a set
my_set.add(5)
print(f"""my_set = {my_set}""")

# Declare a set with duplicates
my_set_w_duplicates = {1, 2, 3, 3, 3, 4, 5, 3}
```

```
# print duplicate set. Does not save duplicates
print(f"""my_set_w_duplicates = {my_set}""")

# Transform a set into a frozenset
my_frozenset = frozenset(my_set)

# print frozenset
print(f"""my_frozenset = {my_frozenset}""")

# Try to add a new element to a frozenset
print("\n------- The code below will throw an error -------")
my_frozenset.add(7)
```

```
my_set = {1, 2, 3, 4}
my_set = {1, 2, 3, 4, 5}
my_set_w_duplicates = {1, 2, 3, 4, 5}
my_frozenset = frozenset({1, 2, 3, 4, 5})

------- The code below will throw an error -------
```

```
---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)
Input In [19], in <cell line: 25>()
     23 # Try to add a new element to a frozenset
     24 print("\n------- The code below will throw an error -------")
---> 25 my_frozenset.add(7)

AttributeError: 'frozenset' object has no attribute 'add'
```

6. Sequence data types

6.1. List

Lists are a mutable and versatile data type in Python, similar to arrays in other programming languages like C/C++. However, one advantage of lists is that they allows to store and hold different types of data at the same time. You can declare lists by wrapping data within square brackets - **[ ]** - and separating elements by a comma **","**.

```
Format to create a list
my_list = [item_1, item_2,...., item_n]
```

Lists are one of the most useful data types in Python, with three main properties: **mutability, extensibility, and convertibility**. You will explore these properties and how to work with lists in more detail in sections 6.1.2 and 6.1.3.

Before dive into those details, let's first understand how to declare a list. </div>

6.1.1. Declaring List

As previously mentioned, you can declare a list by enclosing objects within square brackets - [ ] - and separating each element with a comma ",". If you look at code sample 6.1.1.1, you will see that the only difference between creating the number_variable and number_list variables is the use of **square brackets**. Furthermore, you can also see the difference in the output when you print both variables

This is a reminder from Python that you are working with lists.

Another way to check the type of a variable is to print its **type( )**. In code 6.1.1.2, you can see that the number_variable is of type int and number_list is of type list. Since these objects are of different classes and, therefore, different object types, they will not behave the same way.

Operating with different variable types

If you look at code 6.1.1.3, you will see that both variables contain the value 10. However, when you multiply both variables by 5, Python treats them differently and returns different outputs. The **number_variable** will perform the mathematical operation 10 * 5 = 50, while the **number_list** will insert the item "10" 5 times in the list, creating a new list with five "10" elements

Note: *Many code errors are related to operations with different types of variables. It is important to always check that you are working with the correct types of variables. See code sample 6.1.1.4 for an example*

In [20]:
```
# Code 6.1.1.1.

# Declaring a number variable
number_variable = 10

# Declaring a list of one number
number_list = [10]

# print types for number and number_list
print(f"""number_variable = {number_variable}""")
print(f"""number_list = {number_list}""")
```

```
number_variable = 10
number_list = [10]
```

In [21]:
```
# Code 6.1.1.2.

# print types for number and number_list
```

```
print(f"""number is a {type(number_variable)}""")
print(f"""number_list is a {type(number_list)}""")
```

```
number is a <class 'int'>
number_list is a <class 'list'>
```

In [22]:
```
# Code 6.1.1.3.

# number_variable times 5
print(f"""number_variable * 5
{number_variable*5}""")

# number_list times 5
print(f"""number_list * 5
{number_list*5}""")
```

```
number_variable * 5
50
number_list * 5
[10, 10, 10, 10, 10]
```

In [23]:
```
# Code 6.1.1.4.

# sum number_variables and number_list
print("\n------- The code below will throw an error -------")
print(f"""number_variables + number_list
{number_variable + number_list}""")
```

```
------- The code below will throw an error -------
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Input In [23], in <cell line: 5>()
      1 # Code 6.1.1.4.
      2 
      3 # sum number_variables and number_list
      4 print("\n------- The code below will throw an error -------")
      5 print(f"""number_variables + number_list
----> 6 {number_variable + number_list}""")

TypeError: unsupported operand type(s) for +: 'int' and 'list'
```

6.1.2. Accessing lists elements

After declaring (i.e., creating) a list, you can access the items within it by referencing the **index** of each element. Each element is stored at a specific position within the list, with the index of **the first element being 0**. You can access a specific element of a list by specifying its index within **square brackets ([])**. You can also define a range of elements within square brackets ([]) to access multiple elements at once (a technique known as "slicing").

```
Format to access list elements.
my_list[0]   -> returns first element (index forward).
my_list[0:2] -> returns element in position 0 and 1. Does not
include upper limit (slicing).
my_list[-1]  -> return the last element of the list (index
backwards).
```

Let's practice accessing lists with codes 6.1.2.1 to 6.1.2.5.

Note: *The index of the last item of a list will always be* ***n-1*** *where n represents the length of the list.*

In [24]:
```
# Code 6.1.2.1.

# Declaring a list
my_list = [1, 2, 3, "Hello", "World!"]

# Lenght of the list
print(f"""The length of my_list is:
{len(my_list)}""")
```

```
The length of my_list is:
5
```

In [25]:
```
# Code 6.1.2.2.

# Accessing first element
print(f"""The first element of my_list is:
{my_list[0]}""")
```

```
The first element of my_list is:
1
```

In [27]:
```
# Code 6.1.2.3.

# Accessing last element
print(f"""The last element of my_list is:
{my_list[-1]}""")
```

```
The last element of my_list is:
World!
```

In [ ]:
```
# Code 6.1.2.4

# Slicing elements 2 & 3.
# When slicing Python will return a list.
print(f"""The second and third elements are:
{my_list[1:3]}""")
```

In [28]:
```
# Code 6.1.2.5.

# Use element 4 and 5 to print: "Hello, World!"
print(my_list[3], my_list[4], sep = ", ")
```

```
Hello, World!
```

6.1.3. Advantages of lists.

As mentioned previously, lists are: ~~~ a. Mutable b. Extensible c. Convertible ~~~ Let's dive deep into each characteristic and see examples of them.

a. Lists are mutable.

Lists are mutable But...what does it means? Well, it means you can change the values of a previously declared list elements. For example, if you defined one of the items in a list as "cat," you can later change it to "dog". This flexibility makes lists a powerful tool in Python programming.

Note: *You can update values based on conditions, iterations, function, among others.*

In [29]:
```
# Code 6.1.3.1.

# Declare list of cars
my_cars = ["BMW", "Honda","Mazda", "Mini", "Porsche"]

# print my_cars
print(f"""My favorites car brands are:
{my_cars}""")

# You changed your mind! You want to remove Mini and add Ferrari!
# Update my_cars list. Mini is on index 3, so we can replace it's value
# by accessing the third element and assigning a new value.
my_cars[3] = "Ferrari"

print(f"""My new favorites car brans are:
{my_cars}""")
```

```
My favorites car brands are:
['BMW', 'Honda', 'Mazda', 'Mini', 'Porsche']
My new favorites car brans are:
['BMW', 'Honda', 'Mazda', 'Ferrari', 'Porsche']
```

b. Lists are extensible.

Suppose you just tried out a new car, a Tesla, and now you want to add it to your list of favorite cars. Unlike the example in code 6.3.1.1, you don't want to replace one value with another, you want to **add a complete new value to the list**. Well, there are three ways you can do this:

```
i. append()
ii. extend()
iii. insert()
```

Let's dive into each function!

i. append()

The **append()** methods is one of the most common ways to add a new element to a list. This method allows you to append a single variable or object to the end of the list. The variable can be of any data type (string, integer, etc). Moreover, you could create a more complex structure by appending another list or tuple. Code 6.1.3.2 shows an example of using the append() method to add elements to the my_car list.

Note: *The append() method stores anything you wrap within it as a* ***single object****. So, if you append a new list, Python will add it as one item at the end of the original list. You can see this in the example in code 6.1.3.3.*

ii. extend()

What happens when you want to append each particular element of a list as a new item into the original list?

The **extend()** method allows you to add the elements of one list to another list. It iterates over the elements of the second list and appends each element to the end of the original list. This is different from the append() method, which adds the entire second list as a single object to the original list. You can see the difference between these two methods in code examples 6.1.3.3 and 6.1.3.4.

iii. insert()

Let's say that the order of a list is crucial for you. For example, you want to add a new car brand to your list of cars and keep the alphabetical order. The append() and extend() won't achieve this as they add the new values at the bottom of the original list.

The **insert** method allows you to specify the index at which you want to add a new element to a list. It takes two arguments: the index and the element to be inserted.

If you want to add a new brand to your alphabetically ordered list of cars, you can use the insert() method to specify the correct position for the new element.

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

Let's work with the insert method on code 6.1.3.5. and 6.1.3.6.

Note: *Take into consideration that* ***append, insert,*** *and* ***extend*** *will change the length of the list.*
Additionally, the ***insert*** *method will shift the index of all the items at the right of the one inserted.*

In [30]:
```
# Code 6.1.3.2.

# print list before adding a new variable
print(f"""My favorites car brands are:
{my_cars}""")

# appending "Maserati" to my_cars list
my_cars.append("Maserati")

# print list after adding a new variable
print(f"""My new favorites car brands are:
{my_cars}""")
```

```
My favorites car brands are:
['BMW', 'Honda', 'Mazda', 'Ferrari', 'Porsche']
My new favorites car brands are:
['BMW', 'Honda', 'Mazda', 'Ferrari', 'Porsche', 'Maserati']
```

In [31]:
```
# Code 6.1.3.3.

# Declare my_cars_2 list
my_cars_2 = ["Jeep", "Audi"]

# Let's append my_cars list to my_cars_2 list
my_cars_2.append(my_cars)

# print the result. Check how after the audi
# item, everything is wrapped by [].
print(f"""My new favorites car brands are:
{my_cars_2}""")
```

```
My new favorites car brands are:
['Jeep', 'Audi', ['BMW', 'Honda', 'Mazda', 'Ferrari', 'Porsche', 'Maserati']]
```

In [32]:
```
# Code 6.1.3.4.

# Declare my_cars_2 list
my_cars_2 = ["Jeep", "Audi"]

# Let's extend my_cars list to my_cars_2 list
my_cars_2.extend(my_cars)

# print the result. Check how after the audi
# item, there is no [].
print(f"""My new favorites car brands are:
{my_cars_2}""")
```

```
My new favorites car brands are:
['Jeep', 'Audi', 'BMW', 'Honda', 'Mazda', 'Ferrari', 'Porsche', 'Maserati']
```

In [33]:
```
# Code 6.1.3.5.

# Add "Ferrari" after "BMW"
my_cars.insert(1, "Ferrari")
```

```
# print the result
print(f"""My new favorites car brands are:
{my_cars}""")
```

```
My new favorites car brands are:
['BMW', 'Ferrari', 'Honda', 'Mazda', 'Ferrari', 'Porsche', 'Maserati']
```

In [34]:
```
# Code 6.1.3.6.

# Add Tesla after Porsche. To get the last index
# use the length function.
my_cars.insert(len(my_cars), "Tesla")

# print the result
print(f"""My new favorites car brands are:
{my_cars}""")
```

```
My new favorites car brands are:
['BMW', 'Ferrari', 'Honda', 'Mazda', 'Ferrari', 'Porsche', 'Maserati', 'Tesl
a']
```

c. Lists are convertible.

In Python, it is possible to convert a list into another data type or to convert another data type into a list. The convertibility of lists allows for greater flexibility and versatility when working with data in Python. It allows you to easily manipulate and transform data to fit the needs of your specific program or task. For example, lists can be converted very easy into DataFrames, which are tabular structures similar to Spreadsheets.

Note: *DataFrames will be introduced in Chapter 06 and 07. You will be working extensively with DataFrames as Data Analyst.*

In [5]:
```
# Code 6.1.3.7.

# import pandas
import pandas as pd

# Declare 3 list variables with the brand and price of a car
my_car_1 = ["Tesla - Model S", 105000]
my_car_2 = ["Ferrrari - Roma", 250000]
my_car_3 = ["Porsche - Boxster 718", 65000]

# Declar a list with all cars
my_cars_list = [my_car_1, my_car_2, my_car_3]

# Creating a pandas dataframe (df)
df_cars = pd.DataFrame(my_cars_list, columns = ["Brand - Model", "Price"])

# Output df_cars
df_cars
```

Out[5]:

	Brand - Model	Price
0	Tesla - Model S	105000
1	Ferrrari - Roma	250000
2	Porsche - Boxster 718	65000

6.1.4. Remove list elements

So far, you have learn how to access and insert new elements to a list. Let's see how to remove elements of a list. There are two methods to remove elements:

```
i. list.remove(element) -> Remove the first matching element.
ii. list.pop(index) -> Remove the element at index.
```

i. remove()

The **remove()** is used to remove a specific object from a list. It takes an object as an argument and deletes the first ocurrence of that object in the list. If the object appears multiple times in the list, only the first instance will be removed. Finally, this method does not return any value. An example of using the remove() method can be seen in code 6.1.4.1.

ii. pop()

The **pop()** method removes an item from a list by specifying its index as an argument. This method not only removes the item from the list, but also returns the value of the removed object. An example of using the pop() method can be seen in code 6.1.4.2.

In [36]:

```
# Code 6.1.4.1.

# Declare list of cars
my_cars = ["BMW", "Honda","Mazda", "Mini", "Porsche"]

# remove "BMW"
my_cars.remove("BMW")

# print outcome
print(f"""My new favorites car brands are:
{my_cars}""")
```

```
My new favorites car brands are:
['Honda', 'Mazda', 'Mini', 'Porsche']
```

In [37]:

```
# Code 6.1.4.2.

# Let's remove "Mazda" with the pop() method
# You can store the value removed
car_removed = my_cars.pop(1)
```

```
# print new list and car removed
print(f"""My new favorites car brands are:
{my_cars}
I removed: {car_removed}""")
```

```
My new favorites car brands are:
['Honda', 'Mini', 'Porsche']
I removed: Mazda
```

6.1.5. Reversing a list

Let's review the last part of lists. Thus far, you have learned how to **append**, **insert**, and **extend** new elements to a list. However, append and insert only permit the addition of a single element at a time, while extend allows for the inclusion of multiple elements but always at the end of the list. How could you add multiple new objects at the beginning of a list? a workaround is necessary.This may seem unorthodox at first, but it serves as a prime example of how Python's built-in functions can be utilized to create more complex and sophisticated functions. Keep in mind that Python's built-in functions strive to be as simple as possible, so you can trust that they will perform as intended. To proceed, let us divide the task into three steps:

```
i. reverse the original list.
ii. extend the new elements.
iii. reverse the list back to the original state.
```

The **reverse()** function enables you to reverse the order of items within a list. This built-in function perform the operation **in place**, so it is not necessary to assign the reversed list to a new variable. It is important to keep track of when a list has been reversed and when it has been restored to its original order. Remember that Python executes code sequentially, meaning it will run from the first line of code and continue until the end.

As a best practice, it is recommended to save the reversed list to a separate variable, as demonstrated in code 6.1.5.1. and 6.1.5.2. However, once you become proficient with the logic, you can skip this step, as shown in code 6.1.5.3. Can you identify the cause of the different outputs in code 6.1.5.1. and 6.1.5.2.?

In [38]:
```
# Code 6.1.5.1.

# Step 1: Declare lists
alphabet_list = ["e", "f", "g"]
first_letters = ["a", "b", "c", "d"]

# Step 2: Copy original list
rev_alphabet_list = alphabet_list.copy()

# Step 3: Reversing rev_alphabet_list
```

```
rev_alphabet_list.reverse()

# Step 4: Extend new values
rev_alphabet_list.extend(first_letters)

# Step 5: Reverse rev_alphabet_list again
rev_alphabet_list.reverse()

# print alphabet_list and rev_alphabet_list
print(f"""
original list: {alphabet_list}
reversed list: {rev_alphabet_list}""")
```

```
original list: ['e', 'f', 'g']
reversed list: ['d', 'c', 'b', 'a', 'e', 'f', 'g']
```

In [39]:

```
# Code 6.1.5.2.
# Copy original list
rev_alphabet_list = alphabet_list.copy()

# Reverse rev_alphabet_list
rev_alphabet_list.reverse()

# Reverse first_letters
# Extend function iterates and add the values
# from the first item to the last one.
# Therefore, we need to reverse the first_letter too.
first_letters.reverse()

# Extend new values
rev_alphabet_list.extend(first_letters)

# Reverse rev_alphabet_list
rev_alphabet_list.reverse()

#print alphabet_list and rev_alphabet_list
print(f"""
original list: {alphabet_list}
reversed list: {rev_alphabet_list}""")
```

```
original list: ['e', 'f', 'g']
reversed list: ['a', 'b', 'c', 'd', 'e', 'f', 'g']
```

In [40]:

```
# Code 6.1.5.3.

# Step 1: Declare lists
alphabet_list = ["e", "f", "g"]
first_letters = ["a", "b", "c", "d"]

# print statement before reverse the original list
print(f"""
Before reverse: {alphabet_list}""")

# Reverse alphabet_list
alphabet_list.reverse()

# reverse first_letters
first_letters.reverse()

# Extend new values
```

```
alphabet_list.extend(first_letters)

# Reverse list again
alphabet_list.reverse()

# print alphabet_list and rev_alphabet_list
print(f"""
After reverse: {alphabet_list}""")
```

```
Before reverse: ['e', 'f', 'g']

After reverse: ['a', 'b', 'c', 'd', 'e', 'f', 'g']
```

Careful when copying variables

Before conclude with lists... Do you know why the **copy( )** was used in code 6.1.5.1 and 6.1.5.2? You may think that the following expression would produce the same result:

```
my_list_reversed = my_list
```

Well, when you use the "=" operator to assign a new list, you are telling Python to create a new variable that points to the same memory address as the original list. In other words, you have created two variable names for the same object. As a result, any changes made to either variable will affect both.

Be cautious when assigning your variables. Unintended issues may arise in the future as a result.
Let's examine code 6.1.5.4. and observe how **both variables refer to the same memory address, or 'id'!**

In [8]:
```
# Code 6.1.5.4.

# Declare my_list
my_list = ["b", "c", "d"]

# Create my_list_reversed
my_list_reversed = my_list

# print id for both variables and check if they are the same
print(f"""
Is my_list id: {id(my_list)} = my_list_reversed id: {id(my_list_reversed)}?
{id(my_list) == id(my_list_reversed)}""")

# Let's reverse my_list_reversed and add the letter "a"
# Reverse my_list_reversed
my_list_reversed.reverse()

# Add letter "a"
my_list_reversed.append("a")

print(f"""
{"-" * 50}
my_list: {my_list}
my_list_reversed: {my_list_reversed}
Both lists are reversed and have the letter "a"
{"-" * 50}""")
```

```
Is my_list id: 140566640104192 = my_list_reversed id: 140566640104192?
True

--------------------------------------------------
my_list: ['d', 'c', 'b', 'a']
my_list_reversed: ['d', 'c', 'b', 'a']
Both lists are reversed and have the letter "a"
--------------------------------------------------
```

6.1.6. List of lists

List can contain any type of data, including **other lists**. When a list contains other lists, it is called a **list of lists**. Refer to code 6.1.6.1. for an example on how to create a list of lists.

~~~ list_of_lists = [[l1_item0, ..., l1_itemi] , [l2_item0, ..., l2_itemj], ..., [ln_item0, ..., ln_itemk]] ~~~
To access the items in a list of lists, you can use the indexing notation. For example, to access the 'i' item in the first list, you would use list_of_lists[0][i]. Refer to code 6.1.6.2, for an example on how to access list of lists.

***Note:** You can also use loops to iterate over the items in a list of lists. Loops wll be covered in Chapter 04.*

In [130…
```
# Code 6.1.6.1.
# Declare a list of lists
list_of_lists = [[1, 2, 3],
                 ['a', 'b', 'c'],
                 ['apple', 'banana', 'orange']]
print(list_of_lists)
```

```
[[1, 2, 3], ['a', 'b', 'c'], ['apple', 'banana', 'orange']]
```

In [142…
```
# Code 6.1.6.2.

# Access second list of list_of_lists
print(f"""The second list contains:
{list_of_lists[1]}\n""")

# Access third element of third list of list_of_lists
print(f"""The third element of the third list contains:
{list_of_lists[2][2]}""")
```

```
The second list contains:
['a', 'b', 'c']

The third element of the third list contains:
orange
```

6.2. Tuples
~~~

Tuples are part of the **sequence data types** but they have some differences compared to lists. The most significant distinction is that tuples are **immutable**, meaning their original values cannot be replaced with new ones. You can declare tuples by wrapping data within parenthesis - **( )** - and separating elements by a comma **","**.

```
Format to create a list
my_tuple = (item_1, item_2,...., item_n)
```

Similar to lists, tuples can store various data types and can be accessed by their index. Tuples are often referred to as lists' cousins due to the numerous similarities they share.

Referring back to code 6.1.3.1, you attempted to change the item "Mini," which was located at the third index (fourth position), to "Ferrari." In lists, it is possible to alter the value of an item by assigning a new value. However, if you attempt to do the same with a tuple data type, you will receive an error from Python reminding you of the tuple's immutability. Let's examine code 6.2.1. ***Note:*** *The format for accessing tuples is the same as accessing lists.*

```
my_tuple[0] -> returns first element (index forward).
my_tuple[0:2] -> returns element in position 0 and 1. Does not
include upper limit (slicing).
my_tuple[-1] -> return the last element of the list (index
backwards).
```

</div>

In [9]:
```
# Code 6.2.1.

# Declare a tuple
my_tuple = (1,2,3,4)

# Access elements within my_tuple
# First item
print(my_tuple[0])
# Second and third item
print(my_tuple[1:3])
# Last item
print(my_tuple[-1])

# Let's try to replace a value from my_tuple
print("\n------- The code below will throw an error -------")
my_tuple[1] = 0
```

```
1
(2, 3)
4

------- The code below will throw an error -------
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Input In [9], in <cell line: 16>()
     14 # Let's try to replace a value from my_tuple
     15 print("\n------- The code below will throw an error -------")
---> 16 my_tuple[1] = 0

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

6.3. Range

Suppose you wish to create a list of numbers from 1 to 1000. How would you go about it? Currently, you would need to create a list with 1000 elements and manually type in all the numbers! Can you imagine having to do this multiple times? It would be an arduous task. But... Fortunately, Python has thought ahead and introduced the **range' data type**. The **range** data type is an **immutable** sequence of numbers that allows you to create a list of numbers within a specified range. Ranges are inclusive of the lower limit but not the upper limit.

You can declare a range variable by calling the **range( )** built in function and providing the lower and upper limits. The optional step argument enables you to define the unit of increment between the range limits
Let's examine code 6.3.1.

```
Format to create a range
my_range = range(lower_limit, upper_limit, [step])
```

Accessing ranges

Elements within a range can be accessed in a similar manner to elements within lists or tuples. However, if you slice a range Python will return a new range data type with the lower and upper limits defined by the slice. It is also possible to specify a different "step" when slicing a range.
Look at code 6.3.2. for an example.

```
Slicing range and change step
my_range[lower_limit:upper_limit:new_step]
```

Note: *Ranges are commonly used for* ***looping*** *a specific number of times in* ***for loops.*** *Loops will be covered in* ***Chapter - 05.***

In [43]:
```
# Code 6.3.1.

# Declare my_range variable
my_range = range(0,1000)

print(f"""
The data type of my_ragen is:
{type(my_range)}""")

# Accesing a range element
# Index forward
print(f"""
The first element in my_range is:
{my_range[0]}""")

# Index backward
print(f"""
The last element in my_range is:
{my_range[-1]}""")

# Create a range going from 0 to 50,
# but every 5 steps
my_range_2 = range(0,50, 5)
print(f"""
The second element in my_range_2 is:
{my_range_2[1]}""")
```

```
The data type of my_ragen is:
<class 'range'>

The first element in my_range is:
0

The last element in my_range is:
999

The second element in my_range_2 is:
5
```

In [44]:
```
# Code 6.3.2

# Slice my range from 10 to 20 every 5 steps
my_new_range = my_range[10:21:5]

# print second and third element
print(f"""
The second & third elements of my_new_range are:
{my_new_range[1]}, {my_new_range[2]}""")
```

```
The second & third elements of my_new_range are:
15, 20
```

6.4. Bonus: Numpy Arrays

Numpy, short for Numerical Python, is a fundamental Python library for scientific programming, particularly for numbers. It consists of multidimensional objects called **numpy arrays**. An array is similar to a list, as it is a grid of values that includes information about the data, its location, and how to interpret it. The key difference is that elements in a numpy array must be of the same data type. As a result, numpy will automatically transform all data types within an array to the most global data type. For instance, if one element is a string, all the other elements will be transformed into strings as well.

Arrays can have N-dimensions, hence they are often referred to as **ndarrays** (n-dimensional arrays). A 1-D array represents a **vector**, **column**, or **row**. A 2-D array represents a **matrix**. 3-D or higher dimensional arrays are known as **tensors**. See an example in code 6.4.1.

```
Format to create a numpy array
my_array = np.array([elements])
```

Numpy enables you to perform various linear algebraic operations, such as vector product (cross product), matrix-matrix multiplication, determinants, and matrix inversion, to name a few. While you may not extensively work with numpy arrays at this stage, it is crucial to comprehend how they function as libraries like pandas, which you will frequently use in the future, were built on top of numpy.

Let's examine code 6.4.2 to see some examples of these operations. ***Note:*** *Given that pandas is built on top of numpy, it is possible to perform some linear algebraic operations, like cross product, with DataFrames.*

In [44]:
```
# 6.4.1.
# import numpy library
import numpy as np

# Declare a 2-D numpy array.
my_array = np.array([[2, 2], [3,3]])
print(f"""
my_array
--------
{my_array}
""")

# Create an array with different data types.
# All the values will be converted to a string ('').
my_array2 = np.array([[2,2], [3, "three"]])
print(f"""
my_array2
--------
{my_array2}
""")
```

```
my_array
--------
[[2 2]
 [3 3]]


my_array2
--------
[['2' '2']
 ['3' 'three']]
```

In [43]:
```
# Code 6.4.2.
# Let's resolve a mathematical linear equation with numpy
# 1. x + 2y = 5
# 2. y - 3z = 5
# 3. 3x - z = 4

# Declare matrix for x, y, z
A = np.array([[1, 2, 0],
              [0, 1, -3],
              [3, 0, -1]])

# Declare vector results
b = np.array([5, 5, 4])

# solve the formula x = A^-1 * b
x = np.linalg.inv(A).dot(b)
print(f"""
x = {int(x[0])}
y = {int(x[1])}
z = {int(x[2])}
""")
```

```
x = 1
y = 2
z = -1
```

7. Dictionaries...what?

Is most likely that you have used a dictionary before in your life. A dictionary is defined as a book or resource that **list the words of a language and gives their meanings.** Let's consider the structure of a dictionary. Typically, a book dictionary is organized in two columns: one column contains the "word" value, and the second column holds the "meaning" value. Each "row/entry" has information about a different word. So, you could think of a dictionary as two lists, one list containing all the words and another list containing all the meanings. However, these lists may be sorted differently, so you must correctly associate each word with its corresponding meaning.

Well, dictionaries in Python operate in a similar manner. They allow you to link two variables and maintain that connection. To do that, Python requires you to provide a **key**

(word) and a value (meaning).

Python defines dictionaries as a mapping object that maps **key values** to **arbitrary objects.** You can use any value as a key but mutable types.

7.1. How to declare a dictionary?

There are three ways to declare a dictionary, but in all of them you must provide a pair of key:value items.

```
Declaring a dictionary:
i. my_dict = {key_1:value_1, key_2:value_2,...,key_n:value_n}
ii. my_dict = dict([(key_1, value_1), (key_2, value_2),...,
(key_n, value_n)])
iii. my_dict = dict(key_1=value_1,
key_2=value_2,...,key_n=value_n)
```

Dictionaries are **ordered** (since Python 3.7), **mutable**, and **do not allow duplicates**.

Let's take a look at code 7.1.1. and create your first **dictionary.**

In [45]:
```
# Code 7.1.1.

# Declare dictionary as i.
my_dict_1 = {"brand":"Mazda", \
             "model":"Mazda6"}

# Declare dictionary as ii.
my_dict_2 = dict([("gingerly", "adverb. with great care or caution"), \
             ("areology", "noun. the observation and study of the planet Mars

# Declare dictionary as iii.
# With this option you don't need to use "" for the key, Python will understa
# it as a string not a variable
my_dict_3 = dict(name="Emma", \
                 age = 26)

print(f"""
my_dict_1 = {my_dict_1}
my_dict_2 = {my_dict_2}
my_dict_3 = {my_dict_3}
""")
```

```
my_dict_1 = {'brand': 'Mazda', 'model': 'Mazda6'}
my_dict_2 = {'gingerly': 'adverb. with great care or caution', 'areology': 'n
oun. the observation and study of the planet Mars.'}
my_dict_3 = {'name': 'Emma', 'age': 26}
```

7.2. How to work with dictionaries?

7.2.1. Accessing dictionaries

Unlike lists, you cannot access dictionaries by their element index/position. First, you must indicate to Python which **key** you want to access. Then, you can access the elements that belong to that key. **The deepest element you can access in a dictionary is the characters that make up the deepest element**.

Let's examine code 7.2.1.

```
Accessing a dictionary:
i. Access key elements: my_dict["key"]
ii. Access values: my_dict["key"][index]
```

7.2.2. Add and update elements from a dictionary

Adding new key:value pairs to a dictionary is very easy. You must indicate a key value that does not exist in the current dictionary and assign a value to it. For an example, see at code 7.2.2.1.

```
Add a new key:value pair:
my_dict["new_key"] = new_value
```

In the other hand if you want to update the values of an existing key, you can use the **update( )** function, by indicating the key and the new value of it.
For an example, see at code 7.2.2.2.

```
my_dict["existing_key"] = new_value
```

7.2.3. Dictionaries built-in functions

Let's talk about the most important built in functions for dictionaries: **keys**, **values**, **items**, **pop**, and **popitem**.

i. keys(), values(), and items

The **keys()** method returns a view object that displays a list of all the keys in the dictionary. This view object is dynamic and reflects any changes made to the dictionary.

On the other hand, the **values()** method returns a view object that displays a list of all the values in the dictionary. This view object is dynamic and reflects any changes made to the dictionary.

Finally, the **items()** method is used to retrieve the items (key-value pairs) of a dictionary.

The **keys( )**, **values( )**, and **items( )** method are useful for iterating over the keys and values of a dictionary and performing operations on them. They are often used together.

Refer to code 7.2.3.1. for an example on keys(), values(), and items() methods.

iii. pop() and popitem()

The **pop()** method is used to remove a key-value pair (item) from a dictionary. This method returns the item removed, hence you could store that item if required.

On the other hand, the **popitems()** method removes the item that was last inserted into the dictionary. Additionally, it returns the value of the item removed as a tuple.

Refer to code 7.2.3.2. and 7.2.3.3. for an example on pop() and popitems() methods respectively.

7.3. More complex dictionaries

Let's say you want to add more than one car brand and model into a dictionary. You could add manually each new key:value pair to dictionary. It would be an arduous task. Instead, you could leverage your knowledge of lists. Could you create a key:value situation based on two independent lists?

The **zip( )** built-in function iterates over a pair of sequences and matches the elements from each list based on their positions. It aggregates both iterables and returns a tuple data type. **You can use this tuple to create a dictionary.**

Let's take a look at code 7.3.1. to combine **lists and dictionaries.**

```
Use zip to transform two lists into a dict:
zip(key_list, values_list)
```

Note: *When mapping two independent list with the zip function,* ***the order of the elements matters!***

In [50]:
```
# code 7.2.1.

# Declare a dictionary
my_dict_4 = {"brand": "Mazda",
             "model": ["Mazda6", "Mazda3"]}

# Accesing model key
print(f"""
There are 2 models in my_dict_4:
{my_dict_4["model"]}""")

# Accessing the first model of the models list
print(f"""
There first model of my_dict_4 is
{my_dict_4["model"][0]}""")

# Accessing the first character of the first
# model (letter "M")
print(f"""
The letter of the first model in my_dict_4 is
{my_dict_4["model"][0][0]}""")
```

```
There are 2 models in my_dict_4:
['Mazda6', 'Mazda3']

There first model of my_dict_4 is
Mazda6

The letter of the first model in my_dict_4 is
M
```

In [54]:
```
# Code 7.2.2.1.

# Declare a dictionary
my_dict_5 = {"brand": "Mazda",
             "model": ["Mazda6", "Mazda3"]}
# add a new key:value pair
my_dict_5["color"] = ["red", "blue"]
print(my_dict_5)
```

```
{'brand': 'Mazda', 'model': ['Mazda6', 'Mazda3'], 'color': ['red', 'blue']}
```

In [55]:
```
# Code 7.2.2.2.
# Declare a dictionary
my_dict_6 = {"brand": "Mazda",
             "model": ["Mazda6", "Mazda3"]}
# update model value
my_dict_6.update({"model": ["Mazda6", "CX-5"]})
print(my_dict_6)
```

```
{'brand': 'Mazda', 'model': ['Mazda6', 'CX-5']}
```

In [61]:
```
# Code 7.2.3.1.
print(f"""
Keys of my_dict_6 variables
----------------------------
{my_dict_6.keys()}""")
print(f"""
Values of my_dict_6 variables
-----------------------------
{my_dict_6.values()}""")
print(f"""
Items of my_dict_6 variables
-----------------------------
{my_dict_6.items()}""")
```

```
Keys of my_dict_6 variables
----------------------------
dict_keys(['brand', 'model'])

Values of my_dict_6 variables
-----------------------------
dict_values(['Mazda', ['Mazda6', 'CX-5']])

Items of my_dict_6 variables
-----------------------------
dict_items([('brand', 'Mazda'), ('model', ['Mazda6', 'CX-5'])])
```

In [66]:
```
# Code 7.2.3.2.
my_dict_7 = {"brand": "Mazda",
             "model": ["Mazda6", "Mazda3"]}
# update model value
my_dict_7.pop("model")
print(f"""
New dictionary after applied pop
-------------------------------------
{my_dict_7}""")
```

```
New dictionary after applied pop
-------------------------------------
{'brand': 'Mazda'}
```

In [67]:
```
# Code 7.2.3.3.
my_dict_8 = {"brand": "Mazda",
             "model": ["Mazda6", "Mazda3"],
             "color": ["red", "blue"]}
# update model value
item_removed = my_dict_8.popitem()
print(f"""
New dictionary after applied popitem
-------------------------------------
{my_dict_8}""")
print(f"""
item removed
-------------
{item_removed}""")
```

```
New dictionary after applied popitem
-------------------------------------
{'brand': 'Mazda', 'model': ['Mazda6', 'Mazda3']}

item removed
-------------
('color', ['red', 'blue'])
```

In [47]:
```
# Code 7.3.1.

# Declare list of brands
car_brand = ["BMW", "Mazda", "Porsche"]

# Declare list of models
# Remember order is important!
car_model = [["5 Series", "3 Series"], \
             ["Mazda6", "Mazda3"], \
             ["718", "Panamera"]]
```

```
car_tuple_zip = zip(car_brand, car_model)

# print tuple_zip. We need to use the
# tuple() function to read a zip variable
print(f"""
{"-" * 110}
tuple_zip type  = {type(car_tuple_zip)}
tuple_zip value =
{tuple(car_tuple_zip)}
{"-" * 110}""")

# Transform tuple_zip into dictionary
cars_dict = dict(zip(car_brand, car_model))

# print cars_dict
print(f"""
{"-" * 110}
cars_dict = {cars_dict}
{"-" * 110}
""")
```

```
------------------------------------------------------------------------------
---------------------------------
tuple_zip type  = <class 'zip'>
tuple_zip value =
(('BMW', ['5 Series', '3 Series']), ('Mazda', ['Mazda6', 'Mazda3']), ('Porsch
e', ['718', 'Panamera']))
------------------------------------------------------------------------------
---------------------------------

------------------------------------------------------------------------------
---------------------------------
cars_dict = {'BMW': ['5 Series', '3 Series'], 'Mazda': ['Mazda6', 'Mazda3'],
'Porsche': ['718', 'Panamera']}
------------------------------------------------------------------------------
---------------------------------
```

7.4. Transforming dictionaries into DataFrames

You have already heard about pandas and DataFrames earlier in this chapter (***c. Lists are convertibles)***. Even though pandas will not be cover on this chapter, it is important for you to become familiar with this library.

As you did with lists, you can transform dictionaries into DataFrames.

Look at code 7.4.1. and follow the logic to convert a dictionary into a DataFrame.

In [48]:
```
# Code 7.4.1.

# Import pandas
import pandas as pd

# Create a DataFrame with the from_dict function
df_cars = pd.DataFrame.from_dict(cars_dict)
```

```
# Change dataframe index name
df_cars.index = ["model_1", "model_2"]

# Transpose rows and columns
df_cars.transpose()
```

Out[48]:

	model_1	model_2
BMW	5 Series	3 Series
Mazda	Mazda6	Mazda3
Porsche	718	Panamera

8.Working with everything!

Let's take a look into code 8.1. and create a new code with what you've learned so far.

In [49]:

```
# Code 8.1.

# Import date packages
from datetime import date
from datetime import datetime

# Introduce the user to the program
print(f"""{"-" * 110}
Welcome to Python! I'm thrilled to see you here!
""")

# Ask user's name
user_name = input(prompt = "What's your name?")

# Ask user's DOB
user_birth = input(prompt = f"""Hi {user_name}, it's amazing to meet you.
May I ask your birth date? [Please use this format: mm/dd/yyyy] """)

# Declare current_date variable with today's date
current_date = datetime.today()

# Transform user's DOB from string to date variable
user_birth = datetime.strptime(user_birth, '%m/%d/%Y')

# Calculate days difference between today and users date
days = (current_date - user_birth).days

# Declare variable with dates per year
days_p_year = 365.25

# Calculate user's age by dividing days by days p/year
user_age = int(days / days_p_year)

# Calculate the days remaining (module)
user_days = (days % days_p_year)
```

```
# Print user age in years and days.
print(f"""
Aha! You have lived:
                - {user_age} years and {user_days} days!
                    """)

# Declare a list with NBA teams
nba_teams = [["Phoenix Suns"],
             ["Golden State Warriors", "Los Angeles Clippers",
              "Los Angeles Lakers", "Sacramento Kings"],
             ["Denver Nuggets"],
             ["Miami Heat", "Orlando Magic"],
             ["Atlanta Hawks"],
             ["Chicago Bulls"],
             ["Indiana Pacers"],
             ["New Orleans Pelicans"],
             ["Boston Celtics"],
             ["Detroit Pistons"],
             ["Minnesota Timberwolves"],
             ["Brooklyn Nets", "New York Knicks"],
             ["Charlotte Hornets"],
             ["Cleveland Cavaliers"],
             ["Oklahoma City Thunder"],
             ["Portland Trail Blazers"],
             ["Philadelphia 76ers"],
             ["Memphis Grizzlies"],
             ["Dallas Mavericks", "Houston Rockets", "San Antonio Spurs"],
             ["Utah Jazz"],
             ["Washington Wizards"],
             ["Milwaukee Bucks"]]

# Declare a list with NBA teams states. Keep order with nba_teams list
state_list = ["AZ", "CA", "CO", "FL", "GA", "IL", "IN", "LA", "MA", "MI",
              "MN", "NY", "NC","OH", "OK", "OR", "PA", "TN", "TX", "UT",
              "WA", "WI"]

# Create a dictionary for nba teams. Match states and teams.
nba_dict = dict(zip(state_list, nba_teams))

# Ask user's state
user_state = input(prompt= """I would like to know more about you.
Which state are you currently living?
[please use the two letter abreviation]""")

# Printing nba team of user's state
print(f"""
Aha! Out of the 29 nba teams (excluding Canada), there is (are) \
{len(nba_dict[user_state])} nba team(s) in your state!
""")

# Add random len(nba_dict[user_state]) and then select a random team!

# Declare team set, with the teams of user's state
team_set = set(nba_dict[user_state])

# Store user nba team from their state
user_team = input(prompt = f"""
Please write the name of your team.
{team_set}""")
```

```
# Preparing a great gift for the user!
print(f"""
I know what is the best gift for your next birhtday!
A new T-shirt from this team:
{user_team}""")
```

```
----------------------------------------------------------------------------------
----------------------------------
Welcome to Python! I'm thrilled to see you here!

What's your name?felipe
Hi felipe, it's amazing to meet you.
May I ask your birth date? [Please use this format: mm/dd/yyyy] 05/04/1993

Aha! You have lived:
                - 29 years and 216.75 days!

I would like to know more about you.
Which state are you currently living?
[please use the two letter abreviation]MA

Aha! Out of the 29 nba teams (excluding Canada), there is (are) 1 nba team(s)
in your state!


Please write the name of your team.
{'Boston Celtics'}Boston Celtics

I know what is the best gift for your next birhtday!
A new T-shirt from this team:
Boston Celtics
```

9. Summary

Congratulations on finishing the fundamentals of Python! You are now ready to start creating more complex programs. In the next chapter, you will learn about conditional statements and loops.

In these chapter you have covered the following topics:

- Python data types
  - Strings
  - Numbers
  - Booleans
  - Binaries
  - Sets
  - Sequence Data Types
    - List
    - Tuples
    - Range
  - Dictionaries

AWESOME WORK!