Chapter 2. Native Datatypes

❝ *Wonder is the foundation of all philosophy, inquiry its progress, ignorance its end.* ❞
— *Michel de Montaigne*

Diving In

Datatypes. Set aside your first Python program for just a minute, and let's talk about datatypes. In Python, every value has a datatype, but you don't need to declare the datatype of variables. How does that work? Based on each variable's original assignment, Python figures out what type it is and keeps tracks of that internally.

Python has many native datatypes. Here are the important ones:

1. **Booleans** are either `True` or `False`.
2. **Numbers** can be integers (`1` and `2`), floats (`1.1` and `1.2`), fractions (`1/2` and `2/3`), or even complex numbers.
3. **Strings** are sequences of Unicode characters, *e.g.* an HTML document.
4. **Bytes** and **byte arrays**, *e.g.* a JPEG image file.
5. **Lists** are ordered sequences of values.
6. **Tuples** are ordered, immutable sequences of values.
7. **Sets** are unordered bags of values.
8. **Dictionaries** are unordered bags of key-value pairs.

Of course, there are more types than these. Everything is an object in Python, so there are types like *module*, *function*, *class*, *method*, *file*, and even *compiled code*. You've already seen some of these: modules have names, functions have `docstrings`, &c. You'll learn about classes in Classes & Iterators, and about files in Files.

Strings and bytes are important enough — and complicated enough — that they get their own chapter. Let's look at the others first.

⁂

2.2. Booleans

Booleans are either true or false. Python has two constants, cleverly named `True` and `False`, which can be used to assign boolean values directly. Expressions can also evaluate to a boolean value. In certain places (like `if` statements), Python expects an expression to evaluate to a boolean value. These places are called *boolean contexts*. You can use virtually any expression in a boolean context, and Python will try to determine its truth value. Different datatypes have different rules about which values are true or false in a boolean context. (This will make more sense once you see some concrete examples later in this chapter.)

For example, take this snippet from humansize.py:

```
if size < 0:
    raise ValueError('number must be non-negative')
```

size is an integer, 0 is an integer, and < is a numerical operator. The result of the expression `size < 0` is always a boolean. You can test this yourself in the Python interactive shell:

You can use virtually any expression in a boolean context.

```
>>> size = 1
>>> size < 0
False
>>> size = 0
>>> size < 0
False
>>> size = -1
>>> size < 0
True
```

Due to some legacy issues left over from Python 2, booleans can be treated as numbers. `True` is 1; `False` is 0.

```
>>> True + True
2
>>> True - False
1
>>> True * False
0
>>> True / False
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ZeroDivisionError: int division or modulo by zero
```

Ew, ew, ew! Don't do that. Forget I even mentioned it.

⁂

2.3. Numbers

Numbers are awesome. There are so many to choose from. Python supports both integers and floating point numbers. There's no type declaration to distinguish them; Python tells them apart by the presence or absence of a decimal point.

```
>>> type(1) ①
<class 'int'>
>>> isinstance(1, int) ②
True
>>> 1 + 1 ③
2
>>> 1 + 1.0 ④
2.0
>>> type(2.0)
<class 'float'>
```

① You can use the `type()` function to check the type of any value or variable. As you might expect, `1` is an `int`.

② Similarly, you can use the `isinstance()` function to check whether a value or variable is of a given type.

③ Adding an `int` to an `int` yields an `int`.

④ Adding an `int` to a `float` yields a `float`. Python coerces the `int` into a `float` to perform the addition, then returns a `float` as the result.

2.3.1. Coercing Integers To Floats And Vice-Versa

As you just saw, some operators (like addition) will coerce integers to floating point numbers as needed. You can also coerce them by yourself.

```
>>> float(2) ①
2.0
>>> int(2.0) ②
2
>>> int(2.5) ③
2
>>> int(-2.5) ④
-2
>>> 1.12345678901234567890 ⑤
1.1234567890123457
>>> type(1000000000000000) ⑥
<class 'int'>
```

① You can explicitly coerce an `int` to a `float` by calling the `float()` function.

② Unsurprisingly, you can also coerce a `float` to an `int` by calling `int()`.

③ The `int()` function will truncate, not round.

④ The `int()` function truncates negative numbers towards 0. It's a true truncate function, not a floor function.

⑤ Floating point numbers are accurate to 15 decimal places.

⑥ Integers can be arbitrarily large.

☞ Python 2 had separate types for `int` and `long`. The `int` datatype was limited by `sys.maxint`, which varied by platform but was usually 2^{32}`-1`. Python 3 has just one integer type, which behaves mostly like the old `long` type from Python 2. See PEP 237 for details.

2.3.2. Common Numerical Operations

You can do all kinds of things with numbers.

```
>>> 11 / 2 ①
5.5
>>> 11 // 2 ②
5
>>> -11 // 2 ③
-6
>>> 11.0 // 2 ④
5.0
>>> 11 ** 2 ⑤
121
>>> 11 % 2 ⑥
1
```

① The `/` operator performs floating point division. It returns a `float` even if both the numerator and denominator are `int`s.
② The `//` operator performs a quirky kind of integer division. When the result is positive, you can think of it as truncating (not rounding) to 0 decimal places, but be careful with that.
③ When integer-dividing negative numbers, the `//` operator rounds "up" to the nearest integer. Mathematically speaking, it's rounding "down" since `-6` is less than `-5`, but it could trip you up if you were expecting it to truncate to `-5`.
④ The `//` operator doesn't always return an integer. If either the numerator or denominator is a `float`, it will still round to the nearest integer, but the actual return value will be a `float`.
⑤ The `**` operator means "raised to the power of." 11^2 is `121`.
⑥ The `%` operator gives the remainder after performing integer division. `11` divided by `2` is `5` with a remainder of `1`, so the result here is `1`.

☞ In Python 2, the `/` operator usually meant integer division, but you could make it behave like floating point division by including a special directive in your code. In Python 3, the `/` operator always means floating point division. See PEP 238 for details.

2.3.3. Fractions

Python isn't limited to integers and floating point numbers. It can also do all the fancy math you learned in high school and promptly forgot about.

```
>>> import fractions ①
>>> x = fractions.Fraction(1, 3) ②
>>> x
Fraction(1, 3)
>>> x * 2 ③
Fraction(2, 3)
>>> fractions.Fraction(6, 4) ④
Fraction(3, 2)
>>> fractions.Fraction(0, 0) ⑤
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "fractions.py", line 96, in __new__
    raise ZeroDivisionError('Fraction(%s, 0)' % numerator)
ZeroDivisionError: Fraction(0, 0)
```

① To start using fractions, import the `fractions` module.

② To define a fraction, create a `Fraction` object and pass in the numerator and denominator.

③ You can perform all the usual mathematical operations with fractions. Operations return a new `Fraction` object. `2 * (1/3) = (2/3)`

④ The `Fraction` object will automatically reduce fractions. `(6/4) = (3/2)`

⑤ Python has the good sense not to create a fraction with a zero denominator.

2.3.4. Trigonometry

You can also do basic trigonometry in Python.

```
>>> import math
>>> math.pi ①
3.1415926535897931
>>> math.sin(math.pi / 2) ②
1.0
>>> math.tan(math.pi / 4) ③
0.99999999999999989
```

① The `math` module has a constant for π, the ratio of a circle's circumference to its diameter.

② The `math` module has all the basic trigonometric functions, including `sin()`, `cos()`, `tan()`, and variants like `asin()`.

③ Note, however, that Python does not have infinite precision. `tan(π / 4)` should return `1.0`, not `0.99999999999999989`.

2.3.5. Numbers In A Boolean Context

You can use numbers in a boolean context, such as an `if` statement. Zero values are false, and non-zero values are true.

```
>>> def is_it_true(anything): ①
...   if anything:
...     print("yes, it's true")
...   else:
...     print("no, it's false")
...
>>> is_it_true(1) ②
yes, it's true
>>> is_it_true(-1)
yes, it's true
>>> is_it_true(0)
no, it's false
>>> is_it_true(0.1) ③
yes, it's true
>>> is_it_true(0.0)
no, it's false
>>> import fractions
>>> is_it_true(fractions.Fraction(1, 2)) ④
yes, it's true
>>> is_it_true(fractions.Fraction(0, 1))
no, it's false
```

Zero values are false, and non-zero values are true.

① Did you know you can define your own functions in the Python interactive shell? Just press `ENTER` at the end of each line, and `ENTER` on a blank line to finish.

② In a boolean context, non-zero integers are true; 0 is false.

③ Non-zero floating point numbers are true; `0.0` is false. Be careful with this one! If there's the slightest rounding error (not impossible, as you saw in the previous section) then Python will be testing `0.0000000000001` instead of 0 and will return `True`.

④ Fractions can also be used in a boolean context. `Fraction(0, n)` is false for all values of `n`. All other fractions are true.

⁂

2.4. Lists

Lists are Python's workhorse datatype. When I say "list," you might be thinking "array whose size I have to declare in advance, that can only contain items of the same type, &c." Don't think that. Lists are much cooler than that.

☞ A list in Python is like an array in Perl 5. In Perl 5, variables that store arrays always start with the `@` character; in Python, variables can be named anything, and Python keeps track of the datatype internally.

☞ A list in Python is much more than an array in Java (although it can be used as one if that's really all you want out of life). A better analogy would be to the `ArrayList` class, which can hold arbitrary objects and can expand dynamically as new items are added.

2.4.1. Creating A List

Creating a list is easy: use square brackets to wrap a comma-separated list of values.

```
>>> a_list = ['a', 'b', 'mpilgrim', 'z', 'example'] ①
>>> a_list
['a', 'b', 'mpilgrim', 'z', 'example']
>>> a_list[0] ②
'a'
>>> a_list[4] ③
'example'
>>> a_list[-1] ④
'example'
>>> a_list[-3] ⑤
'mpilgrim'
```

① First, you define a list of five items. Note that they retain their original order. This is not an accident. A list is an ordered set of items.
② A list can be used like a zero-based array. The first item of any non-empty list is always `a_list[0]`.
③ The last item of this five-item list is `a_list[4]`, because lists are always zero-based.
④ A negative index accesses items from the end of the list counting backwards. The last item of any non-empty list is always `a_list[-1]`.
⑤ If the negative index is confusing to you, think of it this way: `a_list[-n] == a_list[len(a_list) - n]`. So in this list, `a_list[-3] == a_list[5 - 3] == a_list[2]`.

2.4.2. Slicing A List

Once you've defined a list, you can get any part of it as a new list. This is called *slicing* the list.

a_list[0] is the first item of a_list.

```
>>> a_list
['a', 'b', 'mpilgrim', 'z', 'example']
>>> a_list[1:3] ①
['b', 'mpilgrim']
>>> a_list[1:-1] ②
['b', 'mpilgrim', 'z']
>>> a_list[0:3] ③
['a', 'b', 'mpilgrim']
>>> a_list[:3] ④
['a', 'b', 'mpilgrim']
>>> a_list[3:] ⑤
['z', 'example']
```

```
>>> a_list[:] ⑥
['a', 'b', 'mpilgrim', 'z', 'example']
```

① You can get a part of a list, called a "slice", by specifying two indices. The return value is a new list containing all the items of the list, in order, starting with the first slice index (in this case `a_list[1]`), up to but not including the second slice index (in this case `a_list[3]`).

② Slicing works if one or both of the slice indices is negative. If it helps, you can think of it this way: reading the list from left to right, the first slice index specifies the first item you want, and the second slice index specifies the first item you don't want. The return value is everything in between.

③ Lists are zero-based, so `a_list[0:3]` returns the first three items of the list, starting at `a_list[0]`, up to but not including `a_list[3]`.

④ If the left slice index is 0, you can leave it out, and 0 is implied. So `a_list[:3]` is the same as `a_list[0:3]`, because the starting 0 is implied.

⑤ Similarly, if the right slice index is the length of the list, you can leave it out. So `a_list[3:]` is the same as `a_list[3:5]`, because this list has five items. There is a pleasing symmetry here. In this five-item list, `a_list[:3]` returns the first 3 items, and `a_list[3:]` returns the last two items. In fact, `a_list[:n]` will always return the first `n` items, and `a_list[n:]` will return the rest, regardless of the length of the list.

⑥ If both slice indices are left out, all items of the list are included. But this is not the same as the original `a_list` variable. It is a new list that happens to have all the same items. `a_list[:]` is shorthand for making a complete copy of a list.

2.4.3. Adding Items To A List

There are four ways to add items to a list.

```
>>> a_list = ['a']
>>> a_list = a_list + [2.0, 3] ①
>>> a_list ②
['a', 2.0, 3]
>>> a_list.append(True) ③
>>> a_list
['a', 2.0, 3, True]
>>> a_list.extend(['four', 'Ω']) ④
>>> a_list
['a', 2.0, 3, True, 'four', 'Ω']
>>> a_list.insert(0, 'Ω') ⑤
>>> a_list
['Ω', 'a', 2.0, 3, True, 'four', 'Ω']
```

① The + operator concatenates lists to create a new list. A list can contain any number of items; there is no size limit (other than available memory). However, if memory is a concern, you should be aware that list concatenation creates a second list in memory. In this case, that new list is immediately assigned to the existing variable `a_list`. So this line of code is really a two-step process — concatenation then assignment — which can (temporarily) consume a lot of memory when you're dealing with large lists.

② A list can contain items of any datatype, and the items in a single list don't all need to be the same type. Here we have a list containing a string, a floating point number, and an integer.

③ The `append()` method adds a single item to the end of the list. (Now we have *four* different datatypes in the list!)

④ Lists are implemented as classes. “Creating” a list is really instantiating a class. As such, a list has methods that operate on it. The `extend()` method takes one argument, a list, and appends each of the items of the argument to the original list.

⑤ The `insert()` method inserts a single item into a list. The first argument is the index of the first item in the list that will get bumped out of position. List items do not need to be unique; for example, there are now two separate items with the value `'Ω'`: the first item, `a_list[0]`, and the last item, `a_list[6]`.

☞ `a_list.insert(0, value)` is like the `unshift()` function in Perl. It adds an item to the beginning of the list, and all the other items have their positional index bumped up to make room.

Let’s look closer at the difference between `append()` and `extend()`.

```
>>> a_list = ['a', 'b', 'c']
>>> a_list.extend(['d', 'e', 'f'])  ①
>>> a_list
['a', 'b', 'c', 'd', 'e', 'f']
>>> len(a_list)  ②
6
>>> a_list[-1]
'f'
>>> a_list.append(['g', 'h', 'i'])  ③
>>> a_list
['a', 'b', 'c', 'd', 'e', 'f', ['g', 'h', 'i']]
>>> len(a_list)  ④
7
>>> a_list[-1]
['g', 'h', 'i']
```

① The `extend()` method takes a single argument, which is always a list, and adds each of the items of that list to `a_list`.

② If you start with a list of three items and extend it with a list of another three items, you end up with a list of six items.

③ On the other hand, the `append()` method takes a single argument, which can be any datatype. Here, you’re calling the `append()` method with a list of three items.

④ If you start with a list of six items and append a list onto it, you end up with… a list of seven items. Why seven? Because the last item (which you just appended) *is itself a list*. Lists can contain any type of data, including other lists. That may be what you want, or it may not. But it’s what you asked for, and it’s what you got.

2.4.4. Searching For Values In A List

```
>>> a_list = ['a', 'b', 'new', 'mpilgrim', 'new']
>>> a_list.count('new')  ①
2
```

```
>>> 'new' in a_list ②
True
>>> 'c' in a_list
False
>>> a_list.index('mpilgrim') ③
3
>>> a_list.index('new') ④
2
>>> a_list.index('c') ⑤
Traceback (innermost last):
  File "<interactive input>", line 1, in ?
ValueError: list.index(x): x not in list
```

① As you might expect, the `count()` method returns the number of occurrences of a specific value in a list.

② If all you want to know is whether a value is in the list or not, the `in` operator is slightly faster than using the `count()` method. The `in` operator always returns `True` or `False`; it will not tell you how many times the value appears in the list.

③ Neither the `in` operator nor the `count()` method will tell you *where* in the list a value appears. If you need to know where in the list a value is, call the `index()` method. By default it will search the entire list, although you can specify an optional second argument of the (0-based) index to start from, and even an optional third argument of the (0-based) index to stop searching.

④ The `index()` method finds the *first* occurrence of a value in the list. In this case, `'new'` occurs twice in the list, in `a_list[2]` and `a_list[4]`, but the `index()` method will return only the index of the first occurrence.

⑤ As you might *not* expect, if the value is not found in the list, the `index()` method will raise an exception.

Wait, what? That's right: the `index()` method raises an exception if it doesn't find the value in the list. This is notably different from most languages, which will return some invalid index (like `-1`). While this may seem annoying at first, I think you will come to appreciate it. It means your program will crash at the source of the problem instead of failing strangely and silently later. Remember, `-1` is a valid list index. If the `index()` method returned `-1`, that could lead to some not-so-fun debugging sessions!

2.4.5. Removing Items From A List

Lists can expand and contract automatically. You've seen the expansion part.
There are several different ways to remove items from a list as well.

Lists never have gaps.

```
>>> a_list = ['a', 'b', 'new', 'mpilgrim', 'new']
>>> a_list[1]
'b'
>>> del a_list[1] ①
>>> a_list
['a', 'new', 'mpilgrim', 'new']
>>> a_list[1] ②
'new'
```

① You can use the `del` statement to delete a specific item from a list.

② Accessing index `1` after deleting index `1` does *not* result in an error. All items after the deleted item shift their positional index to "fill the gap" created by deleting the item.

Don't know the positional index? Not a problem; you can remove items by value instead.

```
>>> a_list.remove('new') ①
>>> a_list
['a', 'mpilgrim', 'new']
>>> a_list.remove('new') ②
>>> a_list
['a', 'mpilgrim']
>>> a_list.remove('new')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: list.remove(x): x not in list
```

① You can also remove an item from a list with the `remove()` method. The `remove()` method takes a *value* and removes the first occurrence of that value from the list. Again, all items after the deleted item will have their positional indices bumped down to "fill the gap." Lists never have gaps.

② You can call the `remove()` method as often as you like, but it will raise an exception if you try to remove a value that isn't in the list.

2.4.6. Removing Items From A List: Bonus Round

Another interesting list method is `pop()`. The `pop()` method is yet another way to remove items from a list, but with a twist.

```
>>> a_list = ['a', 'b', 'new', 'mpilgrim']
>>> a_list.pop() ①
'mpilgrim'
>>> a_list
['a', 'b', 'new']
>>> a_list.pop(1) ②
'b'
>>> a_list
['a', 'new']
>>> a_list.pop()
'new'
>>> a_list.pop()
'a'
>>> a_list.pop() ③
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: pop from empty list
```

① When called without arguments, the `pop()` list method removes the last item in the list *and returns the value it removed*.

② You can pop arbitrary items from a list. Just pass a positional index to the `pop()` method. It will remove that item, shift all the items after it to “fill the gap,” and return the value it removed.

③ Calling `pop()` on an empty list raises an exception.

☞ Calling the `pop()` list method without an argument is like the `pop()` function in Perl. It removes the last item from the list and returns the value of the removed item. Perl has another function, `shift()`, which removes the first item and returns its value; in Python, this is equivalent to `a_list.pop(0)`.

2.4.7. Lists In A Boolean Context

You can also use a list in a boolean context, such as an `if` statement.

Empty lists are false; all other lists are true.

```
>>> def is_it_true(anything):
...   if anything:
...     print("yes, it's true")
...   else:
...     print("no, it's false")
...
>>> is_it_true([])  ①
no, it's false
>>> is_it_true(['a'])  ②
yes, it's true
>>> is_it_true([False])  ③
yes, it's true
```

① In a boolean context, an empty list is false.

② Any list with at least one item is true.

③ Any list with at least one item is true. The value of the items is irrelevant.

⁂

2.5. Tuples

A tuple is an immutable list. A tuple can not be changed in any way once it is created.

```
>>> a_tuple = ("a", "b", "mpilgrim", "z", "example")  ①
>>> a_tuple
('a', 'b', 'mpilgrim', 'z', 'example')
>>> a_tuple[0]  ②
'a'
```

```
>>> a_tuple[-1] ③
'example'
>>> a_tuple[1:3] ④
('b', 'mpilgrim')
```

① A tuple is defined in the same way as a list, except that the whole set of elements is enclosed in parentheses instead of square brackets.
② The elements of a tuple have a defined order, just like a list. Tuple indices are zero-based, just like a list, so the first element of a non-empty tuple is always `a_tuple[0]`.
③ Negative indices count from the end of the tuple, just like a list.
④ Slicing works too, just like a list. When you slice a list, you get a new list; when you slice a tuple, you get a new tuple.

The major difference between tuples and lists is that tuples can not be changed. In technical terms, tuples are immutable. In practical terms, they have no methods that would allow you to change them. Lists have methods like `append()`, `extend()`, `insert()`, `remove()`, and `pop()`. Tuples have none of these methods. You can slice a tuple (because that creates a new tuple), and you can check whether a tuple contains a particular value (because that doesn't change the tuple), and… that's about it.

```
# continued from the previous example
>>> a_tuple
('a', 'b', 'mpilgrim', 'z', 'example')
>>> a_tuple.append("new") ①
Traceback (innermost last):
  File "<interactive input>", line 1, in ?
AttributeError: 'tuple' object has no attribute 'append'
>>> a_tuple.remove("z") ②
Traceback (innermost last):
  File "<interactive input>", line 1, in ?
AttributeError: 'tuple' object has no attribute 'remove'
>>> a_tuple.index("example") ③
4
>>> "z" in a_tuple ④
True
```

① You can't add elements to a tuple. Tuples have no `append()` or `extend()` method.
② You can't remove elements from a tuple. Tuples have no `remove()` or `pop()` method.
③ You *can* find elements in a tuple, since this doesn't change the tuple.
④ You can also use the `in` operator to check if an element exists in the tuple.

So what are tuples good for?

- Tuples are faster than lists. If you're defining a constant set of values and all you're ever going to do with it is iterate through it, use a tuple instead of a list.
- It makes your code safer if you "write-protect" data that doesn't need to be changed. Using a tuple instead of a list is like having an implied `assert` statement that shows this data is constant, and that special thought (and a specific function) is required to

override that.

- Some tuples can be used as dictionary keys (specifically, tuples that contain *immutable* values like strings, numbers, and other tuples). Lists can never be used as dictionary keys, because lists are not immutable.

☞ Tuples can be converted into lists, and vice-versa. The built-in `tuple()` function takes a list and returns a tuple with the same elements, and the `list()` function takes a tuple and returns a list. In effect, `tuple()` freezes a list, and `list()` thaws a tuple.

2.5.1. Tuples In A Boolean Context

You can use tuples in a boolean context, such as an `if` statement.

```
>>> def is_it_true(anything):
...   if anything:
...     print("yes, it's true")
...   else:
...     print("no, it's false")
...
>>> is_it_true(()) ①
no, it's false
>>> is_it_true(('a', 'b')) ②
yes, it's true
>>> is_it_true((False,)) ③
yes, it's true
>>> type((False)) ④
<class 'bool'>
>>> type((False,))
<class 'tuple'>
```

① In a boolean context, an empty tuple is false.

② Any tuple with at least one item is true.

③ Any tuple with at least one item is true. The value of the items is irrelevant. But what's that comma doing there?

④ To create a tuple of one item, you need a comma after the value. Without the comma, Python just assumes you have an extra pair of parentheses, which is harmless, but it doesn't create a tuple.

2.5.2. Assigning Multiple Values At Once

Here's a cool programming shortcut: in Python, you can use a tuple to assign multiple values at once.

```
>>> v = ('a', 2, True)
>>> (x, y, z) = v ①
>>> x
```

```
'a'
>>> y
2
>>> z
True
```

① `v` is a tuple of three elements, and `(x, y, z)` is a tuple of three variables. Assigning one to the other assigns each of the values of `v` to each of the variables, in order.

This has all kinds of uses. Suppose you want to assign names to a range of values. You can use the built-in `range()` function with multi-variable assignment to quickly assign consecutive values.

```
>>> (MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY, SATURDAY, SUNDAY) = range(7) ①
>>> MONDAY ②
0
>>> TUESDAY
1
>>> SUNDAY
6
```

① The built-in `range()` function constructs a sequence of integers. (Technically, the `range()` function returns an iterator, not a list or a tuple, but you'll learn about that distinction later.) `MONDAY`, `TUESDAY`, `WEDNESDAY`, `THURSDAY`, `FRIDAY`, `SATURDAY`, and `SUNDAY` are the variables you're defining. (This example came from the `calendar` module, a fun little module that prints calendars, like the UNIX program `cal`. The `calendar` module defines integer constants for days of the week.)

② Now each variable has its value: `MONDAY` is 0, `TUESDAY` is `1`, and so forth.

You can also use multi-variable assignment to build functions that return multiple values, simply by returning a tuple of all the values. The caller can treat it as a single tuple, or it can assign the values to individual variables. Many standard Python libraries do this, including the `os` module, which you'll learn about in the next chapter.

⁂

2.6. Sets

A set is an unordered "bag" of unique values. A single set can contain values of any immutable datatype. Once you have two sets, you can do standard set operations like union, intersection, and set difference.

2.6.1. Creating A Set

First things first. Creating a set is easy.

```
>>> a_set = {1} ①
>>> a_set
{1}
>>> type(a_set) ②
<class 'set'>
>>> a_set = {1, 2} ③
>>> a_set
{1, 2}
```

① To create a set with one value, put the value in curly brackets (`{}`).
② Sets are actually implemented as classes, but don't worry about that for now.
③ To create a set with multiple values, separate the values with commas and wrap it all up with curly brackets.

You can also create a set out of a list.

```
>>> a_list = ['a', 'b', 'mpilgrim', True, False, 42]
>>> a_set = set(a_list) ①
>>> a_set ②
{'a', False, 'b', True, 'mpilgrim', 42}
>>> a_list ③
['a', 'b', 'mpilgrim', True, False, 42]
```

① To create a set from a list, use the `set()` function. (Pedants who know about how sets are implemented will point out that this is not really calling a function, but instantiating a class. I *promise* you will learn the difference later in this book. For now, just know that `set()` acts like a function, and it returns a set.)
② As I mentioned earlier, a single set can contain values of any datatype. And, as I mentioned earlier, sets are *unordered*. This set does not remember the original order of the list that was used to create it. If you were to add items to this set, it would not remember the order in which you added them.
③ The original list is unchanged.

Don't have any values yet? Not a problem. You can create an empty set.

```
>>> a_set = set() ①
>>> a_set ②
set()
>>> type(a_set) ③
<class 'set'>
>>> len(a_set) ④
0
>>> not_sure = {} ⑤
>>> type(not_sure)
<class 'dict'>
```

① To create an empty set, call `set()` with no arguments.

② The printed representation of an empty set looks a bit strange. Were you expecting `{}`, perhaps? That would denote an empty dictionary, not an empty set. You'll learn about dictionaries later in this chapter.

③ Despite the strange printed representation, this *is* a set…

④ …and this set has no members.

⑤ Due to historical quirks carried over from Python 2, you can not create an empty set with two curly brackets. This actually creates an empty dictionary, not an empty set.

2.6.2. Modifying A Set

There are two different ways to add values to an existing set: the `add()` method, and the `update()` method.

```
>>> a_set = {1, 2}
>>> a_set.add(4)  ①
>>> a_set
{1, 2, 4}
>>> len(a_set)  ②
3
>>> a_set.add(1)  ③
>>> a_set
{1, 2, 4}
>>> len(a_set)  ④
3
```

① The `add()` method takes a single argument, which can be any datatype, and adds the given value to the set.

② This set now has 3 members.

③ Sets are bags of *unique values*. If you try to add a value that already exists in the set, it will do nothing. It won't raise an error; it's just a no-op.

④ This set *still* has 3 members.

```
>>> a_set = {1, 2, 3}
>>> a_set
{1, 2, 3}
>>> a_set.update({2, 4, 6})  ①
>>> a_set  ②
{1, 2, 3, 4, 6}
>>> a_set.update({3, 6, 9}, {1, 2, 3, 5, 8, 13})  ③
>>> a_set
{1, 2, 3, 4, 5, 6, 8, 9, 13}
>>> a_set.update([10, 20, 30])  ④
>>> a_set
{1, 2, 3, 4, 5, 6, 8, 9, 10, 13, 20, 30}
```

① The `update()` method takes one argument, a set, and adds all its members to the original set. It's as if you called the `add()` method with each member of the set.

② Duplicate values are ignored, since sets can not contain duplicates.

③ You can actually call the `update()` method with any number of arguments. When called with two sets, the `update()` method adds all the members of each set to the original set (dropping duplicates).

④ The `update()` method can take objects of a number of different datatypes, including lists. When called with a list, the `update()` method adds all the items of the list to the original set.

2.6.3. Removing Items From A Set

There are three ways to remove individual values from a set. The first two, `discard()` and `remove()`, have one subtle difference.

```
>>> a_set = {1, 3, 6, 10, 15, 21, 28, 36, 45}
>>> a_set
{1, 3, 36, 6, 10, 45, 15, 21, 28}
>>> a_set.discard(10) ①
>>> a_set
{1, 3, 36, 6, 45, 15, 21, 28}
>>> a_set.discard(10) ②
>>> a_set
{1, 3, 36, 6, 45, 15, 21, 28}
>>> a_set.remove(21) ③
>>> a_set
{1, 3, 36, 6, 45, 15, 28}
>>> a_set.remove(21) ④
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
KeyError: 21
```

① The `discard()` method takes a single value as an argument and removes that value from the set.

② If you call the `discard()` method with a value that doesn't exist in the set, it does nothing. No error; it's just a no-op.

③ The `remove()` method also takes a single value as an argument, and it also removes that value from the set.

④ Here's the difference: if the value doesn't exist in the set, the `remove()` method raises a `KeyError` exception.

Like lists, sets have a `pop()` method.

```
>>> a_set = {1, 3, 6, 10, 15, 21, 28, 36, 45}
>>> a_set.pop() ①
1
>>> a_set.pop()
3
>>> a_set.pop()
36
>>> a_set
{6, 10, 45, 15, 21, 28}
```

```
>>> a_set.clear()  ②
>>> a_set
set()
>>> a_set.pop()  ③
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
KeyError: 'pop from an empty set'
```

① The `pop()` method removes a single value from a set and returns the value. However, since sets are unordered, there is no "last" value in a set, so there is no way to control which value gets removed. It is completely arbitrary.
② The `clear()` method removes *all* values from a set, leaving you with an empty set. This is equivalent to `a_set = set()`, which would create a new empty set and overwrite the previous value of the `a_set` variable.
③ Attempting to pop a value from an empty set will raise a `KeyError` exception.

2.6.4. Common Set Operations

Python's `set` type supports several common set operations.

```
>>> a_set = {2, 4, 5, 9, 12, 21, 30, 51, 76, 127, 195}
>>> 30 in a_set  ①
True
>>> 31 in a_set
False
>>> b_set = {1, 2, 3, 5, 6, 8, 9, 12, 15, 17, 18, 21}
>>> a_set.union(b_set)  ②
{1, 2, 195, 4, 5, 6, 8, 12, 76, 15, 17, 18, 3, 21, 30, 51, 9, 127}
>>> a_set.intersection(b_set)  ③
{9, 2, 12, 5, 21}
>>> a_set.difference(b_set)  ④
{195, 4, 76, 51, 30, 127}
>>> a_set.symmetric_difference(b_set)  ⑤
{1, 3, 4, 6, 8, 76, 15, 17, 18, 195, 127, 30, 51}
```

① To test whether a value is a member of a set, use the `in` operator. This works the same as lists.
② The `union()` method returns a new set containing all the elements that are in *either* set.
③ The `intersection()` method returns a new set containing all the elements that are in *both* sets.
④ The `difference()` method returns a new set containing all the elements that are in `a_set` but not `b_set`.
⑤ The `symmetric_difference()` method returns a new set containing all the elements that are in *exactly one* of the sets.

Three of these methods are symmetric.

```
# continued from the previous example
>>> b_set.symmetric_difference(a_set)  ①
{3, 1, 195, 4, 6, 8, 76, 15, 17, 18, 51, 30, 127}
```

```
>>> b_set.symmetric_difference(a_set) == a_set.symmetric_difference(b_set) ②
True
>>> b_set.union(a_set) == a_set.union(b_set) ③
True
>>> b_set.intersection(a_set) == a_set.intersection(b_set) ④
True
>>> b_set.difference(a_set) == a_set.difference(b_set) ⑤
False
```

① The symmetric difference of `a_set` from `b_set` *looks* different than the symmetric difference of `b_set` from `a_set`, but remember, sets are unordered. Any two sets that contain all the same values (with none left over) are considered equal.

② And that's exactly what happens here. Don't be fooled by the Python Shell's printed representation of these sets. They contain the same values, so they are equal.

③ The union of two sets is also symmetric.

④ The intersection of two sets is also symmetric.

⑤ The difference of two sets is not symmetric. That makes sense; it's analogous to subtracting one number from another. The order of the operands matters.

Finally, there are a few questions you can ask of sets.

```
>>> a_set = {1, 2, 3}
>>> b_set = {1, 2, 3, 4}
>>> a_set.issubset(b_set) ①
True
>>> b_set.issuperset(a_set) ②
True
>>> a_set.add(5) ③
>>> a_set.issubset(b_set)
False
>>> b_set.issuperset(a_set)
False
```

① `a_set` is a subset of `b_set` — all the members of `a_set` are also members of `b_set`.

② Asking the same question in reverse, `b_set` is a superset of `a_set`, because all the members of `a_set` are also members of `b_set`.

③ As soon as you add a value to `a_set` that is not in `b_set`, both tests return `False`.

2.6.5. Sets In A Boolean Context

You can use sets in a boolean context, such as an `if` statement.

```
>>> def is_it_true(anything):
...   if anything:
...     print("yes, it's true")
...   else:
```

```
...     print("no, it's false")
...
>>> is_it_true(set()) ①
no, it's false
>>> is_it_true({'a'}) ②
yes, it's true
>>> is_it_true({False}) ③
yes, it's true
```

① In a boolean context, an empty set is false.

② Any set with at least one item is true.

③ Any set with at least one item is true. The value of the items is irrelevant.

⁂

2.7. Dictionaries

A dictionary is an unordered set of key-value pairs. When you add a key to a dictionary, you must also add a value for that key. (You can always change the value later.) Python dictionaries are optimized for retrieving the value when you know the key, but not the other way around.

☞ A dictionary in Python is like a hash in Perl 5. In Perl 5, variables that store hashes always start with a `%` character. In Python, variables can be named anything, and Python keeps track of the datatype internally.

2.7.1. Creating A Dictionary

Creating a dictionary is easy. The syntax is similar to sets, but instead of values, you have key-value pairs. Once you have a dictionary, you can look up values by their key.

```
>>> a_dict = {'server': 'db.diveintopython3.org', 'database': 'mysql'} ①
>>> a_dict
{'server': 'db.diveintopython3.org', 'database': 'mysql'}
>>> a_dict['server'] ②
'db.diveintopython3.org'
>>> a_dict['database'] ③
'mysql'
>>> a_dict['db.diveintopython3.org'] ④
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
KeyError: 'db.diveintopython3.org'
```

① First, you create a new dictionary with two items and assign it to the variable a_dict. Each item is a key-value pair, and the whole set of items is enclosed in curly braces.

② 'server' is a key, and its associated value, referenced by a_dict['server'], is 'db.diveintopython3.org'.

③ 'database' is a key, and its associated value, referenced by a_dict['database'], is 'mysql'.

④ You can get values by key, but you can't get keys by value. So a_dict['server'] is 'db.diveintopython3.org', but a_dict['db.diveintopython3.org'] raises an exception, because 'db.diveintopython3.org' is not a key.

2.7.2. Modifying A Dictionary

Dictionaries do not have any predefined size limit. You can add new key-value pairs to a dictionary at any time, or you can modify the value of an existing key. Continuing from the previous example:

```
>>> a_dict
{'server': 'db.diveintopython3.org', 'database': 'mysql'}
>>> a_dict['database'] = 'blog'  ①
>>> a_dict
{'server': 'db.diveintopython3.org', 'database': 'blog'}
>>> a_dict['user'] = 'mark'  ②
>>> a_dict  ③
{'server': 'db.diveintopython3.org', 'user': 'mark', 'database': 'blog'}
>>> a_dict['user'] = 'dora'  ④
>>> a_dict
{'server': 'db.diveintopython3.org', 'user': 'dora', 'database': 'blog'}
>>> a_dict['User'] = 'mark'  ⑤
>>> a_dict
{'User': 'mark', 'server': 'db.diveintopython3.org', 'user': 'dora', 'database': 'blog'}
```

① You can not have duplicate keys in a dictionary. Assigning a value to an existing key will wipe out the old value.

② You can add new key-value pairs at any time. This syntax is identical to modifying existing values.

③ The new dictionary item (key 'user', value 'mark') appears to be in the middle. In fact, it was just a coincidence that the items appeared to be in order in the first example; it is just as much a coincidence that they appear to be out of order now.

④ Assigning a value to an existing dictionary key simply replaces the old value with the new one.

⑤ Will this change the value of the user key back to "mark"? No! Look at the key closely — that's a capital U in "User". Dictionary keys are case-sensitive, so this statement is creating a new key-value pair, not overwriting an existing one. It may look similar to you, but as far as Python is concerned, it's completely different.

2.7.3. Mixed-Value Dictionaries

Dictionaries aren't just for strings. Dictionary values can be any datatype, including integers, booleans, arbitrary objects, or even other dictionaries. And within a single dictionary, the values don't all need to be the same type; you can mix and match as needed. Dictionary keys are more restricted, but they can be strings, integers, and a few other types. You can also mix and match key datatypes within a dictionary.

In fact, you've already seen a dictionary with non-string keys and values, in your first Python program.

```
SUFFIXES = {1000: ['KB', 'MB', 'GB', 'TB', 'PB', 'EB', 'ZB', 'YB'],
            1024: ['KiB', 'MiB', 'GiB', 'TiB', 'PiB', 'EiB', 'ZiB', 'YiB']}
```

Let's tear that apart in the interactive shell.

```
>>> SUFFIXES = {1000: ['KB', 'MB', 'GB', 'TB', 'PB', 'EB', 'ZB', 'YB'],
...             1024: ['KiB', 'MiB', 'GiB', 'TiB', 'PiB', 'EiB', 'ZiB', 'YiB']}
>>> len(SUFFIXES) ①
2
>>> 1000 in SUFFIXES ②
True
>>> SUFFIXES[1000] ③
['KB', 'MB', 'GB', 'TB', 'PB', 'EB', 'ZB', 'YB']
>>> SUFFIXES[1024] ④
['KiB', 'MiB', 'GiB', 'TiB', 'PiB', 'EiB', 'ZiB', 'YiB']
>>> SUFFIXES[1000][3] ⑤
'TB'
```

① Like lists and sets, the `len()` function gives you the number of keys in a dictionary.
② And like lists and sets, you can use the `in` operator to test whether a specific key is defined in a dictionary.
③ `1000` *is* a key in the `SUFFIXES` dictionary; its value is a list of eight items (eight strings, to be precise).
④ Similarly, `1024` is a key in the `SUFFIXES` dictionary; its value is also a list of eight items.
⑤ Since `SUFFIXES[1000]` is a list, you can address individual items in the list by their 0-based index.

2.7.4. Dictionaries In A Boolean Context

You can also use a dictionary in a boolean context, such as an `if` statement.

Empty dictionaries are false; all other dictionaries are true.

```
>>> def is_it_true(anything):
...   if anything:
...     print("yes, it's true")
...   else:
...     print("no, it's false")
...
>>> is_it_true({}) ①
no, it's false
>>> is_it_true({'a': 1}) ②
yes, it's true
```

① In a boolean context, an empty dictionary is false.

①

② Any dictionary with at least one key-value pair is true.

⁂

2.8. None

None is a special constant in Python. It is a null value. None is not the same as False. None is not 0. None is not an empty string. Comparing None to anything other than None will always return False.

None is the only null value. It has its own datatype (NoneType). You can assign None to any variable, but you can not create other NoneType objects. All variables whose value is None are equal to each other.

```
>>> type(None)
<class 'NoneType'>
>>> None == False
False
>>> None == 0
False
>>> None == ''
False
>>> None == None
True
>>> x = None
>>> x == None
True
>>> y = None
>>> x == y
True
```

2.8.1. None In A Boolean Context

In a boolean context, None is false and not None is true.

```
>>> def is_it_true(anything):
...   if anything:
...     print("yes, it's true")
...   else:
...     print("no, it's false")
...
>>> is_it_true(None)
```

```
no, it's false
>>> is_it_true(not None)
yes, it's true
```

⁂

2.9. Further Reading

- Boolean operations
- Numeric types
- Sequence types
- Set types
- Mapping types
- `fractions` module
- `math` module
- PEP 237: Unifying Long Integers and Integers
- PEP 238: Changing the Division Operator