MODULE 1.1

Building Blocks of Python

Program

- Introduction
- Building blocks of Python code
- Variables
  - Data types
  - Strings in Python
  - Lists and tuples
  - Sets and dictionaries
- Branching and loops
  - If-elif-else
  - For
  - While
- Functions
  - Built-in functions
  - Defining your own functions

Learning Objectives

- You will be able **to use** the following *building blocks* of Python code in your projects: expressions, variables, quoting, branching, loops, functions and exception handling
- You will be able **to distinguish** between different data types in Python code, like numbers, strings, Booleans, lists, tuples, dictionaries, sets and their characteristics and be able **to use** in-built type casting functions in scripts
- You will be able **to use** branching structures and loop structures in Python code
- You will be able **to call** built-in function in Python and **to define** your own functions to organize your code.

Building blocks of Python code

Comments in Python

- Comments are line that are ignored by the interpreter.
- There are two ways to create comments.
  - Every line that starts with # will be ignored by the interpreter.
  - Every lines between triple quotes are also ignored by the interpreter.
- Always document your document, but don’t overdo it… Find a balance

Expressions

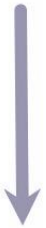
- Programming / coding all starts with **expressions**
- Expressions are combinations of values, variables, operators, and functions that *evaluate to a single* value, like:
  - Numeric / arithmetic expressions
  - String expressions
  - Conditional expressions
  - Logical expressions

Numeric Expressions

- Because of the lack of mathematical symbols on computer keyboards - we use “computer-speak” to express classic math operations.
  - Asterisk is used for multiplication → *
  - Exponentiation (raise to a power) looks different from in math → **.
  - % modulo operator
  - Order of operations

Operator	Operation
+	Addition
-	Subtraction
*	Multiplication
/	Division
**	Power
%	Remainder

Parenthesis
Power
Multiplication
Addition
Left to Right



String Expressions

- Concatenation: You can concatenate two or more strings together using the + operator. For example, `"hello" + "world"` will give you the string

`"helloworld"`.

- Repetition: You can repeat a string a certain number of times using the * operator. For example, `"hello" * 3` will give you the string `"hellohellohello"`.
- Indexing: You can access individual characters in a string using square brackets []. For example, `"hello"[0]` will give you the character `"h"`.
- Slicing: You can extract a substring from a string using the syntax

`[start:stop:step]`.

Conditional Expressions

- With the use of comparison operators, we can createBoolean expressions which are expressions that evaluate True or False based on the current state of variables?
- Comparison operators look at variables but do not change the variables.

Operator	Meaning
<	Less than
<=	Less or equal
==	Equal
>=	Greater or equal
>	Greater than
!=	Not equal

Building Blocks of Python Code

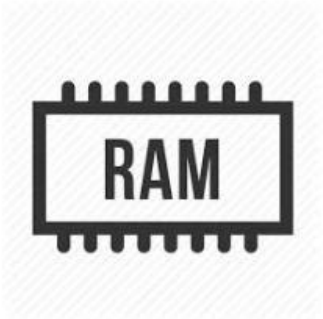
- Variables - hold temporary information in the script
  - Variable assignment
  - Data types: numbers, strings, Booleans, lists, tuples, dictionaries and sets
- Quoting - specifies how to represent characters in a string
- Branching
  - Conditions
  - Control Structures
  - Exception handling
- Loops - let you do the same thing over and over
- Functions - reusable code that you can use to perform a set of instructions

Variables

Variables

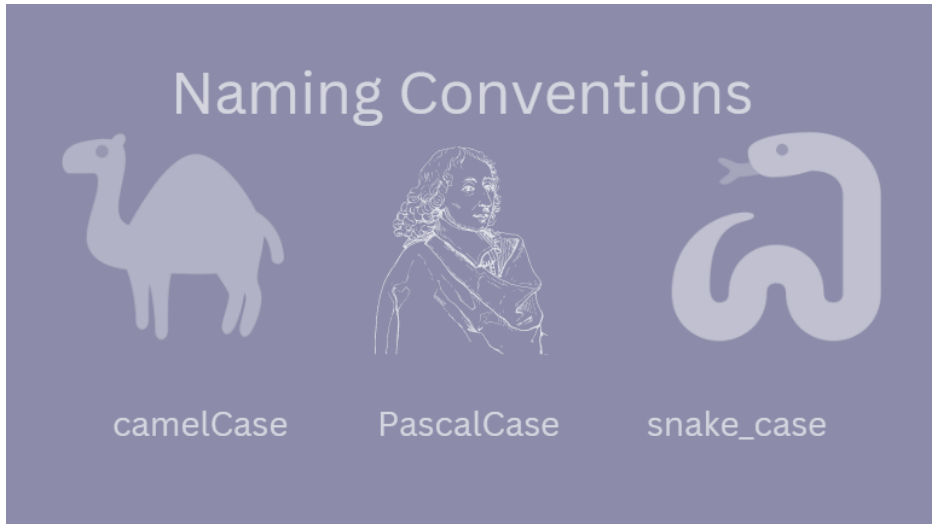
- Variables are used to store data.
- The data of the variable is stored in the memory.
- The data that is stored in the variable can be changed during runtime.
- In order to store data inside a variable, we need two parameters.
  - The name of the variable.
  - The data that will be stored.

```
x = 12
y = 'Hello'
```



Python Variable Name Rules

- Must start with a letter or underscore.
- Must consist of letters and numbers and underscores.
- Cannot start with a number.
- Case Sensitive



https://medium.com/@code.ceeker/naming-conventionscamel-case-pascal-case-kebab-case-and-more-dc4e515b9652

Assignment Statements

- We assign a value to a variable using the assignment statement (=).
- An assignment statement consists of expression on the right-hand side and a variable to store the result on the left-hand.
- Variable assignment and modification:
  - x = 2 - A new variable (x) has been created with the value of 2.
  - x = x + 2
    - A new assignment for x has been written. In order to store the data inside the variable (x), the mathematical expression must be evaluated.

Data Types

Data Type	Syntax	Iterable?	Mutable?
Integer	x = 5	No	No
Float	x = 5.0	No	No
Boolean	x = True	No	No
String	x = "hello"	Yes	No
List	x = [1, 2, 3]	Yes	Yes
Tuple	x = (1, 2, 3)	Yes	No
Set	x = {1, 2, 3}	Yes	Yes
Dictionary	x = {"a": 1, "b": 2}	Yes	Yes

None Data Type

- A None Data Type is a single object to say – “No Value”!
- It represents nothing.
- Do not mistake it with a 0 / False!

Using Operators on Different Data Types

- Using the + Operator on Numbers will result in addition.
- Using the + Operator on Strings will result in the concatenation.
- Some operations are prohibited. We cannot add an integer to a string.

```
>>> eee = 'hello ' + 'there'
>>> eee = eee + 1
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
TypeError: cannot concatenate 'str' and
'int’ objects
```

Type Casting

- Type casting function are used to create an object from another type. Examples:
  - int(): converts a value to an integer
  - float(): converts a value to a floating-point number
  - str(): converts a value to a string
  - bool(): converts a value to a Boolean (True or False) value
  - list(): converts a sequence or iterable to a list
  - tuple(): converts a sequence or iterable to a tuple

Strings

- A String Data Type is a sequence of characters.
- Each character is a single letter.

String	SOME TEXT								
Char	S	O	M	E		T	E	X	T

Defining Strings

- To define a String in Python, use either single quotes or double-quotes.
- Python interpreter treats them the same!
- That means we can use both options to define a string; the only difference is how we, as programmers decide to use them.

```
>>> string1 = "Text"
>>> string2 = 'Text'
>>> str_w_qoutes1 = "Text with
'quotes'"
```

Escape Characters

- There is a unique string in built-in the language, the backslash!
- With the use of the ”escape string,” we signal the interpreter that we want to ignore the next character!
- An escaped character is one character long!
- It comes after the backslash!

Escape Characters

Character	Name	Syntax
‘	Single quote	\’
“	Double quote	\”
\	Back slash	\\
n	New line	\n
t	Tab	\t
r	Carriage return	\r

Raw Strings

- Using Raw Strings, Python ignores all backslashes, not treating them as escape characters anymore!
- To define a raw string, we add ‘r’ before defining our string.

```
>>> file_path= 'c:\temp\newfile.txt'
>>> print(file_path)
c:      emp
ewfile.txt
>>> file_path= r'c:\temp\newfile.txt'
>>> print(file_path)
c:\temp\newfile.txt
```

Length of Strings

- When we want to know a string's length, we can use the built-in function à len()
- The len() function returns the length of a string for us.

```
>>> print(file_path)
c:\temp\newfile.txt
>>> len(file_path)
19
```

Indexing

- We can use square brackets – [] with the index of the wanted letter in them to find a single letter inside a string!
- The index is the place number of the letter.
- The count always starts with 0.

String	SOME TEXT								
Char	S	O	M	E		T	E	X	T
Index	0	1	2	3	4	5	6	7	8

Indexing Continued…

- We can index strings with negative numbers!
- In which case indexing occurs from the end of the string backward!

String	SOME TEXT								
Char	S	O	M	E		T	E	X	T
Index	0	1	2	3	4	5	6	7	8
Index	-9	-8	-7	-6	-5	-4	-3	-2	-1

String Methods

Method Name	Explanation	Example
count()	Counts the number of times a substring appears.	'the sun is the best'.count('the')
upper()	Turns all letters into uppercase.	‘A miXEd StrinG’.upper()
lower()	Turns all letters into lowercase.	‘aNOTher MiXeD stRING‘.lower()
replace()	Replaces all occurrences of a substring with another substring.	'I thpeak in lithp'.replace('th', 's')
find()	Finds the index of the first appearance of a substring. If none is found, -1 is returned.	'Hallelujah!'.find('jah')
isdigit()	Returns True/False, depending on if the string is built of only digits (0-9)	‘911’.isdigit()

Lists

- A list is a series of objects or items.
- Lists are defined using square brackets – [].
- They can hold any object in them – even another list!

```
>>> empty_list = []
>>> clothes_list = ['shirt', 'pants', 'socks']
>>> all_my_cool_items = ['Lava lamp', 3.141592654,
2 ** 10, ['a', 'b', 'c']]
```

Indexing

Accessing a Single Item from a List

- Indexing in lists acts just like in strings – using square brackets and beginning with an index of 0.

```
>>> all_my_cool_items = ['Lava lamp', 3.141592654,
2 ** 10, ['a', 'b', 'c']]
>>> all_my_cool_items[0]
'Lava lamp'
>>> all_my_cool_items[-2]
1024
>>>
```

Slicing

Accessing a Sub-List of a List

- This, again, is like slicing strings. While string slicing returns a string, accessing a slice of a list returns a list.

```
>>> square_nums = [1, 4, 9, 16, 25, 36]
>>> square_nums[2:5]
[9, 16, 25]
>>> square_nums[::-2]
[36, 16, 4]
>>>
```

Replacing Values

- In contrast to strings, individuals can easily replace values in a list, without creating a new one.

```
>>> nums = [1, 2, 3, 4, 5]
>>> nums[3] = 1000
>>> nums
[1, 2, 3, 1000, 5]
```

How long is a list?

- How can we count the number of items in a list?
- Use the len() function!
- len() tells us the number of elements of any sequence (strings, lists, and more!)

```
>>> friends = ['Abby', 'Alyssa', 'David']
>>> len(friends)
3
>>> print('I have {} friends!'.format(len(friends)))
I have 3 friends!
```

Using Arithmetic Operators

- Like strings, we can concatenate or multiply lists, by using ‘+’ and ‘*,’ respectively. This does not change the original list but creates a new list.

```
>>> a = [1, 2, 3]
>>> b = list('abc')
>>> a + b
[1, 2, 3, 'a', 'b', 'c']
>>> b * 3
['a', 'b', 'c', 'a', 'b', 'c', 'a', 'b', 'c']
```

Using the ‘in’ Operator

- Again, like strings, using in or not in with lists checks if an object is one of the items in a list.

```
>>> cities = ['London', 'Paris', 'Berlin']
>>> 'Berlin' in cities
True
>>> 'New York' not in cities
True
```

List Methods

- There are not many list methods, but every one of them is very useful!

```
>>> dir(list)
['append', 'clear', 'copy', 'count', 'extend', 'index',
'insert', 'pop', 'remove', 'reverse', 'sort']
```

Tuples

- Tuples are very similar to lists. They differ only in that tuples cannot be modified.
- To define a tuple, you use regular parentheses – () – instead of square brackets.

```
>>> names = ('Andy', 'Simon', 'Josh')
>>> type(names)
<class 'tuple'>
>>> names[1] = 'Alex'
Traceback (most recent call last):
  File "<pyshell#9>", line 1, in <module>
    names[1] = 'Alex'
TypeError: 'tuple' object does not support item assignment
>>>
```

List vs Tuple

	Lists	Tuples
Define	Square Brackets - [ ]	Regular Brackets - Parentheses - ( )
Attributes	Add data Remove data Change data	Cannot be changed
Size	+	−
Runtime	+	−

Creating a Set

- Sets are a new sequence type, created using curly brackets:

```
In [1]: letters = {'a', 'b', 'c', 'd', 'e'}

In [2]: letters
Out[2]: {'a', 'b', 'c', 'd', 'e'}

In [3]: type(letters)
Out[3]: set
```

Set Qualities

- Sets have two main qualities:
- Sets do not have any specific order.
  - They do not even support indexing!
- Sets hold only distinct values.
  - This means that there are no duplicate values!

```
In [5]: letters = {'a', 'a', 'b', 'a', 'b', 'c', 'a'}

In [6]: letters
Out[6]: {'a', 'b', 'c'}
```

Creating a Distinct List

- Problem:
- The result can be any order!

```
In [7]: nums = [1, 2, 1, 5, 5, 4, 4, 4, 3, 2, 1, 5, 1]

In [8]: nums
Out[8]: [1, 2, 1, 5, 5, 4, 4, 4, 3, 2, 1, 5, 1]

In [9]: nums_set = set(nums)

In [10]: nums_set
Out[10]: {1, 2, 3, 4, 5}

In [11]: new_nums = list(set(nums))

In [12]: new_nums
Out[12]: [1, 2, 3, 4, 5]
```

Using the in Keyword

- Just like in lists, tuples, and dicts – using the in keyword can check if an item is in the set.

```
In [24]: capitals = {'Doha', 'Amman', 'Baghdad'}

In [25]: 'Doha' in capitals
Out[25]: True
```

Set Methods and Operators

Method Name	Explanation	Example
add	Adds a new value to the set	{1, 2, 3}.add(4)
remove	Removes a value from the set	{1, 2, 3}.remove(2)
intersection	Returns a set with the items in the set that appear in the second set.	{1, 3, 5}.intersection({3, 4, 5}) >>> {3, 5}
set1 – set2	Returns a set with the items in set1 that don't appear in set2.	{1, 3, 5} – {3, 4, 5} >>> {1}
issubset	Returns True/False, depending on if all items in the set appear in the second set.	{0, 4, 7}.issubset(set(range(10))) >>> True

Creating a Dictionary

- Dictionaries are created with curly brackets!

```
In [42]: empty_dict = {}

In [43]: dict_with_one_val = {'key': 'value'}
```

- Every new item in the dictionary has a:
  - key – the name of the value
  - value – the value itself
- The key and value are separated by a colon.

Getting a Value from a Dictionary

- The value of a key can be fetched by using square brackets (like in lists!), but this time with the key name, not the index ID.

```
In [46]: person_info['age']
Out[46]: 37

In [47]: person_info['height']
Out[47]: 1.83
```

dict.keys()

- The keys method returns a sequence of all keys in the dictionary:

```
In [60]: for key in grocery_list.keys():
    ...:     print(key)
    ...:
eggs
milk
```

dict.values()

- The values method returns a sequence of all values in the dictionary:

```
In [61]: for value in grocery_list.values():
    ...:     print(value)
    ...:
12
3
```

dict.items()

- The items method returns a sequence of tuples.
- Each tuple is built up of (key, value).

```
In [63]: list(grocery_list.items())
Out[63]: [('eggs', 12), ('milk', 3)]
```

Unpacking dict.items()

- When each list item is a tuple of length 2, we can use unpacking in our for loop!
- This means that each iteration in our loop assigns 2 values – one to key and one to value.

```
In [64]: for key, value in grocery_list.items():
    ...:     print(key, ':', value)
    ...:
eggs : 12
milk : 3
```

Branching and loops

If-elif-else

```
if condition_1:
        # code to execute if condition_1 is True elif
condition_2:
        # code to execute if condition_1 is False and condition_2 is True
else:
        # code to execute if condition_1 and condition_2 are False
```

For and while loop

- For loop in Python:

```
for variable in iterable:
      # code to execute
```

- While loop in Python

```
while condition:
      # code to execute
```

Functions

List of Commonly Built-In Functions

- **print():** prints the specified message to the console or other standard output device.
- **len():** returns the number of items in an iterable object (such as a string, list, tuple, or dictionary).
- **input():** prompts the user to enter a value from the keyboard and returns the input as a string.
- **range():** generates a sequence of numbers from start to stop (exclusive) with the specified step size.
- **type():** returns the data type of a variable or value.
- **str(), int(), float(), bool():** converts a value to a string, integer, floating-point number, or Boolean value, respectively.
- **max(), min():** returns the maximum or minimum value in an iterable object (such as a list or tuple).
- **sum():** returns the sum of all the values in an iterable object (such as a list or tuple).
- **sorted():** returns a new list containing the sorted elements of an iterable object.

The print() Function

- The print function prints out any value to the screen.
- How could we have known that?
- The function help() to describes function's use.

```
>>> help(print)
Help on built-in function print in module builtins:

print(...)
print(value, ..., sep=' ', end='\n', file=sys.stdout, flush=False)
```

More Print Function Inputs

- The print function can receive more inputs. For example, we can configure the values that will separate different inputs. Or we can configure what we wish to have printed at the end of each print. The default is a newline character, but this can be changed to an empty string for the print to stay on the same line.

```
>>> help(print)
Help on built-in function print in module builtins:

print(...)
print(value, ..., sep=' ', end='\n', file=sys.stdout, flush=False)

Prints the values to a stream, or to sys.stdout by default.
Optional keyword arguments:

file: a file-like object (stream); defaults to the current sys.stdout.
sep:  string inserted between values, default a space.
end:  string appended after the last value, default a newline.
flush:  whether to forcibly flush the stream.
```

User Input

- We can instruct Python to pause and read data from the user using the input() function

- The `input()` function returns a string.

```
name = input('Who are you? ‘)
print('Welcome', name)
```

Converting User Input

- If we want to read a number from the user, then we must convert it from a string to a number using a type conversion function.

```
age = input(‘How old are you? ‘)
age = int(age)

print(‘You are’, age, ‘years old.’)
```

User Defined Functions

- Start the function definition with the def keyword, followed by the function name and parentheses containing any input parameters the function will take (if any).
- Use the return statement to specify the value that the function will return when it's called

- Example:

```
def my_function(param1, param2): result = param1 + param2

return result
```

Lab 1.1-1.7

Coding Challenges

Learning Objectives

- You will be able **to use** the following building blocks of Python code in your projects: expressions, variables, quoting, branching, loops, functions and exception handling
- You will be able **to distinguish** between different *data types* in Python code, like numbers, strings, Booleans, lists, tuples, dictionaries, sets and their characteristics and be able **to use** in-built *type casting* functions in scripts
- You will be able **to use** branching structures and loop structures in Python code
- You will be able **to call** built-in function in Python and **to define** your own functions to organize your code.