ENPM605 – Python Applications for Robotics

Lecture2 – Python Basics (Part 1)

Version – 1.2

Lecturer: Z. Kootbally

School: University of Maryland

Semester/Year: Spring/2024

2024/02/13

MARYLAND APPLIED
GRADUATE ENGINEERING

Overview

V1.2

1 Prerequisites

2 Learning Objectives

3 Packages and Modules

- Packages

4 Literal

5 Printing

6 Variables

7 Objects

- Object Identity
- Mutable Objects
- Immutable Objects

8 Memory Management

9 Dynamically Typed Language

- Type Inference
- Variable Rebinding

10 Indentation

11 Introduction to Flow Control

- Selection

12 Built-in Types

- Boolean Type
- None Type
- String Type

13 Numeric Type

- Integer Type
- Float Type
- Exercise

Changelog

V1.2

- v1.2 (02/13): Added numeric type.
- v1.0 (02/04): Original version.

Conventions

Conventions

- This is a *link*

📜 Terminology.
📝 Important note.
✈ Exercise.
📎 Resource.

New VSCode Plugin

1. Uninstall the ***pylint*** and ***autopep8*** extensions from Visual Studio Code.
2. Install the extension ***Ruff***
   - Configure the extension.

Learning Objectives

At the end of this lecture, you will learn the following:

- Difference between a package and a module.
- Different ways to import modules.
- Difference between mutable and immutable types.
- What variables are and how they reference data in memory.
- The boolean type and operators.
- The string type and operations on strings.

Modular Programming

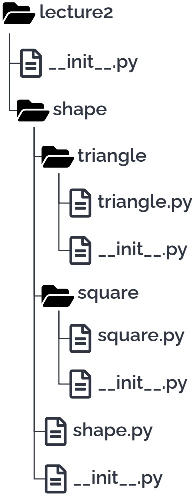
Modular programming refers to the process of breaking a large programming task into separate, smaller, more manageable subtasks or ***modules***.

Related modules can then be organized into ***packages*** to create a larger application.

- A ***module*** in Python is just a *.py file. In our case lecture1.py is a module.
- A ***package*** in Python is just a folder which contains *.py files. In our case, lecture1 is a package.
- We can define our most used functions in a module (in one place) and import it with the `import` keyword.

Python has tons of ***standard modules***. Standard and user-defined modules can be imported the same ways.

Create the Following Structure



```
lecture2
├── __init__.py
└── shape
    ├── triangle
    │   ├── triangle.py
    │   └── __init__.py
    ├── square
    │   ├── square.py
    │   └── __init__.py
    ├── shape.py
    └── __init__.py
```

triangle.py

```python
def compute_perimeter(a, b, c):
    return a + b + c

def compute_area(base, height):
    return 0.5 * base * height
```

square.py

```python
def compute_perimeter(a):
    return 4 * a

def compute_area(a):
    return a ** 2
```

Packages

__init__.py is a special file used to indicate that a directory should be treated as a Python package.

- In this file you can:
  - Initialize global variables that can be used in turn in the modules.
  - Place documentation.
  - Import modules.
  - Leave it blank.

Much of the Python documentation states that an __init__.py file must be present in the package directory when creating a package.

- *Implicit Namespace Packages* were introduced in Python 3.3 which allow for the creation of a package without any __init__.py (it can still be present but not required).
- However, it is still considered a best practice to include __init__.py for compatibility and to provide a place for package-specific initialization if needed.

In Python, variables and files with double leading and trailing underscores (dunders) have a special meaning. You should not name your files or Python symbols (functions, variables, etc) with dunders.

To Do

Use the functions `compute_perimeter(a)`, `compute_perimeter(a, b, c)`, `compute_area(a)`, and `compute_area(base, height)` in shape.py

Approach #1

Import the modules from their respective packages with `import package_name.module_name`

```
import square.square
import triangle.triangle

square_perimeter = square.square.compute_perimeter(4)
square_area = square.square.compute_area(4)
triangle_perimeter = triangle.triangle.compute_perimeter(3, 4, 5)
triangle_area = triangle.triangle.compute_area(3, 4)
```

Approach #2

Import and rename modules for easy use with
`import package_name.module_name as new_name`

```python
import square.square as square_mod
import triangle.triangle as triangle_mod

square_perimeter = square_mod.compute_perimeter(4)
square_area = square_mod.compute_area(4)
triangle_perimeter = triangle_mod.compute_perimeter(3, 4, 5)
triangle_area = triangle_mod.compute_area(3, 4)
```

Approach #3

Import what you only need with

`from package_name.module_name import a_function, a_variable, a_class, ...` or with

`from package_name.module_name import (a_function, a_variable, a_class, ...)`

```python
from square.square import (
    compute_perimeter as square_compute_perimeter,
    compute_area as square_compute_area
)
from triangle.triangle import (
    compute_perimeter as triangle_compute_perimeter,
    compute_area as triangle_compute_area
)

square_perimeter = square_compute_perimeter(4)
square_area = square_compute_area(4)
triangle_perimeter = triangle_compute_perimeter(3, 4, 5)
triangle_area = triangle_compute_area(3, 4)
```

Approach #4

Import everything with `from package_name.module_name import *`

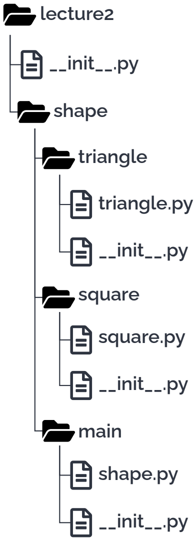
```
from square.square import *
from triangle.triangle import *

square_perimeter = compute_perimeter(4)
square_area = compute_area(4)
triangle_perimeter = compute_perimeter(3, 4, 5)
triangle_area = compute_area(3, 4)
```

As you can see, this approach can generate issues and ***should be avoided***.

- *Namespace Pollution* – Importing everything into the current namespace can lead to namespace pollution, where variable and function names from the imported module can clash with other names.
- *Readability and Maintainability* – It becomes challenging to understand which symbols (variables, functions, classes) come from the imported module. This can make your code less readable and harder to maintain, especially in larger projects.
- *Debugging* – Debugging can be more challenging when you are not sure where a particular symbol comes from, especially in complex codebases.

Create the Following Structure



Create the package **main** and move **shape.py** in that package.

To Do

Use the functions `compute_perimeter(a)`, `compute_perimeter(a, b, c)`, `compute_area(a)`, and `compute_area(base, height)` in shape.py

Since we moved shape.py in a package, we need to tell Python where to find square.py and triangle.py This is done by getting the absolute path to the workspace package lecture2

To Do

Include the following in shape.py

```python
import sys
import os.path

folder = (os.path.abspath(os.path.join(os.path.dirname(__file__), '..')))
sys.path.append(folder)
# Import your functions using approach #1, #2, or #3

# Call your functions
```

Explanation

- `(os.path.abspath(os.path.join(os.path.dirname(__file__), '..')))` is used to construct an absolute path to the directory one level above the directory where the current script resides.
- `sys.path` is a list in Python that contains the paths where Python looks for modules when you use import. By default, this list includes the directory of the script being executed among other standard locations.
- `sys.path.append(folder)`: Adds the path computed in the previous step to `sys.path`. This means Python will now also look in this directory when you try to import a module.

Main Program vs. Imported Module

In Python, `'__main__'` is the name of the scope in which top-level code executes. A module's `__name__` is set equal to `'__main__'` when executed directly and not imported.

A module can discover whether or not it is running in the main scope by checking its own `__name__`, which allows a common idiom for conditionally executing code in a module when it is run directly but not when it is imported.

```python
if __name__ == "__main__":
  # execute only if run directly and not imported
  # you can call functions to do some testing here
```

LITERAL

In programming, a ***literal*** refers to a notation for representing a fixed value in source code. Literals directly represent values in their native format, such as numbers, strings, or boolean values, without requiring computation or evaluation. Examples of literals:

- *String literals*: e.g., `"Hello"`, `'World'`.
- *Numeric literals*: e.g., `3`, `3.14159`.
- *Boolean literals*: e.g., `True`, `False`.

Printing

The `print` function in Python is a built-in function that outputs the specified message to the screen, or other standard output device. The message can be a string, or any other object, the object will be converted into a string before written to the screen.

- *Print literals*: `print("Hello")`, `print(3)`, `print(2.4)`
- *Print mathematical expressions*: `print(2 + 3)`
- *Print an empty line*: `print()`
- ? What is the output of `print('*' * 10)`?

The `print` function is a ***variadic function***, i.e., a function of indefinite arity, i.e., one which accepts a variable number of arguments.

```
print("Welcome", "to ENPM", 809, "E")
```

The ***print()*** function.

Variables

A variable is a name that refers to a value stored in memory.

- *Naming Convention* – Variable names in Python can include letters, digits, and underscores (_), but cannot start with a digit. Python is case-sensitive, so `name`, `Name`, and `NAME` are three distinct variables.
- *Assignment* – The assignment operator (`=`) is used to assign a value to a variable.

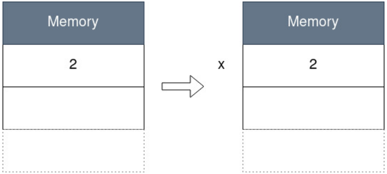
```python
# standard assignment
name = "Guido van Rossum"
# chained assignments
x = y = 10
# multiple assignments
name, age, role = "Guido van Rossum", 64, "BDFL Emeritus"
```

- *Declaration* – Variables in Python are dynamic and do not need to be declared with a specific data type, unlike in some other programming languages. This flexibility allows variables in Python to reference objects of different types over their lifetime.
  - `std::string name = "Guido";` (C++)
  - `String name = "Guido";` (Java)
  - `name = "Guido"` (Python)

Objects

In Python, everything is an object, and variables are more like references or names bound to these objects, rather than direct storage locations. When you create a variable in Python, you are essentially creating a reference to an object in memory.

```python
x = 2
```



The built-in function `type()` returns the type of an object.

```python
a = 10

print(type(a))  # <class 'int'>
print(type(100.5))  # <class 'float'>
print(type(print))  # <class 'builtin_function_or_method'>
```

The *type()* function.

Object Identity

The `id()` function in Python is a built-in function that returns the identity of an object. This identity is unique and constant for this object during its lifetime. The value returned by `id()` is an integer that acts as a unique identifier for the object in question.

```python
a = 10
b = 10.5
c = "hello"

print(id(a))  # 9793376
print(id(b))  # 140409546886992
print(id(c))  # 140409536180784
print(id(10))  # guess the output
print(id(10.5))  # guess the output
print(id("hello"))  # guess the output
```

You will get different results each time you run this program but they will stay the same for each object during the lifetime of the object.

The *id()* function.

Mutable Objects

Mutable objects can have their state changed after they are created. This means you can change, add, or remove elements of a mutable object without creating a new object.

Common mutable types in Python include:

- Lists (`list`)
- Dictionaries (`dict`)
- Sets (`set`)
- User-defined classes (unless explicitly made immutable)

Immutable Objects

Immutable objects cannot be changed after they are created. Any operation that tries to modify an immutable object will instead create a new object.

Common immutable types in Python include integers (int), floats (float), strings (str), and tuples (tuple).

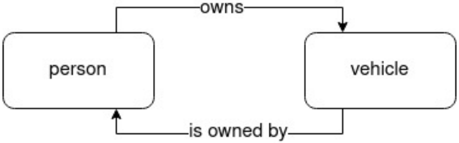
Common immutable types in Python include:

- Integers (`int`)
- Floats (`float`)
- Strings (`str`)
- Tuples (`tuple`)

Memory Management

In Python, the way memory is managed for variables is more complex and abstracted than the traditional stack and heap division found in lower-level languages like C. Python, being a high-level language, uses a private heap for storing objects.

- *Reference Counting*: Python uses reference counting for memory management. This means that each object keeps track of how many references are pointing to it. Once there are no references to an object (i.e., its reference count drops to zero), the object is eligible for garbage collection, which frees up the memory.
- *Garbage Collection*: Besides reference counting, Python (specifically, the CPython implementation) also has a garbage collector for detecting and dealing with cycles in the reference graph, preventing memory leaks that cannot be resolved by reference counting alone.



Dynamically Typed Language

Python being a dynamically typed language means that the type of a variable is determined at runtime, not in advance. This contrasts with statically typed languages, where the type of a variable must be explicitly declared and does not change over time.

Characteristics of Dynamic Typing

- *Type Inference* – Python automatically infers the type of a variable based on the value assigned to it. You do not need to declare a variable's type explicitly.
- *Variable Rebinding* – In Python, you can ***rebind variables to objects of different types***. This flexibility allows for more concise and potentially more readable code, but it also means that developers need to be aware of the types of their variables to avoid type-related errors.
- *Late Binding* – The types of variables are checked only at runtime, which means that errors related to type mismatches are not caught until the code is executed. This can lead to runtime errors that would be caught at compile-time in statically typed languages.

Type Inference

- The type of `10` is `int`, since the variable `x` references `10`, the type of variable `x` is `int`.

```python
x = 10
print(type(x))      # <class 'int'>
```

- The type of `10.5` is `float`, since the variable `y` references `10.5`, the type of variable `y` is `float`.

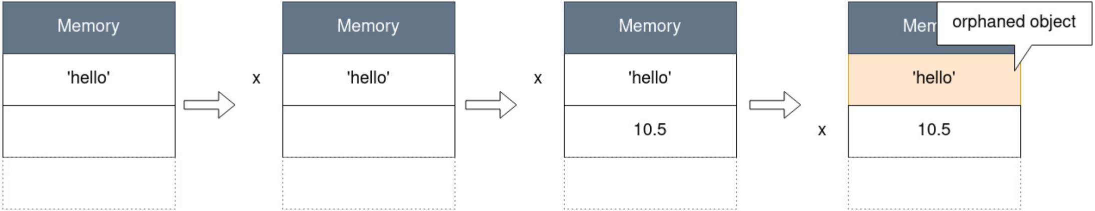
```python
y = 10.5
print(type(y))      # <class 'float'>
```

- The type of `'hello'` is `str`, since the variable `z` references `'hello'`, the type of variable `z` is `str`.

```python
z = 'hello'
print(type(z))      # <class 'str'>
```

Variable Rebinding

```python
x = "hello"
print(type(x))  # <class 'str'>
x = 10.5
print(type(x))  # <class 'float'>
```



Indentation

V1.2

Indentation

Indentation in Python plays a crucial role as it defines the organization and structure of code, particularly around blocks of code like loops, conditionals, and function definitions.

Unlike many other programming languages that use braces {} or keywords to define blocks of code, Python uses indentation to achieve this, making the readability of code a core part of the language's design philosophy.

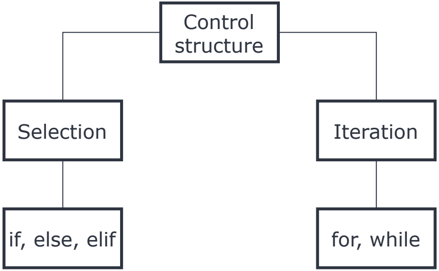
Many IDEs will automatically indent your code if you press ⌨ Enter after a colon. Example:

```python
def greeting(name):
    print("Hello ", name)
```

Flow Control

Flow control in Python refers to the way in which the execution of code statements is regulated or directed. It determines the order in which statements and blocks of code are executed or evaluated based on specific conditions or loops.

Flow control is fundamental to programming, as it allows developers to make decisions in their code, perform actions repeatedly, and manage the execution path of their programs.



Selection

Selection is used for decisions or branching (choosing between 2 or more alternative paths). The different types of selection statements in Python are: **`if`**, **`else`**, and **`elif`**.

The if Statement

- Syntax

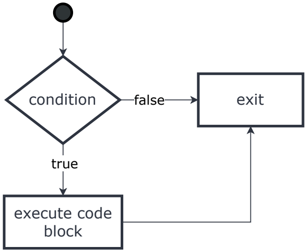
```
if condition:
    code block
```

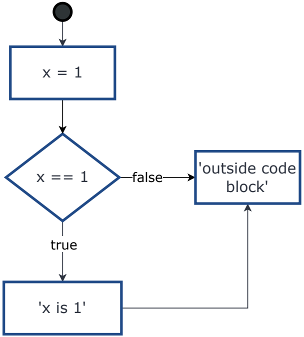
- Example

```
x = 1

if x == 1:
    print("x is 1")

print("outside code block")
```

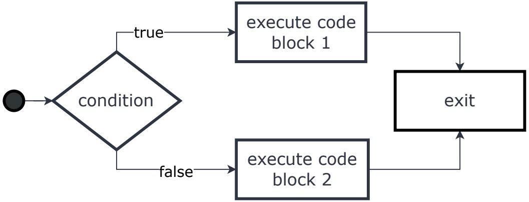




The if-else Statement

- Syntax

```
if condition:
   code block 1
else:
   code block 2
```

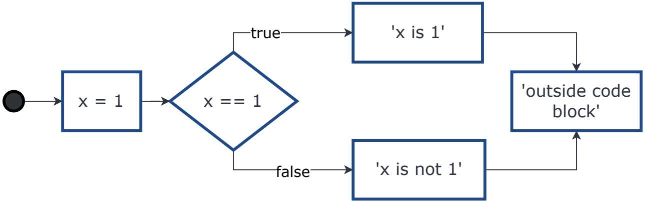


- Example

```
x = 1

if x == 1:
    print("x is 1")
else:
    print("x is not 1")

print("outside code block")
```



The elif Statement

In many cases we can group **if** and **else** statements together with the **elif** (**else if**) keyword.

- Syntax

```
if condition1:
  code block 1
elif condition2:
  code block 2
elif condition3:
  code block 3
...
else:
  code block n
```

- Example

```
x = 5

if x == 1:
    print("x is 1")
elif x == 2:
    print("x is 2")
elif x == 3:
    print("x is 3")
else:
    print("x is not 1, 2, or 3")
```

Built-in Types

Python comes with a set of built-in types that are part of the Python language standard. These types are always available and provide the foundational building blocks for structuring data in Python programs.

Built-in types covered in this lecture are displayed in the table below.

Category	Name	Type	Examples	Mutability
Numeric	Integer	`int`	`1, 123, 2000`	*Immutable*
Numeric	Floating point number	`float`	`2.5, 5.10, 10.0`	*Immutable*
Numeric	Boolean	`bool`	`True`, `False`	*Immutable*
Sequence	String	`str`	`"Hi", '2'`	*Immutable*
Sequence	List	`list`	`[1,'you',20.5]`	*Mutable*
Sequence	Tuple	`tuple`	`('hi',2,1.23)`	*Immutable*
Mapping	Dictionary	`dict`	`{"age":"30","i":10}`	*Mutable*
Set	Set	`set`	`{"a","b"}`	*Mutable*

Exhaustive list of Python built-in types.

Boolean Type

Python provides the Boolean type `bool` that can be either set to **`True`** or **`False`**.

Name	Type	Description
Boolean	`bool`	Logical value indicating **`True`** or **`False`**

- In Python, **`True`** is 1 and **`False`** is 0 but in a condition, any non-zero value or non-empty sequence (`list`, `str`, `tuple`, or `dict`) will evaluate to **`True`**.
- You can use the built-in `bool()` function to return or convert a value to a Boolean value, i.e., **`True`** or **`False`**, using the standard truth testing procedure.

```python
x = "hello"
if x:
    print("x is True")
else:
    print("x is False")
```

```python
print(bool(0))   # False
print(bool(1))   # True
print(bool(-2))   # True
print(bool(""))   # False
print(bool(" "))   # True

x = "hello"
print(bool(x))   # True
```

Relational Operators

A relational operator, compares ***the values*** of two operands and returns `True` or `False` based on whether the condition is met.

Let a `=` 1 and b `=` 2

Operator	Description	Example
`==`	`True` if the values of two operands are equal	a `==` b is `False`.
`!=`	`True` if the values of two operands are NOT equal	a `!=` b is `True`.
`>`	`True` if the value if lhs operand `>` rhs operand	a `>` b is `False`.
`<`	`True` if the value if lhs operand `<` rhs operand	a `<` b is `True`.
`>=`	`True` if the value if lhs operand `>=` rhs operand	a `>=` b is `False`.
`<=`	`True` if the value if lhs operand `<=` rhs operand	a `<=` b is `True`.

Logical Operators

Logical operators operate on propositions that only consider the values `True` or `False` as inputs.

Let `a = True` and `b = False`.

Operator	Description	Example
`and`	`True` if both op. are `True`	`a and b` is `False`.
`or`	`True` if at least one of the op. is `True`	`a or b` is `True`.
`not`	Reverse the logical state of operands or expressions	`not (a and b)` is `True`.

- Examples

```python
a = "hello"
b = 0
print(bool(a and b)) # False
print(bool(a or b)) # True
print(bool(not (a or b))) # False
```

Membership Operators

Membership operators test if an element belongs in a sequence.

Operator	Description
`in`	`True` if an element is in a sequence
`not in`	`True` if an element is NOT in a sequence

- Examples

```python
x = "hello"
print("h" in x)  # True
print("he" in x)  # True
print("O" in x)  # False
```

Identity Operators

Identity operators are used to compare the memory locations of two objects, essentially checking if they are the same object. When the identity operators are used, Python first calls the function `id()` for each object and then compares their identities.

Operator	Description
`is`	`True` if two objects have the same id
`is not`	`True` if two objects do not have the same id

- Examples

```
a = "hello"
b = 2
print(a is b)  # False
print(a is not b)  # True
```

None Type

In Python, None represents the absence of a value or a null value. It is an object of its own datatype, the NoneType. There is only one None object in the Python runtime, which ensures that all references to None are pointing to the same object in memory, making it a singleton.

- *Comparing with* None – To check if a variable is None, you should always use the identity operator is rather than the equality operator ==

```
a = None
if a is None:
    print("a is None")
```

- NoneType – The type of None is NoneType, reflecting its uniqueness in the Python type system.

```
print(type(None))  # <class 'NoneType'>
```

String Type

A Python string (`str`) is a sequence of characters. In Python, strings are ***immutable***, meaning once a string is created, the characters within it cannot be changed.

- Both single and double quotes can be used for string literals. According to the *documentation*, they are the same.

```
print(type("hello"))  # <class 'str'>
print(type('world'))  # <class 'str'>
```

- The `str()` function in Python is a built-in function that creates a string representation of an object. It returns a string version of the object if the object has a string representation.

```
number = 123
print(type(number))  # <class 'int'>
number_str = str(number)
print(type(number_str))  # <class 'str'>
```

Escape Sequences

Escape sequences allow you to include special characters in strings that would otherwise be difficult or impossible to type directly into the code. They are preceded by a backslash (`'\'`), which signals Python to interpret the subsequent character(s) in a special way.

Some of the common escape sequences in Python:

- `'\n'` – Newline; moves the cursor to the beginning of the next line.
- `'\t'` – Horizontal tab; moves the cursor to the next tab stop.
- `'\\'` – Backslash; inserts a literal backslash character
- `'\''` – Single quote; allows single quotes to be included in single-quoted strings
- `'\"'` – Double quote; allows double quotes to be included in double-quoted strings
- `'\r'` – Carriage return; moves the cursor to the beginning of the current line
- `'\b'` – Backspace; deletes the previous character in the string

```python
print("This string\nhas\nbeen\nsplit")
print("1\t2\t3\t4")
print("He said: \"I'm here!\"")
print('He said: "I\'m here!"')
print('''He said: "I'm here!"''')
print("""He said: "I'm here!" """) # note the space before the ending triple quotes
print("C:\Users\tony\notes.txt") # TODO: Fix this
```

String Interpolation

String interpolation in Python refers to the process of inserting or embedding expressions within string literals to be evaluated and formatted into a final string representation.

Python offers several methods for string interpolation, allowing developers to dynamically construct strings. Here are the most commonly used methods.

The % Operator

The % operator is used to format a set of variables enclosed in a ***tuple*** (a fixed size list), together with a format string, which contains normal text together with ***argument specifiers***, special symbols like `%s` and `%d`.

```python
name = "John"
age = 30
print("His name is %s and he is %d years old." % (name, age))
```

Examples of argument specifiers:

- `%c`: Character.
- `%s`: String.
- `%i` and `%d`: Signed integer.
- `%u`: Unsigned integer.
- `%o`: Octal integer.
- `%x`: Hexadecimal integer using lowercase letters (a-f).
- `%X`: Hexadecimal integer using uppercase letters (A-F).
- ...

The str.format() Method

Introduced in Python 2.6, the `str.format()` method is more powerful and flexible than the `%` operator. It uses curly braces as placeholders for variables to be interpolated into the string.

```python
name = "Alice"
age = 25
print("Her name is {} and she is {} years old.".format(name, age))
# With positional arguments
print("Her name is {1} and she is {0} years old.".format(age, name))
```

```python
# With keyword arguments
print("Her name is {name} and she is {age} years old.".format(name="Alice", age=25))
```

F-Strings (Formatted String Literals)

F-strings, introduced in Python 3.6, offer a concise and readable way to embed expressions inside string literals, using curly braces. The expressions are replaced with their values.

```python
name = "Alice"
age = 25
print(f"Her name is {name} and she is {age} years old.") # note the f before the string
print(F"Her name is {name} and she is {age} years old.") # note the F before the string
```

The difference between the last two lines is purely stylistic; there is no functional difference between them.

String Concatenation

String concatenation in Python is the process of combining two or more strings into a single string.

- *The + Operator* – The **+** operator is the most straightforward way to concatenate strings. You simply place it between the strings you want to join.

```python
first_name = "John"
last_name = "Doe"
print(first_name + " " + last_name)  # John Doe
```

- *The join() Method* – The `join()` method is a string method that concatenates the elements of an iterable (like a list or tuple) into a single string, using the string on which `join()` is called as the separator.

```python
words = ["Hello", "world"]
sentence = " ".join(words)
print(sentence)  # Hello world
```

- *String Interpolation* – All approaches of string interpolation can be used for string concatenation.

```python
first_name = "John"
last_name = "Doe"
full_name = "%s %s" % (first_name, last_name)
full_name = "{} {}".format(first_name, last_name)
full_name = F"{first_name} {last_name}"
```

Built-in Functions

Python provides many built-in *functions* for strings.

- Examples
  - `len()` returns the length of a string.

    ```
    print(len("hello"))  # 5
    ```

  - `str()` returns a string representation of an object.

    ```
    print(str(3), type(str(3)))  # 3 <class 'str'>
    print(str(3 + 4), type(str(3 + 4)))  # 7 <class 'str'>
    ```

String Methods

Python strings come with a variety of built-in *methods* that allow you to perform common manipulation and inspection tasks.

- Example
  - `capitalize()` a *copy* of the string with its first character capitalized and the rest lowercased.

    ```
    print("hello".capitalize())  # Hello
    ```

Ordered Sequence

Because strings are ***ordered sequences***, it means we can use indexing (to access only 1 character) and slicing (to access multiple characters at a time).

Each character in a string has an index. Positive indices start at index 0 from the beginning of the string. Negative indices start at index -1 from the end of the string.

String	'h'	'e'	'l'	'l'	'o'
+ Indices	0	1	2	3	4
- Indices	-5	-4	-3	-2	-1

Indexing

Indexing in Python refers to accessing individual characters, elements, or items in a sequence (such as strings, lists, tuples) using their position within the sequence.

- *Basic Indexing* – To access an element in a sequence, you use square brackets [] with the index of the element you wish to access:

```
print("hello"[0])  # h
greeting = "hello"
print(greeting[1])  # e
```

- *Negative Indexing* – The last element is accessed with index -1, the second-to-last with -2, and so on:

```
print("hello"[-5])  # h
greeting = "hello"
print(greeting[-4])  # e
```

- Predict the outputs of the following program:

```
print("hello"[5])
greeting = "hello"
greeting[0] = 'c'
print(greeting)
```

Slicing

Slicing enables you to retrieve a portion (a ***slice***) of the sequence by specifying a start index, an end index, and an optional step. Slicing is performed by using the colon (`:`) operator inside square brackets `[ ]`.

We can produce a slice by providing three integers separated by a colon `[start:stop[:stride]]`

- `start`: Index in the string where to start the slice (***inclusive***).
- `stop`: Index at where to end the slice (***exclusive***).
- `stride`: The stride refers to the step argument within the slicing syntax, which determines the interval at which elements are selected from the sequence being sliced. When not provided, the stride defaults to 1.

Slicing Without a Stride

When the stride is not provided, it defaults to 1.

```python
greeting = "hello"

# Slicing with positive indices
print(greeting[0:3])  # from start up to index 2
print(greeting[:3])  # from start up to index 2
print(greeting[:5])  # from start up to index 4
print(greeting[:])  # from start to end

# Slicing with negative indices
print(greeting[-5:])  # from start to end
print(greeting[-5:-2])  # from start up to index 2

# Slicing with positive and negative indices
print(greeting[-5:2])  # from start up to index 1
print(greeting[-5:4])  # from start up to index 3
```

Slicing With a Stride

```python
quote = "Learn Python, be happy!"
```

- A stride of 1 (the default) selects consecutive elements, which is the same as omitting the stride.

```python
print(quote[::]) # Learn Python, be happy!
```

- If `start` and `stop` are omitted, the entire sequence is considered, but elements are selected according to the stride.

```python
print(quote[::2])  # LanPto,b ap!
print(quote[::3])  # LrPh,eay
```

- A negative stride can be used to reverse the direction in which elements are selected. Care should be taken with negative strides, as the `start` and `stop` indices need to be compatible with the direction of traversal.

```python
print(quote[:8:-1])  # !yppah eb ,noh
```

  - Starting from the end of the string because `start` is omitted and `step` is -1, the slice will include every character in reverse order until it reaches the character at index 8 , which is not included in the result.

Exercise

Using the variable `quote`, write the Python code to do the following:

1. *Task 1* – Use only positive indices and a negative stride so the output is `'nohtyP'`
2. *Task 2* – Use only negative indices and a negative stride so the output is `'nohtyP'`
3. *Task 3* – Reverse the whole string so the output is `'!yppah eb ,nohtyP nraeL'`

```python
quote = "Learn Python, be happy!"
# code for task 1
# code for task 2
# code for task 3
```

String Interning

String interning is a method used in Python to optimize memory usage and improve performance when dealing with string objects. The basic idea behind string interning is to store only one copy of each distinct immutable string value, which means that strings with the same content share the same memory location.

Compile-time vs Runtime Interning

A string will not be interned unless it is loaded at compile time as a constant string (including expressions). Any string constructed at runtime (e.g., through methods and functions) will not be interned.

```python
a = "hello"
b = "hello"
c = "h" + "ello"
d = "he" + "llo"
d = "".join(["h", "e", "l", "l", "o"])
e = "{0}{1}".format("h", "ello")

print(a is b)  # True
print(c is d)  # False
print(a is d)  # False
print(a is e)  # False
```

Manual Interning

Python offers the `intern()` function from the anually intern a string. When you intern a string using `sys.intern()`, you ensure that all strings with the same content point to the same memory address.

```python
a = "hello"
b = "hello"
c = sys.intern("h" + "ello")
d = sys.intern("he" + "llo")
d = sys.intern("".join(["h", "e", "l", "l", "o"]))
e = sys.intern("{0}{1}".format("h", "ello"))

print(a is b)  # True
print(c is d)  # True
print(a is d)  # True
print(a is e)  # True
```

Numeric Type

Python 3 has ***three numeric types***. Numeric types in Python are ***immutable***.

Name	*Type*	*Description*
Integers	`int`	Whole numbers: `1, 123, 2000, etc`
Floating point numbers	`float`	Numbers with a decimal point: `2.5, 5.10, 10.0, etc`
Complex numbers	`complex`	Complex numbers: `(1+2j)`, `(3+8j)`, etc

Numeric Type

V1.2

Operators

- `+` Adds values on either side of the operator.
- `-` Subtracts right hand operand from left. hand operand.
- `*` Multiplies values on either side of the operator.
- `/` Divides left hand operand by right hand operand. This always does floating point division.
- `%` Divides left hand operand by right hand operand and returns remainder.
- `**` Performs exponential (power) calculation on operands.
- `//` Floor Division - The division of operands where the result is the quotient in which the digits after the decimal point are removed. But if one of the operands is negative, the result is floored, i.e., rounded away from zero (towards negative infinity).

Integer Type

Many programming languages have a limit to the size of an integer but the Python 3 `int` has no ***maximum*** size and can expand to the limit of the available memory of your machine.

- To create an integer from an object, the function `int(value[, base])` can be used.

```
print(int(3.5))             # 3

print(int('3'))             # 3

print(int('101011', 2))     # 43
```

Integer Interning

CPython also performs interning on integers.

```python
a, b = 20, 20
print(a is b)   # True

a, b = -5, -5
print(a is b)   # True

a, b = 200000000000000, 200000000000000
print(a is b)   # True
```

Float Type

Floating point numbers (`float`) are used to represent numbers having a fractional part (e.g., 12.5). The largest value for `float` on a 64 bit computer is 1.7976931348623157e+308. The smallest value for `float` on a 64 bit computer is 2.2250738585072014e-308.

- To convert an object to a float, the function `float(value)` can be used.

```
print(float("  3.5   \n"))  # 3.5
print(float('3'))           # 3.0
print(float(3))             # 3.0
```

Exercise

Print only the second half of a string.

Next Class

Python Basics – Part II