Because tuples are immutable, there are only two built-in tuple methods, `count()` and `index()`. The `count()` method returns the number of times its argument occurs in the tuple. For instance,

```
t = (-7, 0, 7, -7, 0, 0)
t.count(-7)     # => 2
```

The `index()` method returns the index of the first occurrence of its argument. For instance,

```
t = ("foo", "bar", "baz", "foo", "bar", "baz", "baz")
t.index("baz")  # => 2
```

The **range** built-in type is a compact way or representing sequences of integers. A `range` can be constructed with the `range(start, stop, step)` constructor function, as in the following examples:

```
list(range(0, 3, 1))    # => [0, 1, 2]
list(range(2, 6, 1))    # => [2, 3, 4, 5]
list(range(0, 3))       # => [0, 1, 2] (step=1 by default)
list(range(3))          # => [0, 1, 2] (start=0 by default)
```

Note that we have wrapped the ranges in `list()` functions, which converted each range to a list. This was only so we can see the values it represents; alone, an expression like `range(0, 3)` returns itself. This is why a range is such a compact data point—all that needs to be stored in memory are the `start`, `stop`, and `step` arguments because the intermediate values are implicit.

1.8 Dictionaries

OW
v1

The built-in Python **dictionary** class `dict` is an unordered collection of elements, each of which has a unique **key** and a **value**. A key can be any immutable object, but a string is most common. A value can be any object. The basic syntax to create a `dict` object with keys `k`x and values `v`x is `{k1: v1, k2: v2, ...}`. For instance, we can define a `dict` as follows:

```
d = {"foo": 5, "bar": 1, "baz": -3}
```

Accessing a value requires its key. To access a value in dictionary `d` with key `k`, use the syntax `d[k]`. For example,

```
d = { # It is often useful to break lines at each key-value pair
    "name": "Spiff",
    "age": 33,
    "occupation": "spaceman",
    "enemies": ["Zorgs", "Zargs", "Zogs"]
}
print(f"{d['name']} is a {d['age']} year old"
      f"{d['occupation']} who fights {d['enemies'][0]}.")
```

This returns

```
Spiff is a 33 year old spaceman who fights Zorgs.
```

A value `v` with key `k` can be added to an existing dictionary `d` with the syntax `d[k] = v`. For instance, (Filik et al. 2019)

```
d = {} # Empty dictionary
d["irony"] = "The use of a word to mean its opposite."
d["sarcasm"] = "Irony intended to criticize."
```

Dictionaries are mutable; therefore, we can change their contents, as in the following example:

```
d = {}
d["age"] = 33   # d is {"age": 33}
d["age"] = 31   # d is {"age": 31}
```

Dictionaries have several handy methods; these are listed in table 1.7.

Note that most of these methods apply to dictionary instance `d`, either mutating `d` or returning something from `d`. However, the `fromkeys()` method is called from

Table 1.7:

Methods	Descriptions
`d.clear()`	Clears all items from `d`
`d.copy()`	Returns a shallow copy of `d`
`dict.fromkeys(s[, v])`	Returns a new `dict` with keys from sequence `s`, each with optional
`d.get(k)`	Returns the value for key `k` in `d`
`d.items()`	Returns a view object of key-value pairs in `d`
`d.keys()`	Returns a view object of keys in `d`
`d.pop(k)`	Removes and returns the value for key `k` in `d`
`d.popitem()`	Removes and returns the last-inserted key-value pair from `d`
`d.setdefault(k, v)`	Returns the value for the key `k` in `d`; inserts `v` if absent
`d.update(d_)`	Updates `d` with key-value pairs from another dictionary `d_`
`d.values()`	Returns a view object of values in `d`

the class dict because it has nothing to do with an instance. Such methods are called **class methods**; the other methods we've considered thus far are **instance methods**.

Dictionary **view objects**—returned by items(), keys(), and values()—are dynamically updating objects that change with their dictionary. For instance,

```
d = {"a": 1, "b": 2}
d_keys = d.keys()
print(f"View object before: {d_keys}")
d["c"] = 3
print(f"View object after: {d_keys}")
```

This returns

```
View object before: dict_keys(['a', 'b'])
View object after: dict_keys(['a', 'b', 'c'])
```

View objects can be converted to lists with the list() function, as in list(d_keys).

Example 1.4

Write a program that meets the following requirements:

1. It defines a list of strings names = ["Mo", "Jo", "Flo"]
2. It constructs a dict instance data with keys from the list names
3. It creates and populates a sub-dict with the follow properties for each name:
   a. Mo—year: sophomore, major: Mechanical Engineering, GPA: 3.44
   b. Jo—year: junior, major: Computer Science, GPA: 3.96
   c. Flo—year: sophomore, major: Philosophy, GPA: 3.12
4. It prints each of the students' name and year
5. It replaces Jo's GPA with 3.98 and prints this new value
6. It removes the entry for Mo and prints a list of remaining keys in data

The following program meets the given requirements:

```
names = ["Mo", "Jo", "Flo"]
data = dict.fromkeys(names) # => {"Mo": None, "Jo": None, "Flo": None}

#%% Populate Data
data["Mo"] = {}
data["Mo"]["year"] = "sophomore"
data["Mo"]["major"] = "Mechanical Engineering"
data["Mo"]["GPA"] = 3.44
data["Jo"] = {}
data["Jo"]["year"] = "junior"
data["Jo"]["major"] = "Computer Science"
data["Jo"]["GPA"] = 3.96
data["Flo"] = {}
data["Flo"]["year"] = "sophomore"
data["Flo"]["major"] = "Philosophy"
data["Flo"]["GPA"] = 3.12

#%% Data Operations and Printing
print(f"Mo is a {data['Mo']['year']}. "
      f"Jo is a {data['Jo']['year']}. "
      f"Flo is a {data['Flo']['year']}.")
data["Jo"]["GPA"] = 3.98
print(f"Jo's new GPA is {data['Jo']['GPA']}")
data.pop("Mo")
print(f"Names sans Mo: {list(data.keys())}")
```

This prints the following in the console:

```
Mo is a sophomore. Jo is a junior. Flo is a sophomore.
Jo's new GPA is 3.98
Names sans Mo: ['Jo', 'Flo']
```

1.9 Functions

V3
v1

In Python, **functions** are reusable blocks of code that accept input arguments and return one or more values. As we have seen, a method is a special type of function that is contained within an object. We typically do not refer to methods as “functions,” instead reserving the term for functions that are not methods. A function that computes the square root of the sum of the squares of two arguments can be defined as:

```
def root_sum_squared(arg1, arg2):
    sum_squared = arg1**2 + arg2**2
    return sum_squared**(1/2)
```

The syntax requires the block of code following the `def` line to be indented. A block ends where the indent ends. The indent should, by convention, be 4 space characters. The function ends with a **return statement**, which begins with the keyword `return` followed by an expression, the value of which is returned to the caller code. The variable `sum_squared` is created inside the function, so it is local to the function and cannot be accessed from outside. **Calling** (using) this function could look like

```
root_sum_squared(3, 4)
```

This call returns the value `5.0`.

The arguments `arg1` and `arg2` in the previous example are called **positional arguments** because they are identified in the function call by their position; that is, `3` is identified as `arg1` and `4` is identified as `arg2` based on their positions in the argument list. There is another type of argument, called a **keyword argument** (sometimes called a “named” argument), that can follow positional arguments and have the syntax `<key>=<value>`. For instance, we could augment the previous function as follows:

```
def root_sum_squared(arg1, arg2, pre="RSS ="):
    sum_squared = arg1**2 + arg2**2
    rss = sum_squared**(1/2)
    print(pre, rss)
    return rss
```

The `pre` positional argument is given a default value of `"RSS ="`, and the function now prints the root sum square with `pre` prepended. Calling this function with

```
sum_squared(4, 6)
```

prints the following to the console:

```
RSS = 7.211102550927978
```

Alternatively, we could pass a value to `pre` with the call