course

February 20, 2025

[ ]:
```python
import builtins
from pprint import pprint
from typing import Any


def printr(*args):
    """Print the repr of the input."""
    if len(args) == 1 and not isinstance(args[0], str):
        builtins.print(repr(args[0]))
        return

    builtins.print(repr(" ".join(str(s) for s in args)))
```

1 Chapter 1: Data types

Introduction to Basic Types, Operations with Basic Types, Exercises

1.0.1 1.1: Basic types

There are several basic types in Python. The most common ones are:

- `int` - integer numbers: `1`, `2`, `3`, `42`, `100000`, `-5`, `-100000`

Also `int("100")`, `1_000_000`, `0x100` (hex), `0b100` (binary), `0o100` (octal)

- `float` - floating point numbers: `2.0`, `3.14`, `-0.5`, `0.0001`

Also `float("3.14")`, `1e3`, `1e-3`, `1.0e6`

- `str` - string: `'hello'`, `"world"`, `'This is a string!'`
  - Both `''` and `""` are used to define strings. They are equivalent.
  - It's also possible to define multi-line strings using triple quotes (`'''` or `"""`). This is useful when writing e.g. docstrings or SQL queries that span across several lines.
  - It can be useful to switch quote style when creating strings that need to contain quotes or apostrophes:

[ ]:
```python
print('He said: "Hello!"')
print("Don't do that!")
```

```
He said: "Hello!"
Don't do that!
```

- `bool` - The boolean type. Values are `True` and `False`.
- `NoneType` - The "null" type. Only one value: `None`.
  - Not the same as "undefined". A variable can be defined and have a value of `None`.

1.1 1.2: Compound types - Types that can contain other types

1.1.1 `list` - An ordered collection of objects/values of any type

- Created with `list(...)` or `[...]`
- `[1, 2, 3]`
- `list((1, 2, 3))`
- `["a", "b", "c"]`
- `[1, "a", 2.0, [1, 2, 3]]`

1.1.2 `tuple` - An ordered, *immutable* collection of objects/values of any type

- Created with `tuple(...)` or `(...)`
- `(1, 2, 3)`
- `tuple((1, 2, 3))`
- `("a", "b", "c")`
- `(1, "a", 2.0, (1, 2, 3))`

1.1.3 `dict` - Dictionary. A mapping between keys and values (like JSON):

- Created with `dict(key1=value1, ...)` or `{"key1": value1, ...}`
- `{"key1": "some string", "key2": 123, ...}`
- `{1: "one", 2: "two"}`

[ ]:
```
formats_by_country = {
    "DK": ["0200", "0500", "0700"],
    "DE": ["0600"],
    "PL": ["1100", "1110"],
}
```

1.1.4 `set` - An *unordered* collection of *unique* items

- Created with `set(...)` or `{...}`
- Note: `{}` are used for both sets and dictionaries. `{}` is an empty dictionary, not a set. Use `set()` to create an empty set.
- `{1, 2, 3}`
- `set((1, 2, 3))`
- `{"a", "b", "c"}`

- Sets are unordered, so you can't access individual items directly.
- Sets can only contain unique items. If you try to add a duplicate item, it will be ignored:

```
[ ]: s = {3, 2, 1, 2, 3}
     print(s)
```

```
{1, 2, 3}
```

- Sets are *mutable*, but the items in the set must be *immutable*.

1.2 1.3: Operations with Basic Types

1.2.1 Operations with int

```
[ ]: a = 10
     b = 3
```

```
[ ]: # Addition
     print(a + b)
```

```
13
```

```
[ ]: # Subtraction
     print(a - b)
```

```
7
```

```
[ ]: # Multiplication
     print(a * b)
```

```
30
```

```
[ ]: # Division
     print(a / b)
```

```
3.3333333333333335
```

```
[ ]: a = 10
     b = 3
```

```
[ ]: # Floor Division
     print(a // b)
```

```
3
```

```
[ ]: # Modulus
     print(a % b)
```

```
1
```

```
[ ]: # Exponentiation
     print(a**b)
```

```
1000
```

1.2.2 Operations with float

[ ]:
```python
a = 10.5
b = 3.2
```

[ ]:
```python
# Addition
print(a + b)
```

```
13.7
```

[16]:
```python
# Subtraction
print(a - b)
```

```
7.3
```

[17]:
```python
# Multiplication
print(a * b)
```

```
33.6
```

[18]:
```python
# Division
print(a / b)
```

```
3.28125
```

[ ]:
```python
a = 10.5
b = 3.2

# Floor Division
print(a // b)
```

```
3.0
```

[ ]:
```python
# Modulus
print(a % b)
```

```
0.8999999999999995
```

[ ]:
```python
# Exponentiation
print(a**b)
```

```
1852.7027964066515
```

[ ]:
```python
# Mixing int and float
a = 10
b = 2.5
result = a + b
print(result)
print(type(result))
```

```
12.5
<class 'float'>
```

Converting `float` to `int` truncates the decimal part (rounds towards 0)

[ ]:
```
print(int(10.5))
print(int(-10.5))
```

```
10
-10
```

Be careful with floating point arithmetic. Due to the way floating point numbers are stored in memory, you may get unexpected results when working with them:

[ ]:
```
c = 0.1 + 0.2
d = 0.3
print(c == d)
```

```
False
```

[ ]:
```
print(c)
```

```
0.30000000000000004
```

1.2.3 Operations with strings

[ ]:
```
print = printr
```

[ ]:
```
# Concatenation
str1 = "Hello"
str2 = "World"
concatenated = str1 + ", " + str2 + "!"
print(concatenated)
```

```
'Hello, World!'
```

[ ]:
```
# Repetition
repeated = str1 * 3
print(repeated)
```

```
'HelloHelloHello'
```

[ ]:
```
# Indexing / Slicing
first_letter = str1[0]
print(first_letter)

sliced = str1[0:3]
print(sliced)
```

```
'H'
'Hel'
```

```
[ ]: hello = "Hello, World!"
```

```
[ ]: # Upper and lower case
     upper_case = hello.upper()
     print(upper_case)

     lower_case = hello.lower()
     print(lower_case)
```

```
'HELLO, WORLD!'
'hello, world!'
```

```
[ ]: # Length
     length = len(hello)
     print(length)
```

```
13
```

```
[ ]: # Replace a substring
     replaced = hello.replace("World", "Python")
     print(replaced)
```

```
'Hello, Python!'
```

```
[ ]: # Split a string on a delimiter into a list of substrings
     split_str = hello.split(" ")
     print(split_str)
```

```
['Hello,', 'World!']
```

```
[ ]: email_address = "TineHansen123@gmail.com"
     split_email = email_address.split("@")
```

```
[ ]: username = split_email[0]
     domain = split_email[1]
     print("Username:", username)
     print("Domain:", domain)
```

```
'Username: TineHansen123'
'Domain: gmail.com'
```

```
[ ]: # Join list of strings into one string using delimiter
     joined_str = "-".join(["a", "b", "c", "d"])
     print(joined_str)
```

```
'a-b-c-d'
```

1.2.4 Exercise 1: Querying a list of names

Go to `github.com/connesy/python-course-BI-2025` and download the `exercises_part_1.ipynb` file.

Put it in your `DevStuff` folder and open it with VSCode.

```
[ ]: # Strip leading and trailing whitespace
     whitespace_str = "   Hello World   "
     stripped = whitespace_str.strip()
     print(stripped)
```

```
'Hello World'
```

```
[ ]: # Find the (starting) index of a substring
     string = "Hello World"
     index = string.find("World")

     print("World starts at index:", index)
     print(string[index:])
```

```
'World starts at index: 6'
'World'
```

```
[ ]: # Formatted strings, or "f-strings" - using variables in strings
     str1 = "Hello"
     str2 = "World"
     formatted = f"{str1}, {str2}!"
     print(formatted)
```

```
'Hello, World!'
```

```
[ ]: print = builtins.print
```

```
[ ]: from random import random

     measurement = random() * 100

     output = f"The measurement is: {measurement}"
     print(output)
```

```
The measurement is: 53.5185830826288
```

```
[ ]: names_string = "'Alice', 'Bob', 'Charlie'"
```

```
[ ]: query = f"""
     SELECT *
     FROM users
     WHERE users.name IN ({names_string})
     """
     print(query)
```

```

SELECT *
FROM users
```

```
WHERE users.name IN ('Alice', 'Bob', 'Charlie')
```

1.2.5 Exercise 2: f-strings

1.2.6 Operations with booleans

Boolean values:

```
[ ]: True, False
```

Logical not operator:

```
not True  : False
not False : True
```

Logical and operator:

```
True  and True  : True
True  and False : False
False and True  : False
False and False : False
```

Logical or operator:

```
True  or True  : True
True  or False : True
False or True  : True
False or False : False
```

Comparison operators evaluate to `True` or `False`:

```
10 == 20 : False
10 != 20 : True
10  > 20 : False
10  < 20 : True
10 >= 20 : False
10 <= 20 : True
```

Since comparisons evaluate to `True` or `False`, you can combine them with boolean operators, and you can use parentheses to group expressions:

```
[ ]: x = 1
     y = 2
     z = 3
     (x == 1 and z > x) or (y > z and x * 3 >= z)
```

```
[ ]: True
```

You can also chain comparisons, so `x < y <= z` is equivalent to `x < y and y <= z`, except that in the first case, `y` is evaluated only once.

This is useful when you want to check if a value is within a range:

```
[ ]: if 0 < x <= 100:
         ...
```

1.2.7 Exercise 3: Building an XOR gate

Other types can also behave as “truthy” or “falsy”.

“Falsy” values are treated as `False` in boolean expressions, and are typically empty values, e.g.:

```
bool(0)      : False
bool(0.0)    : False
bool("")     : False (empty string)
bool(())     : False (empty tuple)
bool([])     : False (empty list)
bool({})     : False (empty dictionary)
bool(set()) : False (empty set)
```

Likewise, “truthy” values are treated as `True` in boolean expressions, and are typically non-empty or non-zero values, e.g.:

```
bool(1)                    : True
bool(0.1)                  : True
bool("Hello")              : True (non-empty string)
bool((1,2))                : True (non-empty tuple)
bool([1, 2])               : True (non-empty list)
bool({'key': 'value'}) : True (non-empty dictionary)
bool({1, 2})               : True (non-empty set)
```

That means you can use them in `if` statements, `while` loops, etc.:

```
[ ]: data = [...]  # Some data

     if data:
         print("Data is not empty, processing...")
         ...

     else:
         print("Data is empty, nothing to process")
```

```
Data is not empty, processing…
```

1.3 1.4: Operations with compound data types

1.3.1 Indexing and slicing (lists, tuples)

```
[ ]: my_list = [1, 2, 3, 4, 5]
     print("Original list:", my_list)
```

```
Original list: [1, 2, 3, 4, 5]
```

Accessing elements by index

```
[ ]: first_element = my_list[0]
     print("First element:", first_element)

     last_element = my_list[len(my_list) - 1]
     print("Last element:", last_element)
```

```
First element: 1
Last element: 5
```

It's quite annoying to have to write `my_list[len(my_list) - n]` to acces the `n`-th last element of a list. Python allows you to use negative indices to access elements from the end of the list:

```
[ ]: last_element = my_list[-1]
     print("Last element:", last_element)

     second_last_element = my_list[-2]
     print("Second-last element:", second_last_element)
```

```
Last element: 5
Second-last element: 4
```

Slicing a list

Index ranges ("slices") are *inclusive* on the left and *exclusive* on the right:

```
[ ]: my_list = [1, 2, 3, 4, 5]
     sub_list = my_list[1:4]
```

```
[ ]: print("Sliced list from index 1 to (but not including) 4:", sub_list)
```

```
Sliced list from index 1 to (but not including) 4: [2, 3, 4]
```

This means that you can create consecutive slices without having to remember to end the first slice on `i` and start the next slice on `i+1`, as in e.g. `C`, `Java` and `JavaScript`:

```
[ ]: split = 2
     first_slice = my_list[0:split]  # Includes elements at index 0 and 1 (not 2)
     second_slice = my_list[split:4]  # Includes elements at index 2 and 3
     print(first_slice, second_slice)  # No overlap
```

```
[1, 2] [3, 4]
```

It's also possible to use negative indeces when slicing (these are also inclusive on the left and exclusive on the right):

```
[ ]: all_but_the_last = my_list[0:-1]
     print(all_but_the_last)

     last_element = my_list[-1]
     print(last_element)
```

```
[1, 2, 3, 4]
5
```

Since the last index is exclusive, how do you create a slice that includes the last element of a list?

In Python, you can create slices that are “too long” for the list:

[ ]:
```python
my_list = [1, 2, 3, 4, 5]
print(my_list[0:100])
```

```
[1, 2, 3, 4, 5]
```

… which means that you *could* use `len(my_list)` as the generic last index for a slice, as `len(my_list)` is 1 higher than the last index of the list:

[ ]:
```python
print(my_list[0 : len(my_list)])
```

```
[1, 2, 3, 4, 5]
```

However, Python has a clever trick!

You can leave out the first index to start at the beginning of the list, and leave out the second index to go to the end of the list (including the last element):

[ ]:
```python
all_but_the_first = my_list[1:]
print(all_but_the_first)

all_but_the_last = my_list[:-1]
print(all_but_the_last)
```

```
[2, 3, 4, 5]
[1, 2, 3, 4]
```

And, for completeness’ sake, you can also leave out both indices to get a copy of the entire list:

[ ]:
```python
my_copy = my_list[:]
```

When slicing, you can also specify the `step` size (the default is `1`).

The syntax is `list[start:stop:step]`:

[ ]:
```python
my_list = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
my_list[1:-1:2]
```

[ ]:
```
[2, 4, 6, 8]
```

You can omit the `start` and/or the `stop` index and only specify the `step` size:

[ ]:
```python
my_list[::3]
```

[ ]:
```
[1, 4, 7, 10]
```

… and the `step` size can also be negative, in which case the slice is taken in reverse order.

The caveat is that now `start` must be greater than `stop`:

```
[ ]: my_list[5:0:-1]
```

```
[ ]: [6, 5, 4, 3, 2]
```

It can be a bit confusing to use negative `step` sizes in general cases, but it's a handy way to reverse the entire list:

```
[ ]: my_list[::-1]
```

```
[ ]: [10, 9, 8, 7, 6, 5, 4, 3, 2, 1]
```

1.3.2 Exercise 4: List indexing

1.3.3 Lists operations

Adding an element to a list

```
[ ]: my_list = [1, 2, 3, 4, 5]
     my_list.append(6)
     print("List after appending 6:", my_list)
```

```
List after appending 6: [1, 2, 3, 4, 5, 6]
```

Inserting elements at a specific position

```
[ ]: my_list: list[Any] = [1, 2, 3, 4, 5, 6]
```

```
[ ]: my_list.insert(2, 2.5)
     print("List after inserting 2.5 at index 2:", my_list)
```

```
List after inserting 2.5 at index 2: [1, 2, 2.5, 3, 4, 5, 6]
```

Removing elements from a list

This will remove the first occurrence of the value `x` from the list. If `x` is not in the list, a `ValueError` will be raised.

```
[ ]: my_list: list[Any] = [1, 2, 2.5, 3, 4, 5, 6]
```

```
[ ]: my_list.remove(2.5)
     print("List after removing 2.5:", my_list)
```

```
List after removing 2.5: [1, 2, 3, 4, 5, 6]
```

Popping elements from a list

This "pops" the last element from the list and returns it. The list is modified in place:

```
[ ]: my_list = [1, 2, 3, 4, 5, 6]
```

```
[ ]: popped_element = my_list.pop()
     print("Popped element:", popped_element)
     print("List after popping an element:", my_list)
```

```
Popped element: 6
List after popping an element: [1, 2, 3, 4, 5]
```

Finding the index of an element

```
[ ]: my_list: list[Any] = [1, 2, 3, 4, 5]
```

```
[ ]: my_list.insert(3, "Hello")

     print(my_list.index("Hello"))
```

```
3
```

Counting occurrences of an element

```
[ ]: my_list = [1, 2, 2, 2, 3, 3, 4, 5, 5]
     print("Count of element 2:", my_list.count(2))
```

```
Count of element 2: 3
```

Sorting a list:

```
[ ]: my_list = [2, 5, 3, 4, 6, 1]
     my_list.sort()
     print("Sorted list:", my_list)
```

```
Sorted list: [1, 2, 3, 4, 5, 6]
```

Reversing a list *in place*:

```
[ ]: my_list = [1, 2, 3, 4, 5, 6]
```

```
[ ]: my_list.reverse()
     print("Reversed list:", my_list)
```

```
Reversed list: [6, 5, 4, 3, 2, 1]
```

This is different from using the slice `[::-1]` to reverse a list, since the `reverse()` method modifies the list in place, while the slice creates a new list:

```
[ ]: my_list = [1, 2, 3, 4, 5]

     my_copy = my_list[::-1]
     print("Original list:", my_list)
     print("Reversed copy:", my_copy)

     my_list.reverse()
     print("Original list, reversed:", my_list)
```

```
Original list: [1, 2, 3, 4, 5]
Reversed copy: [5, 4, 3, 2, 1]
Original list, reversed: [5, 4, 3, 2, 1]
```

If you want to append multiple items at once, you can use the `extend()` method to add all the elements from one list to the end of another list:

```
[ ]: my_list = [1, 2, 3, 4, 5]
     my_list.extend([6, 7, 8])
     print("my_list after extending with [6, 7, 8]:", my_list)
```

```
my_list after extending with [6, 7, 8]: [1, 2, 3, 4, 5, 6, 7, 8]
```

… or you can use the `+` operator to concatenate two lists, returning a new list:

```
[ ]: my_list = [1, 2, 3, 4, 5]
     new_list = my_list + [6, 7, 8]

     print("Original list:", my_list)
     print("New list:", new_list)
```

```
Original list: [1, 2, 3, 4, 5]
New list: [1, 2, 3, 4, 5, 6, 7, 8]
```

`extend()` and `+` is different from `append()`, which would add the entire list as a single element at the end of the list:

```
[ ]: my_list = [1, 2, 3, 4, 5]
     my_list.append([6, 7, 8])
     print("my_list after appending [6, 7, 8]:", my_list)
     print("Element at index 5:", my_list[5])
```

```
my_list after appending [6, 7, 8]: [1, 2, 3, 4, 5, [6, 7, 8]]
Element at index 5: [6, 7, 8]
```

… and you cannot add a single element to a list with `+`:

```
[ ]: my_list + 6
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Cell In[87], line 1
----> 1 my_list + 6

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

Checking if an element is in the list:

```
[ ]: my_list = [1, 2, 3, 4, 5]
```

```
[ ]: 4 in my_list
```

```
[ ]: True
```

1.3.4 Tuples

Tuples are similar to lists, but they are *immutable*.

This means that you can't change the elements* of a tuple after it has been created, nor can you add or remove elements from a tuple.

```
[ ]: my_tuple = (1, 2, 3, 4, 5)
```

```
[ ]: my_tuple.append(6)
```

```
---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)
Cell In[91], line 1
----> 1 my_tuple.append(6)

AttributeError: 'tuple' object has no attribute 'append'
```

```
[ ]: my_tuple.pop()
```

```
---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)
Cell In[92], line 1
----> 1 my_tuple.pop()

AttributeError: 'tuple' object has no attribute 'pop'
```

```
[ ]: my_tuple.insert(2, 2.5)
```

```
---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)
Cell In[93], line 1
----> 1 my_tuple.insert(2, 2.5)

AttributeError: 'tuple' object has no attribute 'insert'
```

```
[ ]: my_tuple.remove(2)
```

```
---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)
Cell In[94], line 1
----> 1 my_tuple.remove(2)

AttributeError: 'tuple' object has no attribute 'remove'
```

```
[ ]: my_tuple[3] = 10
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Cell In[95], line 1
----> 1 my_tuple[3] = 10

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

tuples are not as commonly used as lists, but they can be useful when you want to ensure that the data in the tuple doesn't change.

1.3.5 Dictionaries

Creating a dictionary

```
[ ]: athlete = {"name": "Alice", "age": 25, "city": "New York"}
     print(athlete)
```

```
{'name': 'Alice', 'age': 25, 'city': 'New York'}
```

Accessing elements by key

```
[ ]: name = athlete["name"]
     print(name)
```

```
Alice
```

Adding a new key-value pair

```
[ ]: athlete["email"] = "alice@example.com"
     print("Dictionary after adding email:")
     print(athlete)
```

```
Dictionary after adding email:
{'name': 'Alice', 'age': 25, 'city': 'New York', 'email': 'alice@example.com'}
```

Updating the value of an existing key

```
[ ]: athlete["age"] = 30
     print("After updating age:")
     print(athlete)
```

```
After updating age:
{'name': 'Alice', 'age': 30, 'city': 'New York', 'email': 'alice@example.com'}
```

Removing a key-value pair using pop()

```
[ ]: removed_value = athlete.pop("city")
     print("Removed value:", removed_value)
     print("Dictionary after removing city:")
print(athlete)
```

```
Removed value: New York
Dictionary after removing city:
{'name': 'Alice', 'age': 30, 'email': 'alice@example.com'}
```

Checking if a key exists in the dictionary

```
[ ]: has_city = "city" in athlete
     print("Does the dictionary have a 'city' key?", has_city)
```

```
Does the dictionary have a 'city' key? False
```

```
[ ]: has_age = "age" in athlete
     print("Does the dictionary have an 'age' key?", has_age)
```

```
Does the dictionary have an 'age' key? True
```

Getting the value of a key, with a default value if the key does not exist

```
[ ]: alice = {"name": "Alice", "age": 30, "phone": "+4528455749"}
     bob = {"name": "Bob", "age": 24}
     print(alice)
     print(bob)
```

```
{'name': 'Alice', 'age': 30, 'phone': '+4528455749'}
{'name': 'Bob', 'age': 24}
```

```
[ ]: print("Alice's phone number:", alice["phone"])
```

```
Alice's phone number: +4528455749
```

```
[ ]: print("Bob's phone number:", bob["phone"])
```

```
---------------------------------------------------------------------------
KeyError                                  Traceback (most recent call last)
Cell In[105], line 1
----> 1 print("Bob's phone number:", bob["phone"])

KeyError: 'phone'
```

dict.get()

```
[ ]: alice_phone = alice.get("phone", "Not provided")
     print("Alice's phone number:", alice_phone)
```

```
Alice's phone number: +4528455749
```

```
[ ]: bob_phone = bob.get("phone", "Not provided")
     print("Bob's phone number:", bob_phone)
```

```
Bob's phone number: Not provided
```

If no default value is specified, `get()` will return `None` if the key does not exist:

```
[ ]: print(bob.get("phone"))
```

```
None
```

Like JSON, dictionaries can also contain arbitrarily nested dictionaries, lists, tuples, etc.:

```
[ ]: print = lambda *args: pprint(*args, sort_dicts=False)
```

```
[ ]: alice = {
         "name": "Alice",
         "age": 25,
         "email": "alice@gmail.com",
         "address": {
             "street": "123 Main St",
             "city": "New York",
             "zip": 10001,
         },
         "siblings": ["Bob", "Charlie"],
     }
     print(alice)
```

```
{'name': 'Alice',
 'age': 25,
 'email': 'alice@gmail.com',
 'address': {'street': '123 Main St', 'city': 'New York', 'zip': 10001},
 'siblings': ['Bob', 'Charlie']}
```

```
[ ]: print = builtins.print
```

1.3.6 Sets

Creating a set

```
[ ]: fruits = {"apple", "banana", "cherry"}
     print(fruits)
```

```
{'banana', 'cherry', 'apple'}
```

A set only contains unique elements. If you try to add a duplicate element, it will be ignored:

```
[ ]: fruits = {"apple", "banana", "cherry", "banana", "apple"}
     print(fruits)
```

```
{'banana', 'cherry', 'apple'}
```

Adding an element to a set

[ ]:
```
fruits.add("orange")
print(fruits)
```

```
{'banana', 'orange', 'cherry', 'apple'}
```

Removing an element from a set

[ ]:
```
fruits.remove("banana")
print(fruits)
```

```
{'orange', 'cherry', 'apple'}
```

Checking if an element is in the set

[ ]:
```
is_apple_in_set = "apple" in fruits
print("Is 'apple' in the set?", is_apple_in_set)
```

```
Is 'apple' in the set? True
```

One of the very useful features of sets is that they have constant lookup time for membership checks, unlike lists, which have linear lookup time.

To showcase where this could be useful, let's consider the following example:

[ ]:
```
numbers_list = list(range(100_000_000))  # [0, 1, 2, ..., 99_999_999]
```

[ ]:
```
from time import time

start_time = time()
for x in [89_725_612, 47_586_915, 4_751_654, 485_695_489, 895_725_612]:
    x in numbers_list  # Check if x is in the list

end_time = time()
print(f"List search time: {end_time - start_time:.2f} seconds")
```

```
List search time: 2.81 seconds
```

[ ]:
```
numbers_set = set(range(100_000_000))  # {0, 1, 2, ..., 99_999_999}
```

[ ]:
```
start_time = time()
for x in [89_725_612, 47_586_915, 4_751_654, 485_695_489, 895_725_612]:
    x in numbers_set  # Check if x is in the set

end_time = time()
print(f"Set search time: {end_time - start_time:.7f} seconds")
```

```
Set search time: 0.0000494 seconds
```

[ ]:
```
try:
    del numbers_list
    del numbers_set
    del numbers
```

```
except NameError:
    pass
```

Union of two sets

[ ]:
```
set1 = {1, 2, 3}
set2 = {3, 4, 5}
union_set = set1.union(set2)
print("Union of set1 and set2:", union_set)
```

```
Union of set1 and set2: {1, 2, 3, 4, 5}
```

Intersection of two sets

[ ]:
```
intersection_set = set1.intersection(set2)
print("Intersection of set1 and set2:", intersection_set)
```

```
Intersection of set1 and set2: {3}
```

Difference of two sets

[ ]:
```
difference_set = set1.difference(set2)
print("Difference of set1 and set2:", difference_set)
```

```
Difference of set1 and set2: {1, 2}
```

Be careful with `set1.difference(set2)`, since the order of the sets matters! The operation should be read as "elements in `set1` that are not in `set2`", which is not the same as "elements in `set2` that are not in `set1`":

[ ]:
```
print("Difference of set1 with set2:", set1.difference(set2))
print("Difference of set2 with set1:", set2.difference(set1))
```

```
Difference of set1 with set2: {1, 2}
Difference of set2 with set1: {4, 5}
```

Symmetric difference of two sets

In contrast to `difference()`, the `symmetric_difference()` operation gives all elements that are in one of the sets, but not in both:

[ ]:
```
print("Symmetric difference of set1 and set2:", set1.symmetric_difference(set2))
print("Symmetric difference of set2 and set1:", set2.symmetric_difference(set1))
```

```
Symmetric difference of set1 and set2: {1, 2, 4, 5}
Symmetric difference of set2 and set1: {1, 2, 4, 5}
```

1.4 1.5: Multiple assignment

Multiple assignment is a Python feature that allows you to assign multiple values to multiple variables in a single line of code:

[ ]:
```
a, b = 1, 2
print(a)
```

```
1
```

The number of variables on the left side of the = sign must match the number of values on the right side, and the expressions on the right are evaluated from left to right.

All expressions on the right side are evaluated before any assignments are made, which means you can do things like this:

[ ]:
```
# Print the Fibbonaci sequence
a, b = 0, 1
while a < 50:
    print(a, end=", ")
    a, b = b, a + b
```

```
0, 1, 1, 2, 3, 5, 8, 13, 21, 34,
```

`b` and `a + b` are both evaluated before any assignments are made, so `a + b` uses the old values of `a` and `b` before they are updated.

The multiple assignment feature is actually a combination of two separate features: *tuple packing* and *sequence unpacking*.

Tuple packing is a feature that allows you to create a tuple without using parentheses:

[ ]:
```
t = 1, 2, 3
print(t)
```

```
(1, 2, 3)
```

This is mostly used when returning multiple values from a function:

[ ]:
```
def get_name_and_age(person):
    name = person["name"]
    age = person["age"]
    return name, age


alice = {"name": "Alice", "age": 25}
result = get_name_and_age(alice)
print(result)
```

```
('Alice', 25)
```

Sequence unpacking is a feature that allows you to assign multiple values from a sequence (like a list or tuple) to multiple variables:

[280]:
```
# Use tuple packing to create a tuple with three elements
t = 1, 2, 3
print("t:", t)
```

```
# Use tuple unpacking to assign the values of the tuple to three variables
a, b, c = t
print("a:", a, "b:", b, "c:", c)
```

```
t: (1, 2, 3)
a: 1 b: 2 c: 3
```

Together they form the multiple assignment feature, where the right side is first packed into a tuple, and then the values are unpacked into the variables on the left side.

Using that, we can also rewrite the `get_name_and_age` result:

[281]:
```
alice = {"name": "Alice", "age": 25}

# Use tuple unpacking to assign the return values of the function to two␣
 ↪variables
name, age = get_name_and_age(alice)

print("Name:", name)
print("Age:", age)
```

```
Name: Alice
Age: 25
```

This feature is also very useful in e.g. `for`-loops, which we'll see later.

2 Chapter 2: Equality and Identity

In Python, it's important to understand the difference between **equality checks** (`==`) and **identity checks** (`is`). These two operators are used to compare objects, but they do so in different ways.

2.1 2.1: Equality checks (==)

The `==` operator checks if the **values** of two variables are equal:

[ ]:
```
a = 10
b = 10
c = 20

print("a == b :", a == b)
print("a == c :", a == c)
```

```
a == b : True
a == c : False
```

[ ]:
```
list_a = [1, 2, 3]
list_b = [1, 2, 3]
print(list_a == list_b)
```

```
True
```

[ ]:
```python
dict_a = {"name": "Alice", "age": 25}
dict_b = {"name": "Alice", "age": 25}
dict_c = {"name": "Bob", "age": 30}
print(dict_a == dict_b)
print(dict_a == dict_c)
```

```
True
False
```

You can use `!=` to check if two variables are **not** equal:

[ ]:
```python
a = 10
b = 20
print("a == b :", a == b)
print("a != b :", a != b)
```

```
a == b : False
a != b : True
```

2.2 2.2: Identity checks (`is`)

Contrary to the `==` operator, the `is` operator checks if two variables reference the same object in memory:

[ ]:
```python
a = 1000
b = 1000
print("a == b :", a == b)
print("a is b :", a is b)
```

```
a == b : True
a is b : False
```

[ ]:
```python
list_a = [1, 2, 3]
list_b = [1, 2, 3]
print("list_a == list_b :", list_a == list_b)
print("list_a is list_b :", list_a is list_b)
```

```
list_a == list_b : True
list_a is list_b : False
```

What does it mean that two variables reference the same object?

When you create a variable and assign it a value, Python creates an object in memory to represent that value. The variable then *references* (points to) that object.

That means two variables can reference the same object:

[ ]:
```python
some_list = [1, 2, 3]
another_list = some_list
print(some_list == another_list)
print(some_list is another_list)
```

```
True
True
```

[ ]:
```
some_list[2] = 10
print(some_list)
```

```
[1, 2, 10]
```

[ ]:
```
print(another_list)
```

```
[1, 2, 10]
```

This can happen with any *mutable* object, like lists, dictionaries, sets etc. This means that you have to be careful when trying to create "copies" of objects:

[ ]:
```
alice = {"name": "Alice", "age": 25, "country": "Denmark", "city": "Aarhus"}
```

[ ]:
```
bob = alice
bob["name"] = "Bob"
```

[ ]:
```
print(bob)
```

```
{'name': 'Bob', 'age': 25, 'country': 'Denmark', 'city': 'Aarhus'}
```

[ ]:
```
print(alice)
```

```
{'name': 'Bob', 'age': 25, 'country': 'Denmark', 'city': 'Aarhus'}
```

If you need to create a copy of a mutable variable, you can use the `copy()` method, which is available for lists, dictionaries and sets:

[ ]:
```
alice = {"name": "Alice", "age": 25, "country": "Denmark", "city": "Aarhus"}
bob = alice.copy()
bob["name"] = "Bob"
print(alice)
print(bob)
```

```
{'name': 'Alice', 'age': 25, 'country': 'Denmark', 'city': 'Aarhus'}
{'name': 'Bob', 'age': 25, 'country': 'Denmark', 'city': 'Aarhus'}
```

You can use `is not` to check if two variables reference different objects:

[ ]:
```
a = [1, 2, 3]
b = [1, 2, 3]
print("a is b     :", a is b)
print("a is not b :", a is not b)
```

```
a is b     : False
a is not b : True
```

In a few cases, the standard way to compare objects is to use the `is` operator. For example, when checking if a variable is `None`:

[ ]:
```python
value = 10
if value is None:
    ...
```

Here we don't use `if value == None`, but `if value is None`.

This is because `None` is a singleton in Python, meaning there can only ever exist one instance of `None` in memory.

All variables that are `None` reference this single instance:

[282]:
```python
a = None
b = None
print(a is b)
```

```
True
```

The same is true for the boolean values `True` and `False`:

[ ]:
```python
a = True
b = True
print(a is b)

c = False
d = False
print(c is d)
```

```
True
True
```

So, when checking if a variable is `None`, `True` or `False`, you should use the `is` operator.

Otherwise, use the `==` operator to compare values.

3 Chapter 3: Control Flow

3.1 3.1: If-Else Statements

Basic if statement:

[ ]:
```python
x = 10
if x > 5:
    print("x is greater than 5")
```

```
x is greater than 5
```

We can use the fact from earlier that we can chain comparisons:

[ ]:
```python
x = 10
y = 20
if x > 5 and y > 5:
    print("Both x and y are greater than 5")
```

```
Both x and y are greater than 5
```

Python uses *indentation* to group statements, unlike many other languages that use curly braces (`{}`) or keywords like `begin` and `end` to define blocks of code.

The standard indentation is 4 spaces, but you can technically use any number of spaces. Just be consistent!

You can use `tab` - VSCode will replace it with 4 spaces when writing Python.

A code block ends when the indentation returns to the previous level:

```
[ ]: y = 3
     if y > 100:
         print("y is greater than 100")
         print("This is still inside the if block")
         print("This is also inside the if block")

     print("This is outside the if block")
```

```
This is outside the if block
```

If-Else statement:

The `else` block is executed if the condition in the `if` statement is `False`:

```
[ ]: y = 3
     if y > 5:
         print("y is greater than 5")
     else:
         print("y is not greater than 5")
```

```
y is not greater than 5
```

If-Elif-Else statement:

You can chain multiple `if` statements together using `elif` (short for "else if").

This will check each condition in order, and execute the block of code associated with the first condition that is `True`:

```
[ ]: z = 12
     if z >= 100:
         print("z is greater than or equal to 100")
     elif z >= 10:
         print("z is greater than or equal to 10 but less than 100")
     elif z > 5:
         print("z is greater than 5 but less than 10")
     else:
         print("z is 5 or less")
```

```
z is greater than or equal to 10 but less than 100
```

Again, the `else` block is executed if none of the `if`- or `elif`-conditions are `True`.

The if ... elif ... elif ... else construct is similar to the switch statement in other languages, or the CASE ... WHEN ... ELSE ... END construct in SQL.

3.2 Exercise 5: Positive, negative or zero?

3.3 Exercise 6: FizzBuzz

Nested if statements are possible, but can make the code harder to read. You can often simplify nested if statements by using elif:

[ ]:
```
x = 12

if x >= 10:
    if x >= 100:
        if x >= 1000:
            print("x is greater than or equal to 1000")
        else:
            print("x is greater than or equal to 100 but less than 1000")
    else:
        print("x is greater than or equal to 10 but less than 100")
else:
    print("x is less than 10")
```

```
x is greater than or equal to 10 but less than 100
```

Or the equivalent:

Python stops checking conditions as soon as one of them is True, and only executes the block of code associated with the first True condition.

[ ]:
```
if x >= 1000:
    print("x is greater than or equal to 1000")
elif x >= 100:
    print("x is greater than or equal to 100 but less than 1000")
elif x >= 10:
    print("x is greater than or equal to 10 but less than 100")
else:
    print("x is less than 10")
```

```
x is greater than or equal to 10 but less than 100
```

However, in some cases nested if statements are necessary. This is especially true when you need to check for multiple conditions that depend on each other:

[ ]:
```
year = 1900

if year % 4 == 0:
    if year % 100 == 0:
        if year % 400 == 0:
            print("Leap year")
        else:
            print("Not a leap year")
    else:
        print("Leap year")
else:
    print("Not a leap year")
```

```
Not a leap year
```

[283]:
```
athlete = {"name": "Alice", "age": 25, "height": 170, "gender": "female"}

if athlete["age"] >= 18:
    if athlete["gender"] == "female" and athlete["height"] >= 160:
        print("Eligible for the women's basketball team")

    elif athlete["gender"] == "male" and athlete["height"] >= 175:
        print("Eligible for the men's basketball team")

    else:
        print("Not tall enough for the basketball team")

else:
    print("Too young to join the basketball team")
```

```
Eligible for the women's basketball team
```

3.4 3.2: for-loop

The `for`-loop in Python differs a bit from other languages like C or JavaScript.

Like in `R`, Python's `for`-loop iterates over the items of any sequence (a list or a string), in the order that they appear in the sequence:

[ ]:
```
words = ["cat", "window", "defenestrate"]
for word in words:
    print(word, len(word))
```

```
cat 3
window 6
defenestrate 12
```

If you need to iterate over a sequence of numbers, you can use the built-in `range()` function:

[ ]:
```
for i in range(5):
    print(i)
```

```
0
1
2
3
4
```

The `range(n)` function generates a sequence of `n` numbers, starting from `0` and ending at `n-1`.

This makes it compatible with indexes in lists, since `range(n)` will generate all valid indexes for a list of length `n`:

[ ]:
```python
my_list = [1, 4, 9, 16, 25]

n = len(my_list)  # n = 5
for i in range(n):
    print(i, my_list[i])
```

```
0 1
1 4
2 9
3 16
4 25
```

It's also possible to start the range at a different number, and to specify a step size (just like when slicing lists).

The syntax is `range(start, stop)` or `range(start, stop, step)`:

[284]:
```python
list(range(10, 20))
```

[284]:
```
[10, 11, 12, 13, 14, 15, 16, 17, 18, 19]
```

[ ]:
```python
list(range(10, 40, 5))
```

[ ]:
```
[10, 15, 20, 25, 30, 35]
```

.. and you can also use negative step sizes:

[ ]:
```python
list(range(5, 0, -1))
```

[ ]:
```
[5, 4, 3, 2, 1]
```

3.5 Exercise 7: Printing characters in a string

You can also loop over the keys and values of a dictionary. This is done using the `items()` method of the dictionary:

[ ]:
```python
alice = {"name": "Alice", "age": 25, "city": "New York"}
for key, value in alice.items():
    print(f"{key}: {value}")
```

```
name: Alice
age: 25
city: New York
```

This works thanks to the *sequence unpacking* feature we saw earlier, since `items()` returns a sequence of key-value pairs:

```
[ ]: dict_items = list(alice.items())
     print(dict_items)
```

```
[('name', 'Alice'), ('age', 25), ('city', 'New York')]
```

The `for`-loop then *iterates over* each item in `dict_items` and unpacks the two values from each tuple into the variables `key` and `value`.

This is equivalent to doing the following:

```
[ ]: alice = {"name": "Alice", "age": 25, "city": "New York"}
     alice_items = list(alice.items())

     key, value = alice_items[0]
     print(f"{key}: {value}")

     key, value = alice_items[1]
     print(f"{key}: {value}")

     key, value = alice_items[2]
     print(f"{key}: {value}")
```

```
name: Alice
age: 25
city: New York
```

If you need to stop a loop before it has finished, you can use the `break` statement:

```
[285]: for i in range(10):
           print(i)
           if i > 2:
               break
```

```
0
1
2
3
```

If you need to skip the rest of the code block in the `for`-loop and continue with the next iteration, you can use the `continue` statement.

This is very useful e.g. for skipping processing of items in a list when not applicable:

```
[286]: alice = {"name": "Alice", "age": 25, "email": "alice@yahoomail.com"}
       bob = {"name": "Bob", "age": 30}
       charlie = {"name": "Charlie", "age": 35, "email": "charlie@gmail.com"}

       people = [alice, bob, charlie]
       for person in people:
           if "email" not in person:
               print(f"{person['name']} has no email address, skipping ...")
```

```
        continue  # Skip the rest of the loop body, continue with the next␣
↪iteration

    print(f"Emailing {person['name']} at {person['email']}")
    ...  # Code to send an email
```

```
Emailing Alice at alice@yahoomail.com
Bob has no email address, skipping …
Emailing Charlie at charlie@gmail.com
```

3.6 Exercise 8: Finding active members

3.7 3.3: while-loop

A `while`-loop is used to execute a block of code repeatedly as long as a condition is `True`:

[ ]:
```
i = 0
while i < 5:
    print(i, end=", ")
    i = i + 1
```

```
0, 1, 2, 3, 4,
```

Unlike the `for`-loop, you have to manually update the loop variable, since the condition can be anything:

[287]:
```
# Use a while-loop to find the remainder when dividing a number 'n' by 7
n = 100
remainder = n
result = 0
while remainder >= 7:
    result += 1
    remainder -= 7

print(f"{n}/7 =", result, "with remainder", remainder)
```

```
100/7 = 14 with remainder 2
```

You have to be careful with `while`-loops, since it's easy to create an infinite loop, which will run forever and stall your program:

```
i = 0
while i < 10:
    print("Oh no ...")
```

`while`-loops are useful when you don't know how many iterations you need to do in advance, or when you need to keep looking for something (or processing something) until some condition is met.

Like with the `for`-loop, you can use `break` and `continue` in a `while`-loop.

[ ]:
```python
people = [alice, bob, charlie]
while people:  # Loop as long as the list is not empty
    person = people.pop(0)  # Pull out the first person from the list

    if person["name"] == "Bob":
        print("Found Bob!")
        break

print("People left in the list:", len(people))
print("People:", people)
```

```
Found Bob!
People left in the list: 1
People: [{'name': 'Charlie', 'age': 35, 'email': 'charlie@gmail.com'}]
```

If you know in advance how many iterations you need to do, it's almost always easier to use a `for`-loop.

3.8 3.4: Nested loops

Loops, like `if`-statements, can be *nested* inside each other:

[288]:
```python
for x in [1, 2, 3]:
    for y in ["a", "b", "c"]:
        print(f"x: {x}, y: {y}")

    print("Inner loop complete\n")
```

```
x: 1, y: a
x: 1, y: b
x: 1, y: c
Inner loop complete

x: 2, y: a
x: 2, y: b
x: 2, y: c
Inner loop complete

x: 3, y: a
x: 3, y: b
x: 3, y: c
Inner loop complete
```

The inner loop iterates fastest, and will be “reset” and run again for each iteration of the outer loop.

3.9 Exercise 9: Multiplication table

3.10 3.5: Pass

The pass statement does nothing.

It can be used when a statement is required syntactically but the program requires no action. For example:

```
[ ]: for i in range(10):
         pass
```

This loop will run 10 times, but do absolutely nothing.

It might seem a bit silly, but it can be useful as a placeholder e.g. when you're writing code and know you need to create a function, but want to save the implementation for later:

```
[ ]: def process_person(person):
         pass  # TODO: Implement this function
```

4 Chapter 4: Functions

A function is a reusable block of code that only runs when it is *called*.

You can pass data, known as *parameters*, into a function, and the function can *return* data as output.

We can create a simple function that takes a name as input and prints a greeting:

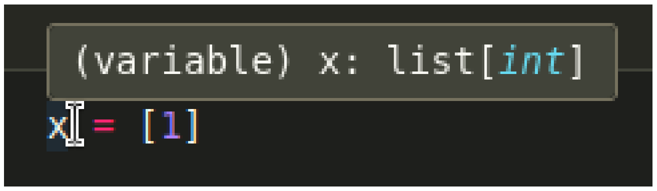
```
[ ]: def greet(name):
         """Print a greeting for the input name."""
         print(f"Hello {name}!")
```

The keyword `def` introduces a function *definition*. It must be followed by the function name and the list of *parameters* in parentheses.

The statements that form the *body* of the function start at the next line, and must be indented.

The first line of the function can optionally be a string; this string is the function's documentation string, or *docstring*.

Tools like VSCode will automatically pick up on this and show the docstring when you hover over the function name:



```
(variable) x: list[int]
x = [1]
```

To call the function, we write the function name followed by parentheses. Any input arguments should be placed within the parentheses:

[ ]:
```python
greet("Alice")
```

```
Hello Alice!
```

To return data from the function, we use the `return` statement. That way we can store the result of the function in a variable:

[ ]:
```python
def double(x):
    """Return the input multiplied by 2."""
    return x * 2

result = double(10)
print(result)
```

```
20
```

4.1 Exercise 10: Add two numbers

4.2 4.1: Default argument values

It is possible to specify a default value for one or more arguments. This creates a function that can be called with fewer arguments than it is defined to allow:

[ ]:
```python
def greet(name, greeting="Hello"):
    """Print a greeting for the input name."""
    print(f"{greeting} {name}!")
```

This function can be called with or without specifying the `greeting` argument:

[ ]:
```python
greet("Alice")
greet("Bob", "Hi there")
```

```
Hello Alice!
Hi there Bob!
```

However, since we didn't specify a default value for `name`, we have to provide a value for `name`. If we don't, Python will raise an error:

[ ]:
```python
greet()
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Cell In[243], line 1
----> 1 greet()

TypeError: greet() missing 1 required positional argument: 'name'
```

If we have a function with more than one default value specified, the arguments are "filled in" from left to right:

[289]:
```
def create_user(name, organization="0700", country="DK"):
    return {"Name": name, "Organizaition": organization, "Country": country}
```

[290]:
```
print(create_user("Alice"))
```

```
{'Name': 'Alice', 'Organizaition': '0700', 'Country': 'DK'}
```

[291]:
```
print(create_user("Bob", "0500"))
```

```
{'Name': 'Bob', 'Organizaition': '0500', 'Country': 'DK'}
```

[292]:
```
print(create_user("Charlie", "1100", "PL"))
```

```
{'Name': 'Charlie', 'Organizaition': '1100', 'Country': 'PL'}
```

Mutable default arguments

[299]:
```
def add_user(user, users=[]):
    users.append(user)
    return users
```

What happens when we call `add_user("Alice")`?

[300]:
```
print(add_user("Alice"))
```

```
['Alice']
```

Hopefully, as expected.

Now what happens when we call `add_user("Bob")`?

[301]:
```
print(add_user("Bob"))
```

```
['Alice', 'Bob']
```

Default values are evaluated *only once*, when the function is *defined, not called*:

[ ]:
```
n = 5
def f(number=n):
    print(number)

n = 10
f()
```

```
5
```

This is important to remember when using mutable default values, like lists or dictionaries.

It is one of the most common "gotchas" for beginners in Python.

If you don't want the default value to be shared between function calls, there's a common pattern to use `None` as the default value for the argument, and then set the argument to the desired value inside the function:

[ ]:
```
def add_user(user, users=None):
    if users is None:
        users = []

    users.append(user)
    return users

print(add_user("Alice"))
print(add_user("Bob"))
print(add_user("Charlie"))
```

```
['Alice']
['Bob']
['Charlie']
```

This works because the `users = []` statement is now executed every time the function is called, instead of only once when the function is defined.

However, this also means that we would be in charge of keeping track of the list of users ourselves:

[303]:
```
users = []

print(add_user("Alice", users))
print(add_user("Bob", users))
print(add_user("Charlie", users))
```

```
['Alice']
['Alice', 'Bob']
['Alice', 'Bob', 'Charlie']
```

4.3 Exercise 11: Add two numbers, with default values

4.4 4.2: Keyword arguments

When calling a function, you can use *keyword arguments* to specify the values of the arguments by name:

[ ]:
```
def create_user(name, organization="0700", country="DK"):
    return {"Name": name, "Organization": organization, "Country": country}

create_user(name="Charlie", organization="1100", country="PL")
```

[ ]:
```
{'Name': 'Charlie', 'Organization': '1100', 'Country': 'PL'}
```

Arguments called without specifying the keyword are called *positional arguments*. Keyword arguments must always follow positional arguments.

Using this syntax, the order of the arguments doesn't matter.:

[ ]:
```
create_user(organization="1100", country="PL", name="Charlie")
```

[ ]: {'Name': 'Charlie', 'Organization': '1100', 'Country': 'PL'}

We can also mix positional and keyword arguments (again, keyword arguments must follow positional arguments):

[ ]:
```
create_user("Dan", "0500", country="DK")
```

[ ]: {'Name': 'Dan', 'Organization': '0500', 'Country': 'DK'}

.. and we can even skip keyword arguments (as long as they have a default value):

[ ]:
```
create_user("Erik", country="DE")
```

[ ]: {'Name': 'Erik', 'Organization': '0700', 'Country': 'DE'}

4.5 Exercise 12: Using keyword arguments and default values

4.6 4.3: Arbitrary arguments and keyword arguments

In a function definition, two parameters of the form `*name` and `**name` can be used to specify *arbitrary* numbers of arguments and keyword arguments, respectively.

When a function parameter starts with `*`, it will take all the *extra positional arguments* that are passed to the function and put them in a tuple.

When a function parameter starts with `**`, it will take all the *extra keyword arguments* that are passed to the function and put them in a dictionary.

[ ]:
```
def func(x, *arguments, **keywords):
    print("x:", type(x), x, sep="\n   ")
    print("arguments:", type(arguments), arguments, sep="\n   ")
    print("keywords:", type(keywords), keywords, sep="\n   ")

func(1, 2, 3, 4, a=10, b=20)
```

```
x:
   <class 'int'>
   1
arguments:
   <class 'tuple'>
   (2, 3, 4)
keywords:
   <class 'dict'>
   {'a': 10, 'b': 20}
```

When is this useful?

[ ]:
```
def sum(*numbers):
    result = 0
    for number in numbers:
        result += number
    return result

print(sum(1, 2))
print(sum(10, 20, 30, 40, 50))
```

```
3
150
```

[ ]:
```
def create_product(name, sku, **metadata):
    name = name.title()  # Make sure name is title-case

    if len(sku) != 8:  # Make sure SKU is 8 characters
        raise ValueError("SKU must be 8 characters long")

    product = {"name": name, "SKU": sku}
    product.update(metadata)
    return product

create_product(
    "snake toy",
    sku="10681312",
    type="Toy",
    price=20.00,
    in_stock=True,
)
```

[ ]:
```
{'name': 'Snake Toy',
 'SKU': '10681312',
 'type': 'Toy',
 'price': 20.0,
 'in_stock': True}
```

Note that the order of the keyword arguments is guaranteed to match the order in which they were provided in the function call.

4.7 Exercise 13: Greetings all!

4.8 Exercise 14: Arbitrary Keyword arguments

4.9 4.4: Unpacking arguments

Remember the sequence unpacking feature we saw earlier?

[ ]:
```
values = [1, 2, 3]

x, y, z = values
print("x:", x, "y:", y, "z:", z)
```

```
x: 1 y: 2 z: 3
```

Just as we can "pack" function arguments into a tuple or dictionary using the * or ** operators, respectively, we can also "unpack" arguments from a sequence or dictionary into individual arguments using the same operators when calling the function:

[307]:
```python
def func(x, y, z):
    print("x:", x, "y:", y, "z:", z)
```

[308]:
```python
values = [1, 2, 3]
func(*values)
```

```
x: 1 y: 2 z: 3
```

The * operator unpacks the values from the list into the individual parameters.

Remember our sum function from earlier?

[ ]:
```python
def sum(*numbers):
    result = 0
    for number in numbers:
        result += number
    return result
```

We can use the * operator to unpack a sequence of numbers when calling it:

[ ]:
```python
numbers = range(100)
sum(*numbers)
```

[ ]:
```
4950
```

In the same manner, we can use the ** operator to unpack a dictionary of keyword arguments when calling a function.

Note: the keys of the dictionary *must* match the parameter names of the function:

[309]:
```python
def func(x, y, z):
    print("x:", x, "y:", y, "z:", z)
```

[310]:
```python
values = {"z": 3, "x": 1, "y": 2}
func(**values)
```

```
x: 1 y: 2 z: 3
```

We can see that, just like when calling a function with keyword arguments, the order of the keys in the dictionary doesn't matter - the value of the x key is mapped to the x parameter, y key to y parameter, etc.

This is very useful when you have a dictionary of arguments that you want to pass to a function, e.g. a configuration dictionary, or some data that is shared between multiple function calls:

[311]:
```python
def build_query(table_name, *column_names, database, limit):
    query = f"""\
    SELECT {", ".join(column_names)}
    FROM {database}.{table_name}
    LIMIT {limit}\
    """
    return query
```

[312]:
```
sql_config = {"database": "ENTERPRISE_DB", "limit": 100}
```

[316]:
```
print(build_query("users", "name", "age", "city", **sql_config))
```

```
SELECT name, age, city
FROM ENTERPRISE_DB.users
LIMIT 100
```

4.10 Exercise 15: Unpacking arguments

5 Chapter 5: Type annotations

5.0.1 5.1: What are type annotations?

Type annotations are a way to specify the *type* of a variable, function parameter, or function return value.

They are completely optional, but they can be very useful for documentation and for getting help from your code editor.

A *parameter type* annotation is written by adding a colon (`:`) after the variable name, followed by the type.

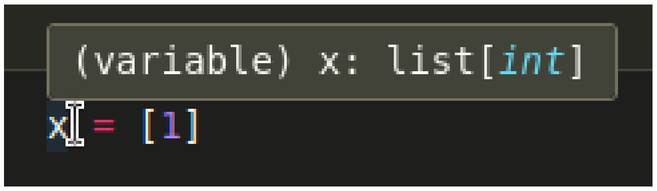
[317]:
```
def func(x: int):
    return x**2
```

A *return type* annotation is written by adding an arrow (`->`) after the closing parenthesis of the parameter list, followed by the return type:

[ ]:
```
def func(x: int) -> int:
    return x**2
```

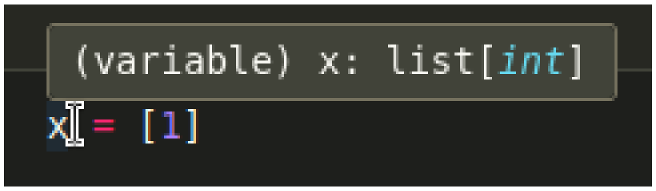
Now, VSCode (or more precisely, the `Python` and `Pylance` extensions) can give you better help when trying to call `func`.

Without the type annotations:



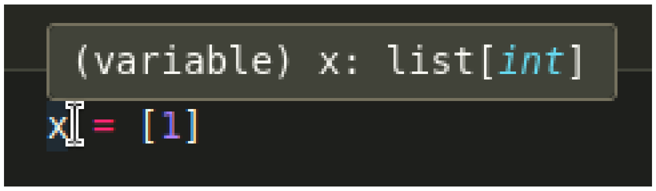
```
(variable) x: list[int]
x = [1]
```

With the type annotations:



```
(variable) x: list[int]
x = [1]
```

If you try to call the function with something other than an `int`, you will get a warning from the editor:



```
(variable) x: list[int]
x = [1]
```

You can also use type annotations for sequences, dictionaries, etc.:

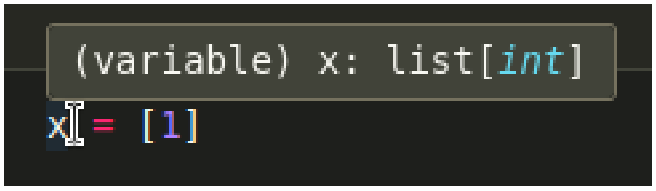
[320]:
```python
def build_query(
    table_name: str,
    column_names: list[str],
    config: dict[str, str],
) -> str:

    query = f"""\
    SELECT {", ".join(column_names)}
    FROM {config["database"]}.{table_name}
    LIMIT {config["limit"]}\
    """
    return query
```

`list[str]` means "a list of strings".

`dict[str, str]` means "a dictionary with string keys and string values".

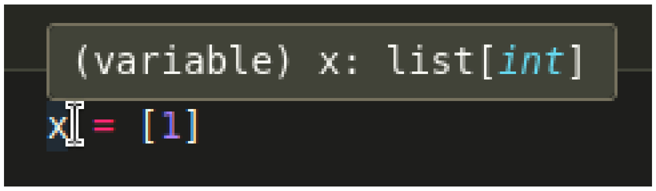
The syntax is `dict[KeyType, ValueType]`.

Now, when I want to call the function, I can hover over the function name to see what types the function expects:



```
(variable) x: list[int]
x = [1]
```

Without the type annotations, my only hope would be that the function is well-documented, or that I can read the function implementation to figure out what types it expects:



```
(variable) x: list[int]
x = [1]
```

5.0.2 5.2: Union types

If you want to allow multiple types for a variable, you can use a *union type*.

There are two ways to write a union type. The first is to use the `Union` class from the `typing` module.

If you are running Python 3.9 or older, this is the only way.

[ ]:
```python
from typing import Union

def sanitize_sku(sku: Union[int, str]) -> str:
    if isinstance(sku, int):
        sku = str(sku)

    if not sku.isdecimal():
        raise ValueError("SKU must be an integer number")

    if len(sku) != 8:
        raise ValueError("SKU must be 8 characters long")

    return sku
```

Think of it as an “or” between types - `sku` can *either* be an `int` *or* a `str`.

In Python 3.10, they introduced a new, additional syntax for union types, using the `|` (pipe) operator.

On a danish layout keyboard, you can find the `|` character by pressing `AltGr` plus the key to the right of the `+/?` key.

[ ]:
```python
def sanitize_sku(sku: int | str) -> str: ...
```

Using this syntax, you don’t have to import `Union` from the `typing` module, since the `|` operator is built into the language.

5.0.3 5.3: Optional types

Using a union is also useful when you want to allow `None` as a value, e.g. when a parameter requires a mutable value.

Recall our `add_user` function from earlier:

[ ]:
```python
def add_user(user, users=None):
    if users is None:
        users = []

    users.append(user)
    return users
```

Adding type annotations, we can see that `users` should *either* be a `list of str` *or* `None`:

[ ]:
```python
def add_user(user: str, users: list[str] | None = None) -> list[str]:
    ...
```

Since this is so common a pattern, Python has a special type alias for this: `Optional`.

Like `Union`, `Optional` is part of the `typing` module, and is used to specify that a variable can be of a certain type *or* `None`.

In essence, `Optional[str]` is equivalent to `Union[str, None]`.

Using `Optional`, we could rewrite the function annotations like this:

[ ]:
```python
from typing import Optional

def add_user(user: str, users: Optional[list[str]] = None) -> list[str]:
    ...
```

However, since both `Union` and `Optional` types can be expressed using the `|` operator in Python 3.10 and newer, I personally prefer using that syntax.

Just don't mix the `Union`/`Optional` syntax with the `|` syntax in the same file, as it can be confusing.

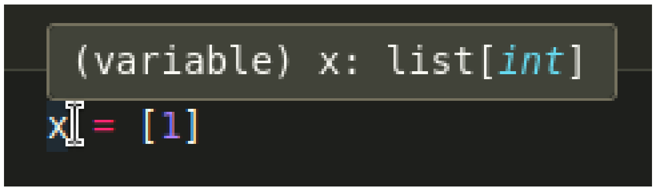
5.0.4 5.4: Type all the things?

Typing code correctly can help you a lot, but it can also become *very* complicated.

Typically, the more complicated and coupled your code is, more complicated type annotations are needed to get the *full* benefit.

However, that doesn't mean that there is no benefit to just annotating *some* things, so try it out and use it where it makes sense to you.

Once you start using type annotations, your editor might also start complaining about wrong or missing types:



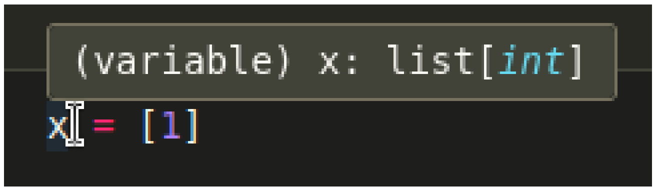
```
(variable) x: list[int]
x = [1]
```

This is because the `Python` extension in `VSCode` uses the type annotations to check your code for type errors.

It tries to be smart and *infer* the types of variables and function return values, but it can't always get it right, especially when dealing with more dynamic code.

In the above case, since `x` started out as a list containing an integer, the extension assumes that `x` will keep being a list of integers.

We can see that if we hover over the `x` variable:



```
(variable) x: list[int]
x = [1]
```

So when I go and append a string to `x`, the extension will complain.

Luckily, the type system has two espace hatches: the `Any` type, and the `# type: ignore` comment.

5.0.5 5.5: The "Any" type

The `Any` type is a special type that can be used to indicate that a variable can be of any type.

You need to first import `Any` from the `typing` module:

```
[ ]: from typing import Any
```

You can now use `Any` as a type annotation:

```
[ ]: x: list[Any] = [1]
     x.append("Hello")
```

5.0.6 5.6: The "type: ignore" comment

As a last resort, you can write `# type: ignore` as a *line comment* to tell the `Python` extension to ignore the type error on that line:

```
[ ]: x: list[int] = [1, 2, 3]
     x.append("Hello")  # type: ignore
```

The exact syntax for the comment is:

`statement<space><space>#<space>type:<space>ignore`

This tells the *type checker* to completely ignore any and all type errors on that entire line.

Please note that type annotations are completely optional, and they are **not** enforced by Python itself.

Instead, they are read and used by external tools, like the `Pylance` extension for `VSCode`.

To see this, let's look at the following code:

```
[ ]: def add(a: int, b: int) -> int:
         return a + b

     add("Hello ", "World!")
```

What happens when we run this code?

```
[ ]: add("Hello ", "World!")
```

```
[ ]: 'Hello World!'
```

This is one of the very powerful features of Python being a dynamically typed language.

You can change the type of a variable at any time, and use one type in place of another in function arguments

… as long as the operations you perform on the variable are valid for the new type.

```
[ ]: def add(a, b):
         return a + b

     print(add(10, 20))
     print(add(1.5, 2.6))
     print(add("Hello ", "World!"))
     print(add([1, 2], [3, 4]))
```

```
30
4.1
Hello World!
[1, 2, 3, 4]
```

This is also called *duck typing*:

If it walks like a duck and it quacks like a duck, then it must be a duck.

In other words, when Python expects an argument of a certain type, it doesn't care whether the value it gets is actually a string.

It only cares that the value can be used to do the things that the function expects to do with a string.

We will see more examples of this later when we get to Classes.

6 Chapter 6: Exceptions

Exceptions are events that occur during the execution of a program that disrupt the normal flow of instructions.

In Python, exceptions are used to handle errors gracefully and to provide meaningful error messages to the user.

6.1 6.1: What are Exceptions?

Exceptions are raised when an error occurs in a program.

For example, if you try to divide a number by zero, Python will raise a `ZeroDivisionError` exception.

If you try to access an index in a list that is out of range, Python will raise an `IndexError` exception.

Here are a few common exceptions you might encounter (and have probably already encountered) in Python:

- `SyntaxError`: Raised when there is an error in Python syntax.

[266]:
```
def func{}:
    pass
```

```
  Cell In[266], line 1
    def func{}:
            ^
SyntaxError: expected '('
```

- `IndentationError`: Raised when indentation is not specified properly.

[267]:
```
def func():
print("Hi ")
```

```
  Cell In[267], line 2
    print("Hi ")
    ^
IndentationError: expected an indented block after function definition on line
```

- `IndexError`: Raised when you try to access an index that is out of range.

[ ]:
```
names = ["Alice", "Bob", "Charlie"]
print(names[5])
```

```
---------------------------------------------------------------------------
IndexError                                Traceback (most recent call last)
Cell In[268], line 2
      1 names = ["Alice", "Bob", "Charlie"]
----> 2 print(names[5])

IndexError: list index out of range
```

- `KeyError`: Raised when you try to access a key that doesn't exist in a dictionary.

[ ]:
```
alice = {"name": "Alice", "age": 25}
print(alice["email"])
```

```
---------------------------------------------------------------------------
KeyError                                  Traceback (most recent call last)
Cell In[269], line 2
```

```
      1 alice = {"name": "Alice", "age": 25}
----> 2 print(alice["email"])

KeyError: 'email'
```

- `TypeError`: Raised when an operation or function is applied to an object of inappropriate type.

```
[ ]: int([1, 2])
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Cell In[270], line 1
----> 1 int([1, 2])

TypeError: int() argument must be a string, a bytes-like object or a real␣
 ↪number, not 'list'
```

- `ValueError`: Raised when a function receives an argument of the right type but inappropriate value.

```
[ ]: int("Hello")
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
Cell In[271], line 1
----> 1 int("Hello")

ValueError: invalid literal for int() with base 10: 'Hello'
```

- `ZeroDivisionError`: Raised when you try to divide by zero.
- `FileNotFoundError`: Raised when trying to open a file or directory that doesn't exist.
- `ModuleNotFoundError`: Raised when a module could not be found.
- `ImportError`: Raised when an import statement fails more generally.
- `RecursionError`: Raised when the maximum recursion depth is exceeded.
- `AssertionError`: Raised when an `assert` statement fails.
- `RuntimeError`: Raised when an error does not fall under any other category.

6.2 6.2: Handling Exceptions

You can handle exceptions in Python using the `try` and `except` blocks:

```
[ ]: def divide(x, y):
         try:
             print(x / y)
         except ZeroDivisionError:
             print("Division by zero!")

     divide(10, 2)
     divide(10, 0)
```

```
5.0
Division by zero!
```

The `try` block should contain the code that might raise an exception.

If an exception is raised while executing the code in the `try` block, the program will jump to the `except` block.

If the type of the exception that is raised matches the type specified in the `except` block, the code inside the `except` block will be executed.

Otherwise, the exception will be passed up to the next level of the program, and ultimately, "crash" the program if uncaught.

```
[ ]: divide(2, "Hello")
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Cell In[273], line 1
----> 1 divide(2, "Hello")

Cell In[272], line 3, in divide(x, y)
      1 def divide(x, y):
      2     try:
----> 3         print(x / y)
      4     except ZeroDivisionError:
      5         print("Division by zero!")

TypeError: unsupported operand type(s) for /: 'int' and 'str'
```

6.3 Exercise 16: Converting a string to an integer

6.4 6.3: Raising exceptions

You can raise exceptions yourself using the `raise` statement.

This is useful when you want to stop the execution of a function if a certain condition is met:

```
[ ]: def create_user(name: str, age: int) -> dict:
         if age < 18:
             raise ValueError("Users must be 18 or older")
```

```
    return {"name": name, "age": age}

print(create_user("Alice", 16))
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
Cell In[323], line 7
      3         raise ValueError("Users must be 18 or older")
      4     return {"name": name, "age": age}
----> 7 print(create_user("Alice", 16))

Cell In[323], line 3, in create_user(name, age)
      1 def create_user(name: str, age: int) -> dict:
      2     if age < 18:
----> 3         raise ValueError("Users must be 18 or older")
      4     return {"name": name, "age": age}

ValueError: Users must be 18 or older
```

Try-except is very useful when you want to handle exceptions and errors gracefully.

An example could be when querying from a database or an API, and the connection times out. Instead of crashing the program, you can catch the exception and try again later.

6.5 6.4: Cleanup with "finally"

The `try` statement has another optional `finally` clause which is intended to define clean-up actions that must be executed under all circumstances.

```
[ ]: try:
         1 / 0
     finally:
         print("Oh no, bad maths!")
```

```
Oh no, bad maths!
```

```
---------------------------------------------------------------------------
ZeroDivisionError                         Traceback (most recent call last)
Cell In[275], line 2
      1 try:
----> 2     1 / 0
      3 finally:
      4     print("Oh no, bad maths!")

ZeroDivisionError: division by zero
```

If a `finally` clause is present, the `finally` clause will execute as the last task before the `try` statement completes.

The `finally` clause runs whether or not the `try` statement produces an exception, and whether or not that exception is then handled.

In other words, the `finally` clause is executed **no matter what**.

In reality, the `finally` clause is often used to clean up external resources, like closing a file or a database connection:

[ ]:
```
file = open("myfile.txt")
try:
    contents = file.read()
    print(contents)
finally:
    file.close()
```

If not for the `finally` clause, the file would stay open and blocked as long as the program was running.

The same issue can happen with database connections, where they are opened when needed but never explicitly closed by the user, leading to a "connection leak".

This `try -> use resource -> finally -> release resource` pattern is so common that Python has a special syntax for it:

The `with` statement.

Some objects define standard clean-up actions to be undertaken when the object is no longer needed, regardless of whether or not the operation using the object succeeded or failed.

The `with` statement allows objects like files to be used in a way that ensures they are always cleaned up promptly and correctly:

[ ]:
```
with open("myfile.txt") as my_file:
    contents = my_file.read()
    print(contents)
```

After the statement is executed, the file `my_file` is always closed, even if a problem was encountered while processing the lines.

If an object can be used with the `with` statement, it is called a *context manager*.

Objects which, like files, provide predefined clean-up actions will indicate this in their documentation.

7 Chapter 7: Modules

[ ]:
```
## Importing Modules

# Importing the math module
import math
```

```
# Using a function from the math module
result = math.sqrt(16)
print(result)  # 4.0

# Importing a specific function from a module
from math import factorial

# Using the imported function
result = factorial(5)
print(result)  # 120

# Importing a module with an alias
import numpy as np

# Using the alias to call a function from the module
array = np.array([1, 2, 3, 4, 5])
print(array)  # [1 2 3 4 5]

# Importing multiple functions from a module
from math import cos, sin, tan

# Using the imported functions
angle = math.radians(45)
print(sin(angle))  # 0.7071067811865475
print(cos(angle))  # 0.7071067811865476
print(tan(angle))  # 0.9999999999999999
```

[ ]:
```
## Exercises

# Exercise 1: Import the random module and use it to generate a random number␣
 ↪between 1 and 10
import random

random_number = random.randint(1, 10)
print(random_number)
```

[ ]:
```
# Exercise 2: Import the datetime module and use it to print the current date␣
 ↪and time
import datetime

current_datetime = datetime.datetime.now()
print(current_datetime)
```

[ ]:
```
# Exercise 3: Import the os module and use it to print the current working␣
 ↪directory
import os

```

```
current_directory = os.getcwd()
print(current_directory)
```

[ ]:
```
# Exercise 4: Import the statistics module and use it to calculate the mean of␣
↪a list of numbers
import statistics

numbers = [1, 2, 3, 4, 5]
mean = statistics.mean(numbers)
print(mean)
```

[ ]:
```
# Exercise 5: Import the sys module and use it to print the Python version
import sys

python_version = sys.version
print(python_version)
```

8 Chapter 8: Namespaces and scopes

There's a catch that often trip beginners up: *namespaces* and the *scope* of variables.

… which is completely fair, since it's a bit of a tricky concept.

[ ]:
```
number = 10


def func():
    number = 20


func()
```

What is the value of `number` after calling `func()`?

[ ]:
```
print(number)
```

The parameters of a function (e.g. the `name` parameter in the `greet` function) are variables that are local to that function - they only exist within the function.

The same goes for any additional variables that are created within the function (in this case, the `number` variable).

This is true *even if the variable has the same name* as a variable outside the function.

[ ]:
```
try:
    del x, y
except NameError:
    pass
```

```
[ ]: number = 10


     def double(x):
         """Return the input multiplied by 2."""
         y = x * 2
         return y


     result = double(number)
     print(x)
```

Let's look at what happened when we ran the above code:

`number = 10`

-> Variable `number` created in global scope:

Global
number = 10

`result = double(number)`

-> We enter the `double(x)` function, and a new, local scope is created.

-> Since we input `10` as the argument, the new, local variable `x` is created with the value `10`:

Local (`double`)
x = 10

`y = x * 2`

-> Python looks for a variable called `x`. It looks in the local scope first, and finds it.

-> It then performs the calculation `x * 2`, and saves the result in a new local variable `y`:

Local (`double`)
x = 10
y = 20

`return y`

-> Python looks for a variable called `y`, and finds it in the local scope. It returns the value (`20`), and the local scope is destroyed when the function is exited.

-> The value `20` is stored in a new variable `result` in the global scope:

Global

number = 10
result = 20

When *referencing* a variable, Python will first look in the local scope, then in any enclosing scope(s), then in the global scope, and finally in the built-in scope.

That means that we can actually reference variables in a function that were created outside of the function:

[ ]:
```
greeting = "Hello"


def greet(name):
    """Print a greeting for the input name."""
    print(f"{greeting} {name}!")


greet("Alice")
```

In this case, `name` is a variable in the scope that is local to the `greet` function, while `greeting` is a variable in the global scope.

On the contrary, when *assigning* a value to a variable, Python will always create that variable in the local scope.

This means that you can have a variable with the same name in the global scope and in a function, and they will be two separate variables. This is called *shadowing*: the local variable *shadows* the global variable.

[ ]:
```
x = 10


def add(x):
    return x + 5  # This x is the local variable, *not* the global one


add(20)
```

9 Chapter 9: Classes

[ ]:
```
## Creating a Class
class Person:
    def __init__(self, name, age):
        self.name = name
        self.age = age
```

```
# Creating an instance of the class
person1 = Person("Alice", 25)
print(person1.name)  # Alice
print(person1.age)  # 25

## Instantiating a Class
person2 = Person("Bob", 30)
print(person2.name)  # Bob
print(person2.age)  # 30



## Calling Method on Instance
class Person:
    def __init__(self, name, age):
        self.name = name
        self.age = age

    def greet(self):
        return f"Hello, my name is {self.name} and I am {self.age} years old."


person3 = Person("Charlie", 35)
print(person3.greet())  # Hello, my name is Charlie and I am 35 years old.

## Access Attribute on Instance
print(person3.name)  # Charlie
print(person3.age)  # 35
```

```
[ ]: ## Exercises



# Exercise 1: Create a class called `Car` with attributes `make`, `model`, and␣
↪`year`. Create an instance of the class and print the attributes.
class Car:
    def __init__(self, make, model, year):
        self.make = make
        self.model = model
        self.year = year



car1 = Car("Toyota", "Corolla", 2020)
print(car1.make)  # Toyota
print(car1.model)  # Corolla
print(car1.year)  # 2020
```

```
[ ]: # Exercise 2: Add a method to the `Car` class that returns a string with the␣
↪car's details. Call the method and print the result.
```

```
class Car:
    def __init__(self, make, model, year):
        self.make = make
        self.model = model
        self.year = year

    def car_details(self):
        return f"{self.year} {self.make} {self.model}"


car2 = Car("Honda", "Civic", 2018)
print(car2.car_details())  # 2018 Honda Civic
```

[ ]:
```
# Exercise 3: Create a class called `Book` with attributes `title`, `author`,␣
↪and `pages`. Create an instance of the class and print the attributes.
class Book:
    def __init__(self, title, author, pages):
        self.title = title
        self.author = author
        self.pages = pages


book1 = Book("1984", "George Orwell", 328)
print(book1.title)  # 1984
print(book1.author)  # George Orwell
print(book1.pages)  # 328
```

[ ]:
```
# Exercise 4: Add a method to the `Book` class that returns a string with the␣
↪book's details. Call the method and print the result.
class Book:
    def __init__(self, title, author, pages):
        self.title = title
        self.author = author
        self.pages = pages

    def book_details(self):
        return f"'{self.title}' by {self.author}, {self.pages} pages"


book2 = Book("To Kill a Mockingbird", "Harper Lee", 281)
print(book2.book_details())  # 'To Kill a Mockingbird' by Harper Lee, 281 pages
```

[ ]:
```
# Exercise 5: Create a class called `Student` with attributes `name`,␣
↪`student_id`, and `grades` (a list of grades). Create an instance of the␣
↪class and print the attributes.
class Student:
    def __init__(self, name, student_id, grades):
        self.name = name
        self.student_id = student_id
        self.grades = grades


student1 = Student("David", "S12345", [90, 85, 88])
print(student1.name)  # David
print(student1.student_id)  # S12345
print(student1.grades)  # [90, 85, 88]
```

[ ]:
```
# Exercise 6: Add a method to the `Student` class that calculates and returns␣
↪the average grade. Call the method and print the result.
class Student:
    def __init__(self, name, student_id, grades):
        self.name = name
        self.student_id = student_id
        self.grades = grades

    def average_grade(self):
        return sum(self.grades) / len(self.grades)


student2 = Student("Emma", "S67890", [92, 87, 85])
print(student2.average_grade())  # 88.0
```