Table of Contents

Introduction

This document introduces the basic built-in types that will be presented in this course. Key points about each type will be listed here along with examples of some operations that can be performed with each type. A list of videos along with their timestamps will be included for more examples and explanations.

Note that this document does not cover all the built-in types that exist in Python. For a full list and explanation of all built-in types in Python, refer to the official Python documentation on built-in types.

Videos

- Types in general (0:00 - 3:27)
    - Covers int, str and float types
- Operations on types (3:27 - 7:21)
    - Covers some operations that can be performed on objects of int, str and float types
- Type conversion (7:22 - 12:00)
    - Covers how to convert between objects of type int, str and float
- String concatenation (0:00 - 2:57)
    - Covers how to concatenate or add strings together
- String methods (0:00 - 8:43)
    - Covers common some string methods
- Lists and the subscription order (0:00 - 7:04)
    - Covers list type and how to access items from a list using subscription
- Lists and mutability (0:00 - 4:17)
    - Covers how to access, add and remove items from a list, and its mutable property
- Tuples and mutability (1:49 - 3:07)
    - Covers tuple type, how to access items from a tuple using subscription, how to slice a tuple, and its immutable property
- Strings and mutability (3:07 - 4:40)
    - Covers str type, how to access specific characters in a string using subscription, obtaining its length, slicing strings, and its immutable property

1 Built-in Types

In Python, objects are created to represent data, and each object has an identity, type, and value.

- Identity: The identity of an object is determined when the object is created, and it is unique and immutable. Think of an object's identity like its address in the computer's memory. The address of an object can be found by using the `id()` function.

- Type: The type of an object determines the kinds of operations that can be performed on it, and how it can be stored in memory. Once an object is created, its type cannot be changed (but there are ways of "converting" objects from one type to another). The type of an object can be found by using the `type()` function.

- Value: The value of an object determines the memory contents for each particular object of that type. The value may or may not be mutable, depending on the type of the object. It is mutable if there is an operation that can be applied to the object to change its value.

Two *distinct* objects can have the same type and the same value, but will have different identities.

Since the kinds of operations that can be performed on an object depends on its type, objects of a particular type are either all mutable or all immutable. In other words, types with immutable objects have values that cannot be changed, and types with mutable objects have values that can be changed. The identity and type of an object is always immutable, but there are ways to "convert" an object of one type into an object of another type by taking an object of type A and creating a *new* object of type B with a value that is derived from the value of object A (e.g., converting a number object into a string object that represents the same number).

1.1 Integer Type (int)

The Python type int represents whole numbers. Objects of type int are immutable as there are no operations that can change the value of any int object.

`>>> 257` `257`	Creating positive numbers: the atomic expression `257` is evaluated, and an int object representing the number *two-hundred-fifty-seven* is created.
`>>> type(257)` `<class 'int'>`	The type of an object can be checked by using the `type()` function, and we can see that the object is of type int.

Operations such as addition, subtraction, and multiplication can be applied to objects of type int, which will result in new objects of type int.

`>>> -435` `-435`	Taking the negative of a number: The unary expression `-435` evaluates to the int object that represents the number *minus four-hundred-and-thirty-five*
`>>> 2 + 2` `4`	Addition: The binary expression `2 + 2` evaluates to the int object that represents the number *four*
`>>> 2 - 3` `-1`	Subtraction: The binary expression `2 - 3` evaluates to the int object that represents the number *minus one*
`>>> 2 * 0` `0`	Multiplication: The binary expression `2 * 0` evaluates to the int object that represents the number *zero*

1.2 Float Type (float)

Float type objects are similar to int type objects, but differ in that they represent decimal numbers. Like int type objects, float type objects are also immutable as there are no operations that can change the value of any float object.

`>>> 3.0`
`3.0`
`>>> type(3.0)`
`<class 'float'>`
	Creating positive numbers: the atomic expression `3.0` is evaluated, and a float object representing the decimal number *three-point-zero* is created.

Operations such as addition, subtraction, and multiplication can be applied to objects of type float, which will result in new objects of type float.

`>>> -6.3` `-6.3`	Taking the negative of a number: The unary expression `-6.3` evaluates to the float object that represents the number *minus six-point-three*
`>>> 2.0 + 2.0` `4.0`	Addition: The binary expression `2.0 + 2.0` evaluates to the float object that represents the number *four-point-zero*
`>>> 5.6 - 9.4` `-3.8`	Subtraction:The binary expression `5.6 - 9.4` evaluates to the float object that represents the number *minus three-point-eight*
`>>> 6.0 * 2.0` `12.0`	Multiplication: The binary expression `6.0 * 2.0` evaluates to the float object that represents the number *twelve-point-zero*

1.3 String Type (str)

The str type represents a sequence of characters. String objects are denoted by quotation marks (single or double). Note that if a string starts with a single quotation mark, it must end with the same type of quotation mark. String objects are immutable as there are no operations that can change the value of any string object.

`>>> 'hello world'`
`hello world`
`>>> type('hello')`
`<class 'str'>`
	Here, the atomic expression `'hello world'` is evaluated, and a str object representing the sequence of characters *hello world* is created.

More examples of expressions that evaluate to objects of type str:

`"world"`	Strings can also be denoted by double quotations, as shown here.
`"What's up"`	To use apostrophes in the string, use double quotes to surround the string instead.
`"4.2"`	This expression evaluates to an object of type str that represents the sequence of characters *4.2*.
`"100% chance"`	Various symbols can also be included in a string, like the percent % symbol here.
`'2+2'`	This expression here is an object of string type, so evaluating this doesn't give an int object of 4, but rather an object of str type which represents the sequence of characters *2+2*.

Operations such as subscription, string concatenation, and string multiplication can be applied to objects of type str, which will result in new objects of type str.

`>>> 'Hello'[0]` `H`	Subscription: `'Hello'[0]` evaluates to the str object that represents the character *H*
`>>> "Hello" + "World"` `HelloWorld`	String Concatenation: `"Hello" + "World"` evaluates to the str object that represents the sequence of characters *HelloWorld*
`>>> "Cat" * 3` `CatCatCat`	String Multiplication: `"Cat" * 3` evaluates to the str object that represents the sequence of characters *CatCatCat*

The str type (also known as str class) is complex, and provides methods that can be applied to objects of type str. Methods are similar to functions, but they are specific to objects of a particular type. They are used to perform operations on objects of a specific type. We will show two string methods here, but more string methods can be found in the string methods section of the Python documentation.

`>>> my_string = 'Hello'`	
`>>> my_string.upper()` `'HELLO'`	.upper(): This method returns a new str object that represents the sequence of characters in *my_string* in uppercase.
`>>> my_string.lower()` `'hello'`	.lower(): This method returns a new str object that represents the sequence of characters in *my_string* in lowercase.

1.4 Boolean Type (bool)

In Python, there are 2 special keywords that are pre-bound to two objects whose type is bool: `True` and `False`. They are often used in expressions, and can also be returned as results after an expression has been evaluated.

`>>> True`
`True`
`>>> type(True)`
`<class 'bool'>`
	Here, the atomic expression `True` is evaluated, and results in the bool object *True*.

Examples of expressions that evaluate to objects of type bool:

`>>> True or False` `True`	The *binary expression* `True or False` evaluates to the bool object *True*
`>>> 3 > 1` `True`	The *binary expression* `3 > 1` evaluates to the bool object *True*
`>>> "Apple" == "Cat"` `False`	The *binary expression* `"Apple" == "Cat"` evaluates to the bool object *False*

1.5 None Type (NoneType)

In Python, there is only one object of NoneType. This special object is referred to by the keyword None. The None object is typically used to indicate that there is *no value* or *nothing*. The None object is not displayed in the Python shell; however, it can be printed.

`>>> None`
`>>>`
`>>> print(None)`
`None`
`>>> type(None)`
`<class 'NoneType'>`
`>>> 3`
`3`
	Here, the atomic expression `None` is evaluated, and it is not shown unless the `print()` function is used to explicitly display it.
	In comparison to the atomic expression literal int `3`, which is displayed by default in the Python shell.

Functions in Python may have side effects. Side effects are any effects caused by the function, which includes printing out any values, and changing a mutable object's value. Some functions have only side effects and hence do not have a meaningful return object. In those cases Python returns the None object. For example:

```
>>> a = print('hello')
hello
>>> print(a)
None
>>> type(a)
<class 'NoneType'>
```

Here we assign the value returned by the `print()` function to the variable `a`; however, since the `print()` function's purpose is only to print the argument (and not to compute something that would be meaningful to return), it returns the None object, which is

assigned to `a`. The type of the object bound to `a` is NoneType. Thus, when we print `a` with the print function, Python displays `None` in the shell.

1.6 Function Type

In Python, all function names are bound to objects in memory. These objects are of function type. Objects that represent functions can have different types based upon how and where they are defined. In this section we will be covering functions that are pre-defined in Python. Such functions are called built-in functions.

Built-in functions have function names that are pre-bound to objects of type builtin function or method in memory, and a full list can be found in the Python documentation.

Some examples of built-in functions and their type:

```
>>> type(print)
<class 'builtin_function_or_method'>
>>> type(input)
<class 'builtin_function_or_method'>
>>> type(len)
<class 'builtin_function_or_method'>
```

Since function names are also identifiers, they can be bound to objects of other types; however, this is not recommended. Doing so can cause issues when using the built-in functions. We can see this here:

```
>>> print = 'a'
>>> type(print)
<class 'str'>
>>> print('hello world')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'str' object is not callable
```

Here, the identifier `print` is bound to `'a'`, which evaluates to a str object. When you try to call the print function later to display *hello world* (using the expression `print('hello world')`), there is an error because the identifier `print` is no longer bound to a function object.

1.7 Module Type (module)

Large programs are often broken down into smaller, more manageable files called modules. Modules are used to group related functions together, which organizes the code, and keeps the main code of the program uncluttered. Python has modules as part of its standard library that can be imported into a program. These modules provide functionality specific to certain tasks, for instance time related operations, or functionality related to random numbers.

To use modules, they must first be imported, which will then create objects of type module.

Some examples of Python modules and their type:

`>>> import time`
`>>> type(time)`
`<class 'module'>`
	`>>> import random`
	`>>> type(random)`
	`<class 'module'>`

Examples of some functions from the Python modules time and random:

`>>> time.sleep(3)`	This function pauses the program for a specified amount of seconds. Here, the program will pause for 3 seconds.
`>>> random.randint(1, 6)` `4`	This function returns a random int object with a value between 1 and 6, inclusive.

1.8 List Type (list)

The list type in Python represents a sequence of objects that may be of different types.

Lists can be created with your code using so-called *list displays*. A list display is an expression that starts with an opening square bracket [followed by zero or more expressions, separated by commas, followed by a closing square bracket].

`>>> alist = [1, 2, 3]`
`>>> print(alist)`
`[1, 2, 3]`
`>>> type(alist)`
`<class 'list'>`
	Here, the expression `[1, 2, 3]` is evaluated, and a list object holding 3 references, each to an object of type int, is created. The identifier `alist` can be used to refer to this list object.

More examples of expressions that create list objects are as follows:

`["apple", "pear", "strawberry", "peach"]`	A list can contain str objects
`[45, 1, 74.0, 1]`	Or, it can contain int or float objects
`[["hello", "world"], 3, 5.0, "!!!"]`	Or objects of different types, including lists

Operations such as subscription, list concatenation, and list multiplication can be applied to objects of type list.

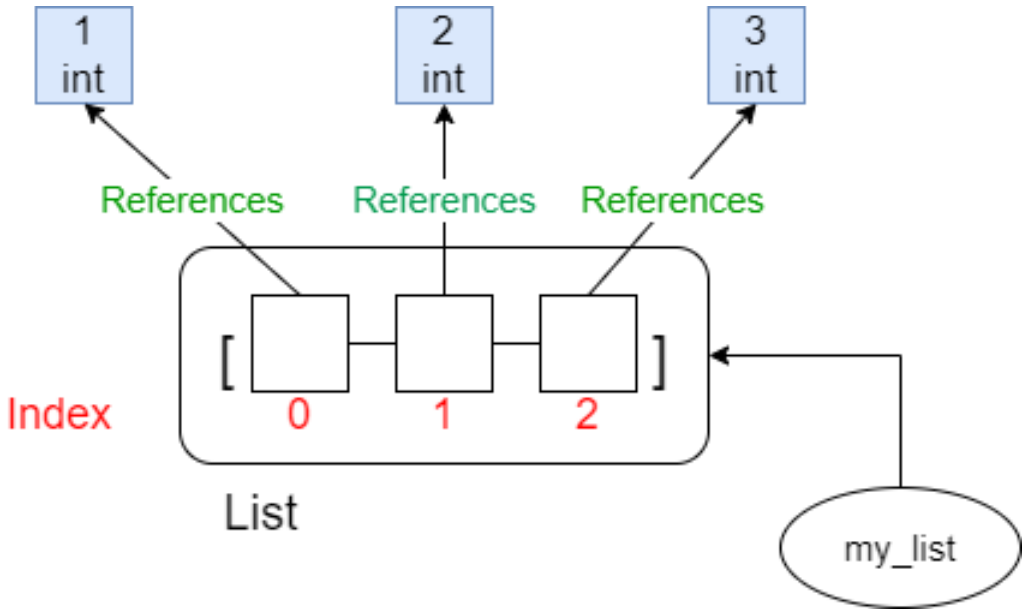
`>>> [1, 2, 3][0]` `1`	Subscription: `[1, 2, 3][0]` evaluates to the first item in the list, which is an int object representing the value *one*

`>>> ["a", "b"] + ["c"]` `["a", "b", "c"]`	List Concatenation: `["a", "b"] + ["c"]` evaluates to a new list object that holds three references to the str objects with the values *'a', 'b'* and *'c'*
`>>> ['cat'] * 3` `['cat', 'cat', 'cat']`	List Multiplication: `['cat'] * 3` evaluates to a new list object that holds 3 references to the same str object with the value *'cat'*

A list does not hold the actual objects themselves, but rather the references to the objects. The entries in a list are ordered and indexed by non-negative integers, starting at 0.

Let's take a look at this list:

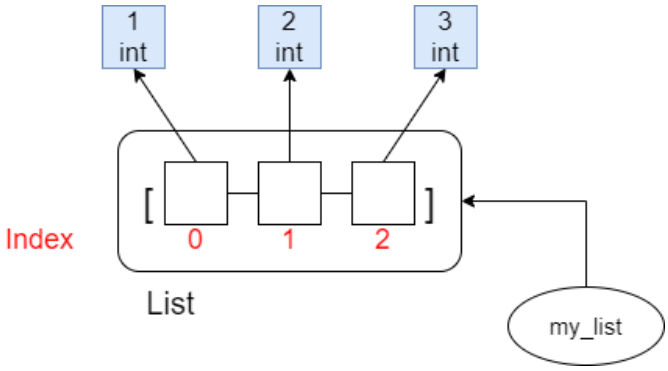
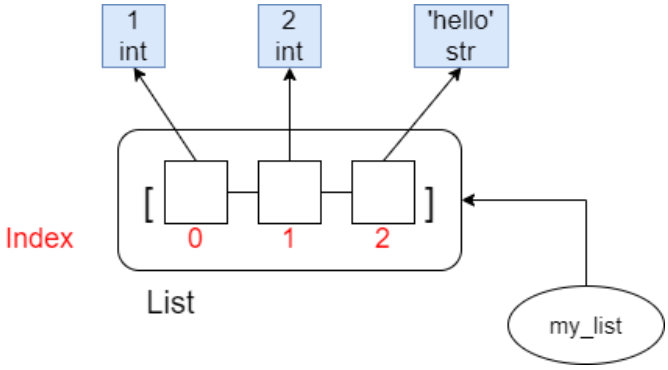
```
>>> my_list = [1, 2, 3]
```



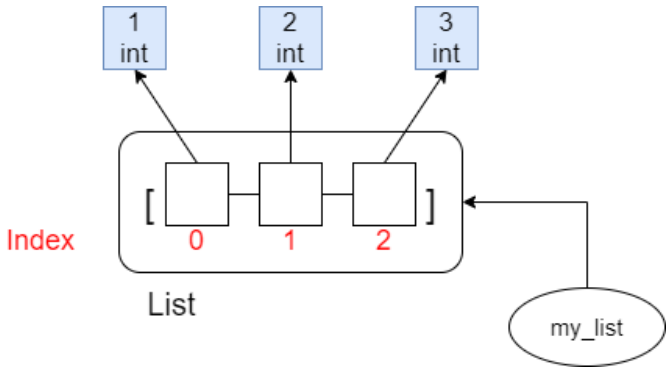
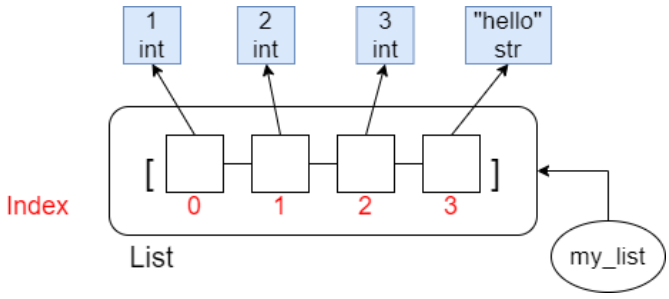
The list object that is created is bound to the variable *my_list*, and it holds 3 items, each being a reference to an object of type int. We can also see that the items in a list are indexed, starting with the first item in the list, which has an index of 0.

Lists are mutable, meaning that the references to the objects in the list can be changed after the list has been created.

An item at a specific position in the list can be replaced with another by using subscription:

`>>> my_list = [1, 2, 3]` 	Here, a list object is created and bound to the variable *my_list*. It holds 3 items, where each item references an object of type int.
`>>> my_list[2] = "hello"` 	Subscription operation is used to access the item at index two (third position in the list since indexes start at 0). The expression `"hello"` is evaluated to a str object, which then is assigned to the specified position in the list. Now, the item at index two of the list will reference an object of str type.

Similar to the str type, the list type also has methods that can be used on list objects. You can refer to the Python documentation for all the methods available to objects of list type. Here, we will show how a new item can be added to the end of the list by using the .append() method:

`>>> my_list = [1, 2, 3]` 	Here, a list object is created and bound to the variable *my_list*. It holds 3 items, where each item references an object of type int.
`>>> my_list.append("hello")` 	The *append* method is used to add an item to the end of the list. The list object changes so that it now contains 4 items, where the first 3 items references objects of type int, and the last item references an object of type str.

1.9 Tuple Type (tuple)

Tuples in Python are used to represent a sequence of objects. Objects in a tuple are ordered, and indexed by non-negative integers. Tuples are immutable, meaning that references inside a tuple object cannot be changed. A tuple display is an expression that starts with an opening round bracket (followed by zero or more expressions, separated by commas, followed by a closing round bracket.

Examples of expressions that create tuple objects are as follows:

`("apple", "pear", "strawberry", "peach")`	A tuple can contain str objects
`(45, 1, 74.0, 1)`	Or, it can contain int or float objects
`(["hello", "world"], 3, 5.0, "!!!")`	Or even objects of different types, including lists and tuples

Operations such as subscription, tuple concatenation, and tuple multiplication can be applied to objects of type tuple.

`>>> ('a', 'b', 'c')[1]` `b`	Subscription: `('a', 'b', 'c')[1]` evaluates to the second item in the tuple, which is a str object representing the value *'b'*
`>>> (1, 2) + (3, 4)` `(1, 2, 3, 4)`	Tuple Concatenation: `(1, 2) + (3, 4)` evaluates to a new tuple object that holds references to int objects with values *one, two, three and four*
`>>> ('cat') * 3` `('cat', 'cat', 'cat')`	Tuple Multiplication: `('cat') * 3` evaluates to a new tuple object that holds 3 references to the same str object with the value *'cat'*

1.10 Dictionary (dict)

Dictionaries in Python are used to store data as key:value pairs. A dictionary value can be of any object type, but keys must be an immutable object, and no two keys in a dictionary can be the same. A dictionary display is an expression that starts with an opening curly bracket { followed by zero or more key:value pairs, separated by commas, followed by a closing curly bracket }.

`>>> a = {'colour': 'blue'}`
`>>> type(a)`
`<class 'dict'>`
	Here, a dict object is created with a single item, where the key is a str object representing the sequence of characters *colour*, and the value associated with this key is a str object representing the sequence of characters *blue*.

More examples of expressions that create dict objects are as follows:

`{"name": "Sam", "age": 18, "colour": "red"}`
`{"years": [1987,2001,2017], "months": ["Jan", "Feb", "Aug"]}`

Dictionaries are indexed by their keys, which allows us to access values by using their associated keys:

```
>>> a = {'colour': 'blue'}
>>> a['colour']
blue
```

Key:value pairs of a dictionary can be added to, changed in, or removed from the dictionary. We'll use this dictionary object as an example:

```
>>> student = {"name": "Sam", "colour": "red"}
```

Adding items: `>>> student["age"] = 23`

```
>>> print(student)
{"name": "Sam", "colour": "red", "age": 23}
```

When `student["age"] = 23` is evaluated, a new key:value pair will be created in the dict object, where the key is a str object representing the string *'age'*, and the value associated with this key is an int object representing the number *twenty-three*

Changing a value:

```
>>> student["name"] = "Peter"
>>> print(student)
{"name": "Peter", "colour": "red", "age": 23}
```

When `student["name"] = "Peter"` is evaluated, the value associated with the key with the str object representing the string *'name'* is changed. The new value associated to this key will now have a str object representing the string *'Peter'*

Removing items:

Similar to the str and list type, objects of dict type also have methods. You can refer to the Python documentation for all the methods available to objects of dict type. Here, we will show how an item in a dict type object can be removed by using the .pop() method:

```
>>> student.pop("colour")
>>> print(student)
{"name": "Peter", "age": 23}
```

When `student.pop("colour")` is evaluated, the key (str object *'colour'*) and its associated value (str object *'red'*) will be removed from the dictionary.