# **Day 3**

## Data Types

1. **Integers**: Represent numbers in an unlimited range. Limited only by the machine's memory. Whole numbers (negative and positive)
2. **Booleans:** Evaluate to **True** or **False**, equivalent to 1 or 0 respectively.
3. **Floating Point Numbers**: Represent double-precision numbers.
4. **Complex Numbers**: Represent numbers as a pair of double-precision numbers.
5. **Strings**: A sequence of Unicode characters, e.g., a word or a sentence that can be manipulated.
6. **Literals** are constant values that never change, such as strings, numbers, or special characters.

**print("The answer:", 5 + 2)** # 5 and 2 are literals, not assigned to a variable

## Operators

**+** : Addition

**-** : Subtraction

**\*** : Multiplication

**/** : Division (always returns a floating-point number)

## User Input

**age = input("Enter your age:") #stores user data within the var age**

## User Input Program

# Price per kg

**Asparagus = 30.54**

**Beetroot = 1.45**

**Broccoli = 14.43**

**Garlic = 35.81**

**Potatoes = 2.04**

# user input

**print("Enter the amount of Asparagus (kg):")**

**aspa = float(input())**

**print("Enter the amount of Beetroot (kg):")**

**beet = float(input())**

**print("Enter the amount of Broccoli (kg):")**

**broc = float(input())**

**print("Enter the amount of Garlic (kg):")**

**garl = float(input())**

**print("Enter the amount of Potatoes (kg):")**

**pota = float(input())**

# calculate

**aspa = aspa \* Asparagus**

**beet = beet \* Beetroot**

**b = (((broc \* Broccoli) \* 20) / 100) # 20% discounted price**

**broc = (broc \* Broccoli) - b # apply discount**

**garl = garl \* Garlic**

**p = (((pota \* Potatoes) \* 30) / 100) # 30% discounted price**

**pota = (pota \* Potatoes) - p # apply discount**

**total = aspa + beet + broc + garl + pota**

**print("The total the store has to pay: R", total)**

## Booleans

Boolean data type has corresponding integer values. There are only two possible values that a Boolean variable can have, True (1) or False (0).

When returning Booleans as strings they are seen as “True” and “False”, and never as “1” and “0”. True and False are case-sensitive in Python.

Boolean tests whether conditions are valid or not. The **three logical operators** used to test conditions between two arguments are:

* The and-operator
* The or operator
* The not operator

Considered False:

* False
* None
* Zero for any numeric data type, 0, 0.0, 0j
* An empty sequence or mapping. Like a list or tuple, ' ', ( ), [ ], { }
* Instances of user-defined classes, where a class that defines a \_\_bool\_\_() method returns zero or False.
* All values returned otherwise are always considered true. This means that many objects will always return true.

Operators and built-in functions that have a Boolean result always return (False or 0) or (True or 1). The Boolean or and and operations always return only one of the options, either True or False.

Manipulate Booleans

Syntax Description

* **A or B** If either a or b is True, then the result will be True. If both a and b are False then the result will be False.
* A and B If a and b are True, then the result will be True. Otherwise, the result will be False.
* Not A If a is True, False is returned. If a is False, True is returned.



## Booleans Program

**x = bool()**

**y = bool()**

**print ('Enter x as 1 or 0:')**

**x = int(input())**

**print ('Enter y as 1 or 0:')**

**y = int(input())**

**z = str(not bool(x and y))**

**print ('The Boolean value of x is', str(bool(x)))**

**print ('The Boolean value of y is', str(bool(y)))**

**print ('The Boolean value of (x and y) is', str(bool(x and y)))**

**print ('The Boolean value of (x not y) is', z)**

## Float

Floating Point Numbers (Floats)

Float is the data type that manages numbers with decimal places with very accurate precision. The float data type can be called as a function with zero or 1 argument of any data type. If no argument is given, then float returns 0.0.’

# Example of float function:

**x = float(5)**

**y = float("21.765")**

String value cast to a float must contain only numbers and only one occurrence of the dot (.) character.

**float("FF909A")**

# will raise an exception.

Formatting Floats

The f in the format part of the print statement indicates that the number to be formatted is a float.

The + sign indicates that the changed amount must be signed, which means that if the dollar compared to the rand decreases, that a − sign would be printed before the amount difference.

**print ("Today's Dollar price compared to the Rand: R%f" % 6.85871)**

# Output: Today's Dollar price compared to the Rand: R6.858710

**print ("Today's Dollar price compared to the Rand: R%.2f" % 6.85871)**

# Output: Today's Dollar price compared to the Rand: R6.86

**print ("Change since yesterday: R%+.2f" % 0.5)**

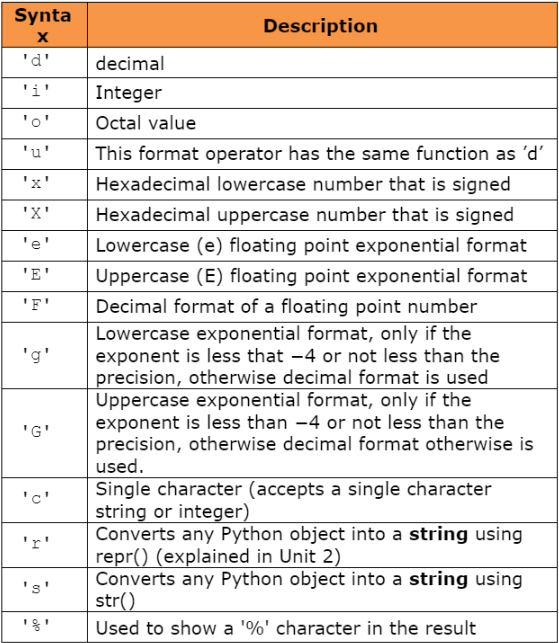
# Output: Change since yesterday: R+0.50

**print ("Negative amount: %+f" % -0.23)**

# Output: Negative amount: -0.230000

**print ("Positive amount: %+f" % 0.23)**

# Output: Positive amount: +0.230000

Formatting with Tokens

# Octal format (%o)

**print ("18 (decimal) as an octal value: %o" % 18)**

# Output: 18 (decimal) as an octal value: 22

# Uppercase Hexadecimal format (%X)

**print ("45 (decimal) as an uppercase (X) hexadecimal value: %X"% 45)**

# Output: 45 (decimal) as an uppercase (X) hexadecimal value: 2D

# Exponential format (%e)

**print("4.6 (float): (e) floating point exponential value: %e"% 4.6)**

# Output: 4.6 (float): (e) floating point exponential value: 4.600000e+00

## Complex Numbers

Complex numbers are represented by a real part and an imaginary part in Python. They can be created using the **complex(real, imag)** function. The real and imaginary parts of a complex number can be accessed using **.real** and **.imag** respectively.

The absolute value of a complex number is calculated using **abs()**. Complex numbers can be manipulated using arithmetic operations, and the real and imaginary parts are operated on separately.

## String

Strings in programming represent text as a sequence of characters. In Python, strings are of the data type "str" and are **immutable**, meaning they cannot be changed once created. They are composed of Unicode characters, making them versatile for handling different languages and symbols.

Strings can be created using single (' '), double (" "), or triple (''' ''' or """ """) quotes. An empty string can be created with str(), and when given an argument that is not a string, it converts it into a string representation of that argument.

## Lamda Expressions

Lambda expressions in Python allow the creation of small, anonymous functions. They are handy for situations where defining a full function might be overkill. Lambda functions are limited to a single expression.

# Creating a lambda function that returns the sum of its two arguments

**sum\_lambda = lambda a, b: a + b**

**result = sum\_lambda(3, 4)**

**print(result) # Output: 7**

# Defining a function that generates another function

**def make\_incrementor(n):**

**return lambda x: x + n**

# Creating a function that increments by 42

**f = make\_incrementor(42)**

**result1 = f(0)** # Output: 42

**result2 = f(1)** # Output: 43

# Creating a list of pairs

**pairs = [(1, 'one'), (2, 'two'), (3, 'three'), (4, 'four')]**

# Sorting the list based on the second element of each pair

**pairs.sort(key=lambda pair: pair[1])**

**print(pairs) # Output: [(4, 'four'), (1, 'one'), (3, 'three'), (2, 'two')]**

## Docstrings

In Python, **documentation strings (docstrings)** are used to provide information about the purpose, usage, and behavior of functions, modules, and classes. Following certain conventions for formatting docstrings is important for code readability.

Conventions for Docstrings:

1. The first line should be a brief, concise summary of the object's purpose. It should start with a capital letter and end with a period.
2. If there are more lines in the docstring, the second line should be blank to visually separate the summary from the rest of the description.
3. The following lines can include one or more paragraphs describing the object's calling conventions, side effects, etc.
4. Multi-line docstrings should be indented properly. The first non-blank line after the opening quotes determines the indentation level for the entire docstring. Indentation should be equivalent to spaces and not tabs.

Example of a Multi-line Docstring:

**def my\_function():**

**""" Do nothing, but document it.**

**No, really, it doesn't do anything. """**

**pass**

Function Annotations:

Function annotations are optional metadata about the types used in user-defined functions. Annotations are stored in the \_\_annotations\_\_ attribute of the function.

Parameter annotations are defined with a colon after the parameter name, followed by an expression evaluating the value of the annotation. Return annotations are defined with ->, followed by an expression.

**def f(ham: str, eggs: str = 'eggs') -> str:**

**print("Annotations:", f.\_\_annotations\_\_)**

**print("Arguments:", ham, eggs)**

**return ham + ' and ' + eggs**

Coding Style (PEP 8):

1. Use 4-space indentation and no tabs.
2. Limit lines to 79 characters.
3. Use blank lines to separate functions, classes, and blocks of code.
4. Put comments on a line of their own.
5. Use docstrings for functions and modules.
6. Use spaces around operators and after commas.
7. Name classes and functions consistently (UpperCamelCase for classes, lowercase\_with\_underscores for functions).
8. Use UTF-8 or plain ASCII for code meant to be used internationally.
9. Avoid non-ASCII characters in identifiers if code may be maintained by people speaking a different language.

Adhering to PEP 8's coding style guide promotes readable and maintainable Python code.