Chapter 2, Beginning with Python

John M. Morrison

April 21, 2022

Contents

0 Running Python

The chapter s0.pdf contains a guide to installing Python, so this should now be up and running.

You will also want a plain text editor (not a word processor). If you are a UNIX/Mac user, there is good old `vim`. VSCode is a great editor. It can be customized to have vim bindings, and it supports a host of languages and has useful extensions for them that are easy to install. If you are not a vim ninja, it is a top choice.

Throughout this book, we will use the symbol `unix>` to represent your system prompt, whether it is a Windoze cmd window or a UNIX terminal window. You can start Python at the command line like so.

```
unix> python3
Python 3.10.0 (v3.10.0:b494f5935c, Oct  4 2021, 14:59:19)
[Clang 12.0.5 (clang-1205.0.22.11)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

What you see is the Python prompt. To exit, type control-D or `quit()`. To make a program, you can use your text editor.

A useful accompaniment to this chapter is a series [2] of videos by the zany Net Ninjas. Corey Schaefer's YouTube channel [4] contains a wealth of information on Python.

1 Scalar Types

We begin by looking at the simplest types Python's type system. We will explore this via the interactive Python prompt. All of these types are immutable objects. The most basic type is the integer type, `int`. Integers have the expected behavior in the presence of arithmetic operators. We demonstrate the basic operators here.

You also see how comments are done; everything on a line after a # is a comment.

```
>>> 42 + 58          #addition
100
>>> 21-57            #subtraction
-36
>>> 66*5             #multiplication
330
>>> 44/3             #division
14.666666666666666
>>> 44//3            #integer division
14
>>> 365%7            #mod
1
>>> 2**20            #exponentiation
1048576
```

Integers do not overflow. This type admits integers of arbitrary precision, subject to the (gargantuan) limits on memory.

```
>>> 2**1000
10715086071862673209484250490600018105614048117055336074437503883703510511249361
22493198378815695858127594672917553146825187145285692314043598457757469857480393
45677748242309854210746050623711418779541821530464749835819412673987675591655439
46077062914571196477686542167660429831652624386837205668069376
>>>
```

Programming Exercises

1. Can you compute 2^{10000}?
2. There is an infix binary operator on the integers, `^`. Can you experiment with this and figure out what it does? *Hint.* Look at the binary expansions of numbers you operate on. Use the built-in Python function `bin` to compute binary expansions for the integers you fiddle with. We show `bin` in action here. A `0b` prefix means, "this is a binary number."

   ```
   >>> bin(42)
   '0b101010'
   >>> 0b1110
   14
   ```

3. Look up `bin`, `hex`, and `oct` in [3] and read about them. Experiment with them and see their action.
4. There is an infix binary operator on the integers, `&`. Can you experiment with this and figure out what it does? Look at binary expansions for a clue.
5. There is an infix binary operator on the integers, `|`. Can you experiment with around with this and figure out what it does?

Since we have seen a floating point number, let us formally introduce those. This type is known as `float`. You will see few surprises.

```
>>> 2.0 + 3
5.0
>>> 6.02e23 * 100
6.02e+25
>>> 5.3 - 6.1
-0.7999999999999998
>>> 3/666
0.0045045045045045045
>>> 1.0001**10000
2.7181459268249255
```

Notice that when you add an integer and a floating point number, that the integer gets converted into a floating point number. As we said before, Python's floating point numbers are IEEE 754 double-precision 64 bit numbers.

Programming Exercise: Exploring Numbers This is a little puzzler project in which you do some scientific calculations. You are allowed *only* these facts. Remember in the metric system, centi- means 1/100, milli- means 1/1000 and kilo- means 1000. Use Python's interactive mode to make this happen.

- 1 in = 2.54 cm (length)
- 1 liter = 33.8 fluid ounces (volume) = 1000 cm3
- 1 mile = 5280 ft (length)
- 1 foot = 12 in (length)
- 1 hour = 60 min (time)
- 1 min = 60 sec (time)
- 1 yr = 365.24 days (time)
- 1 kg = 2.204 lbs
- 1 hr = 60 min, 1 min = 60 s, 1 day = 24 hr(time)
- 1 ton = 2000 lbs
- 1 acre = 1/640 mi^2

Now use these facts to answer these questions.

1. Given that light travels at 2.9979e8 (that's 2.9979*108 in scientific notation) meters per second, figure out how fast light moves in miles per second. Then convert this to miles per hour.

2. Tell the time it take for light to go from the sun to the earth if the mean distance of the sun to the earth is 93.0 million miles.
3. Use the fact that a liter of water weighs one kilogram and that one gallon of water weighs 8.33 lbs to determine the number of cubic inches in a gallon and the number of pounds in a cubic foot of water.
4. Let us assume that humans weigh an average of 140 lbs and that humans have about the same density as water. If the population of the earth is 7.0 billion humans, estimate the total volume of humanity in cubic miles. Do you find this counterintuitive?
5. An *acre-foot* of water is enough water to cover one acre one foot deep. How many gallons of water are in an acre-foot? What does that water weigh in tons?

Programming Exercises: A peek ahead Note: you will see we are using the new Python3 style of f-string, instead of the old "%" and `format` methods. Eventually `%` that will be deprecated. If you are going to write new code, use the newer methods.

1. Enter these things in an interactive session.

   ```
   f"{3/7:.1f}"
   f"{3/7:.2f}"
   f"{3/7:.8f}"
   ```

   What kind of object is returned? What do you see? What does the number to the right of the point do? If it's not clear from the examples shown, try a few more.
2. Enter this in an interactive session.

   ```
   f"{3/7:.8e}"
   ```

   Experiment with different numbers. What is nice about this?
3. Now test-drive this.

   ```
   f"{2:.5f}"
   ```
4. What happens if you put an integer in front of the point? What if that integer is negative? Experiment and determine. You will learn some nifty stuff about formatting numbers.

The most commonly-used type in any programming languages is the string; you just got a preview of strings in the exercises above. Strings are used in Python to hold globs of text. Strings support the infix binary operator + and an infix operator * that takes an integer and a string as arguments.

```
>>> "Happy " + "Happy"
'Happy Happy'
```

```
>>> "*"*50
'**************************************************'
```

The * operator requires an integer and a string and it returns a string that repeats the string operand the integer number of times. If the integer is 0 or negative, an empty string is returned.

Python has a boolean type, `bool` which has two elements `True` and `False`. There are two infix binary operators on Booleans; they are `and` and `or`. If `P` and `Q` are predicates (Boolean-valued expressions), then `P and Q` is true precisely when both `P` and `Q` are true. The predicate `P or Q` is true precisely when at least one of `P` and `Q` is true.

There is also a unary prefix operator `not` which reverses the truth-value of its operand. The order of operations in Boolean expressions is `not`, `and`, then the lowest is `or`. As you would expect, you can use parentheses to override this order of operations; when in doubt avail yourself of them.

Python has the standard relational operators. They all return a Boolean value, as you would expect. We show them here in a table.

`<`	less than
`>`	greater than
`<=`	less than or equal to
`>=`	greater than or equal to
`==`	equal to
`!=`	not equal to

For number types they do numerical comparison. For strings, they compare asciicographically. Python has an additional infix binary operator, `is`, which tests for equality of identity.

All of the scalar types we have seen so far are immutable objects. Once created they cannot be changed in-place. You might ask, "why this immutability?" Immutability allows Python to pool objects, which enables the recycling of commonly used objects without wasting a lot of time creating and deallocating them. In fact, when Python runs, it pre-loads small integers into its memory

For evidence of this take note of this little Python session.

```
>>> hex(id(0))
'0x10b242470'
```

We see the virtual address where Python is storing zero. Now watch this.

```
>>> id(1) - id(0)
32
```

Notice that 1 is stored 32 bits away from zero. Now observe this.

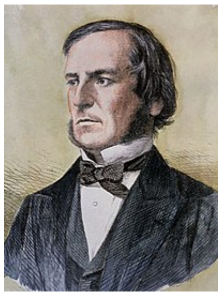
```
>>> id(256) - id(0)
8192
>>> id(260) - id(0)
3130752
```

We see that 0-256 are in the little integer pool. Once you get to 257 we surnmise things get stored elsewhere because of what we see here.

```
>>> hex(id(256)
... )
'0x10b244470'
>>> hex(id(257))
'0x10b53edf0'
```

You are encouraged to do a little spelunking and see where the negative integers go.

Programming and Writing Exercise: PEMDAS for George George Boole was a pioneer of modern logic, as well as a versatile mathematician who studied differential equations. He is the source of the name `boolean` you see that refers to a calculus of true/false values.



You are to perform experiments in the interactive shell to make an airtight case for your determination of the order of operations and, or and not.

Documentation Go to this URL, `https://docs.python.org/3/library/stdtypes.html"` for a repository of information on Python's built-in types. You will see the number and boolean types near the beginning. Further down the page is a section entitled, "Text Sequence Type — str." You can explore the string type there.

This URL, `https://docs.python.org/3/library/functions.html` has all of the Python built-in functions. You should poke around in here and look up the functions we have seen so far.

2 Variables and Assignment

You have programmed in some other language, and if you did, you might have seen that variables have a type. This is true in Java, C, and C++. This is not true in Python. Variables are typeless names that allow you to refer to objects.

Do not make the mistake of thinking Python is "weakly typed." Objects are keenly aware of their types. Bear witness to this little session. Every Python object knows its type. Python is *duck typed*; you can read about duck typing in [5].

```
>>> type(1)
<class 'int'>
>>> type(1.0)
<class 'float'>
>>> type("caterwaul")
<class 'str'>
>>> type(True)
<class 'bool'>
```

Objects do not have an identity crisis either. Every object in a Python program has a unique ID during its lifetime. We will use Python's built-in `hex` function to see these as hex numbers. These values will vary for your session. What you see is the memory address of your object in Python's virtual machine.

```
>>> hex(id(1))
'0x100214870'
>>> hex(id(2))
'0x100214890'
>>> hex(id(1))
'0x100214870'
>>> hex(id(1.0))
'0x10036f180'
>>> hex(id("caterwaul"))
'0x1019a4bf0'
>>> hex(id(True))
'0x1001b76b0'
```

The rule for Python variable naming is that the first character must be alphabetical or an underscore. Remaining characters can be alphanumeric or underscores. The operator `=` is used to bind variables to objects. It works in a manner entirely similar to other languages. Here we show it at work.

```
>>> x = 5
>>> y = 6
>>> x*y == 30
True
```

Notice you just met `==`, the isequalto operator. The value `x` is *not* storing the value 5. What it is storing is the location in memory (memory address) where the value 5 is being kept. You can see that what `x` is actually storing is the `id` of 5. In a word: Python variables know *where* to find their objects. In all cases, variables refer indirectly to their objects. What they actually store is a integer indicating where that object is being stored in memory. Bear this in mind as we proceed; it will save you a lot of confusion.

```
>>> hex(id(x))
'0x1002148f0'
>>> hex(id(5))
'0x1002148f0'
```

What happened on the last line, `x*y == 30`? Here, the expression was evaluated by fetching the values of `x` and `y`, substituting them into the expression and finding that $5 * 6$ indeed equals 30.

2.1 The Lowdown on Assignment

The familiar arithmetic operations and the Boolean operations all associate from left to right. To wit, when you evaluate expressions, you work from left to right. Here we see this in action with multiplication and division.

$$4 * 5/2 * 8 = 20/2 * 8 = 10 * 8 = 80.$$

Now see it in a complex operation. Consider this expression.

$$4^3 - 2 * 7 * 5/35 + 4 * 18$$

We begin by doing all exponentiation.

$$4^3 - 2 * 7 * 5/35 + 4 * 18 = 64 - 2 * 7 * 5 + 4 * 18$$

We then munch up the multiplication and division in each term from left to right.

$$64 - 2 * 7 * 5 + 4 * 18 = 64 - 14 * 5 + 72 = 64 - 70 + 72$$

Lastly we resolve addition and subtraction.

$$64 - 14 * 5 + 72 = 64 - 70 + 72 = -6 + 72 = 66.$$

Notice how we work in each case from left to right.

Assignment works backwards. It begins on the right and works left. It also has lower precedence than any other arithmetic operation, so it happens last.

Here is a very typical assignment you might see.

```
>>> x = 5
>>> x = 2*x + 10
>>> x
20
```

Let us look at this in detail. Python first sees the variable `x` being bound to the value `5`. Now we turn our attention to the second line. We begin with

```
    x = 2*x + 10.
```

Multiplication is first carried out. The term `2*x` evaluate to `10` and we have

```
    x = 10 + 10.
```

Now addition occurs.

```
    x = 20
```

Now, `x` is bound to the value `20`. What happened to its prior value `5`? This value got orphaned. To change the value of any variable pointing at any of scalar types we have seen so far, we have we get it to point at an entirely different object. We never modified the object sitting in memory. All of the scalar types are immutable; once created in memory they never change. In particular, the objects `True` and `False` are unique in memory.

Things that can appear on the left-hand side of an assignment are called *lvalues*. Variables are always lvalues; we will meet a few other things as we go along that are also lvalues. Literals, or actual objects, are not. You cannot assign to numbers, Booleans, or strings. Try this stunt and you will get hissed at.

```
5 = 4 + 1
```

Python offers a lagniappe that is a twist on assignment. Look at this.

```
>>> a = 5
>>> b = 4
>>> a,b = b,a
>>> a
4
>>> b
5
```

Just a spoonful of syntactic sugar helps the medicine go down.

Programming Exercises

1. Do this and see how Python hisses.

   ```
   5 = x
   ```

2. What happens here?

   ```
   >>> a = 1
   >>> b = 2
   >>> c = 3
   >>> a,b,c = b,c,a
   ```

3. What happens when you do this?

   ```
   >>> a, b, c = c
   ```

3 Pooling

Python is a *garbage-collected* language. A mechanism called the garbage collector lurks behind the scenes, deallocating the memory for objects no longer in use.

Some objects don't get picked up by the garbage collectors; these exceptions are pooled objects. Python caches small integers in memory; when they reappear because a variable needs to point at them, the variable just points at the pooled value. Python also caches small strings in memory in an area called the *string pool*. Pooling of these immutable objects increases efficiency; equality of pooled strings is achieved by comparing memory addresses, obviating the need to loop through the strings.

Now, it's time to break out the `is` operator and see it at work.

```
>>> x = 5
>>> y = 5
>>> x is y
True
>>> name = "flibbertygibbet"
>>> elisa = "flibbertygibbet"
>>> name is elisa
True
```

The variables `x` and `y` are sharing the common item `5` in memory. The same is true for the two strings pointing at `"flibbertygibbet"`. Note that only two Boolean values are ever stored in memory, `True` and `False`.

```
>>> True is True
True
```

```
>>> False is (6*4 == 5*50)
True
```

Programming Exercises: Time for a dip!

1. Run this code.

   ```
   for k in range(200,300):
       print(k, hex(id(k)))
   ```

   What can you discern about the pooling of small integers?
2. Can you find anything out about negative integers by replaying this theme?

4 Writing a Program

A Python program is just a sequence of Python statements in a file. We will use the creation of this example as an opportunity to introduce the built-in function `print`, which puts things to `stdout`. Enter this with your favorite text editor into a file named `print_example.py`.

```
print("Hello, World")
print(1,2,3,4, sep="|")
print(1,2,3,4, sep = " ", end = "peep")
```

Now open a cmd or terminal window and navigate to the directory containing your program. Run this program at the command line as follows. The items `sep` and `end` are referred to as *keyword arguments*.

```
unix> python print_example.py
Hello, World
1|2|3|4
1 2 3 4peep
unix>
```

A Note to UNIX (Yes, Mac too... Users) When you make your program, place this line at the top

```
#!/usr/bin env python3
```

and use `chmod` to make the program executable If you do this to our litle sample program, you can do this. Notice what the keyword arguments `sep` and `end` do.

```
unix> ./print_example.py
Hello, World
1|2|3|4
1 2 3 4peep
unix>
```

Programming Exercises

1. You can get textual input from the user with the `input` function. It works like this.

   ```
   some_variable = input("Your prompt:  ")
   ```

   Write a program that asks for a name and which replies with `Hello, <Name>`.
2. Write a program that asks for two numbers and which presents the user with their product. Note that `input` returns a string.

5 Objects

The term *object* simply refers to a datum stored in memory along with its associated code. There are three important properties of an object

- **State** This refers to things an object *knows*.
- **Identity** This refers to an object's physical presence in memory. It is not possible for two different objects to occupy the same space in memory. This is what an object *is*.
- **Behavior** This refers to what an object *does*.

Let us look at the scalar types we have seen through this lens. Integers exhibit expected behavior when in the presence of arithmetic and relational operators. They know the value that they store. The same is true of floating-point numbers.

Strings are more complex. Their state is simple; this is just the blob of text being stored in the string.

Strings have a wide variety of behaviors. Let us look at a few. We can index into a string; most languages you have seen have this feature.

```
>>> x[0]
'a'
>>> x[1]
'b'
```

```
>>> x[2]
'c'
>>> x[25]
'z'
```

Notice that the indexing is zero-based. The best way to think about these indices is that they live, like rats, “inside of the walls." We say this because they reside *between* then entries of the string.

```
   ----------------------------------           -----------
    |         |         |         |             |          |
    |    a    |    b    |    c    |    . . .    |    z     |
    0         1         2         4             |          |
    ---------------------------------         25----------26
```

As you can see in the picture, there is an index 26 in this string; it is just at the far right-hand end. Each index points at the character just to its right. There is no character for index 26 to point at. Try and see it; Python will hiss at you.

Let us illustrate another behavior, `find`.

```
>>> x.find("c")
2
>>> x.find("fgh")
5
>>> x.find("cow")
-1
```

When `find` does not find, it punts and returns a `-1`. This is often referred to as a *sentinel value*. Notice how `find` needs to be told what to find.

5.1 How do I find all of the string behaviors?

Visit the URL `https://docs.python.org/3/library/stdtypes.html` for information on all of Python’s built-in types. Then find the section on the “Text Sequence Type.” Here is what you will find at the top.

String literals are written in a variety of ways:

1. Single quotes: `’allows embedded "double" quotes’`
2. Double quotes: `"allows embedded ’single’ quotes"`.
3. Triple quoted: `'''Three single quotes'''`, `"""Three double quotes"""`

Triple quoted strings may span multiple lines, and all associated whitespace will be included in the string literal. You can use single or double quotes to bound a triple-quoted string.

Now scroll down the page a short bit to the section entitled "String Methods." The `capitalize()` method is very simple. We show its action. It creates a new string that is capitalized. The original string is untouched.

```
>>> president = "lincoln"
>>> president.capitalize()
'Lincoln'
```

Let us look at how to read the documentation for `center()`.

```
str.center(width[, fillchar])
```

Return centered in a string of length width. Padding is done using the specified fillchar (default is an ASCII space). The original string is returned if width is less than or equal to len(s).

This method has two arguments. The first one, `width`, is required. The second, `fillchar`, is optional; this is indicated by the presence of the square brackets surrounding it. Let us show this at work.

```
>>> x = "Cows With Guns"
>>> x.center(10)          #too little space: same string comes back
'Cows With Guns'
>>> x.center(50)          #default padding is spaces
'                  Cows With Guns                  '
>>> x.center(50, "*")     #padding with stars
'******************Cows With Guns******************'
>>> x.center()            #error: width is required.
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: center() takes at least 1 argument (0 given)
>>>
```

We should mention that Python has many useful built-in functions. Here are three you want to know about.

Function	**Input**	**Output**
`len`	a string	the string's length
`ord`	a one-character string	the char's ASCII/Unicode value
`chr`	an integer	the char with the given ASCII/Unicode code

Programming Exercises Experiment with these useful string methods. Figure out what they do.

1. `rfind`
2. `endswith`
3. `startswith`
4. `lower`
5. `upper`
6. `strip`, `lstrip` and `rstrip`

5.2 Compound Assignment Operators

Many languages have this feature.

```
>>> x = 5
>>> x += 3
>>> x
8
```

Here `x += 3` is shorthand for `x = x + 3`. If you have an infix binary operator `op`, then `x op= foo` is the same as `x = x op foo`. The compound assignment operator works from right to left. Its precedence, like that of `=` is lower than almost everything else.

Notice this little session with strings.

```
>>> x = "some"
>>> id(x)
4323956024
>>> x += "thing"
>>> x
'something'
>>> id(x)
4323954160
```

The variable `x` is a pointing at a new string, because strings cannot be changed in-place. You can see this because the `id` of the object pointed at by `x` changed.

6 Sequence Types

So far, we have concerned ourselves with scalar types that hold a single datum. Now we will look at two new types, lists and tuples. These types have some common features with strings, because a string can be thought of as a character sequence, as well as a glob of text.

Sequences in Python store sequences of memory addresses where the objects comprising them can be found. The objects themselves are not stored in these containers.

Let us first look at lists. We make a list, show its type, and index into it. Notice that the indexing mechanism looks identical to that of strings.

```
>>> x = [1,2,3,4,5]
>>> type(x)
<class 'list'>
>>> x[0]
1
>>> x[1]
2
>>> x[4]
5
>>> x[5]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: list index out of range
>>>
```

What is different is that lists are mutable. Watch this.

```
>>> id(x)
4323859272
>>> x[0] = 100
>>> id(x)
4323859272
>>> x
[100, 2, 3, 4, 5]
>>>
```

We changed this list in-place. It is the same object, but its state has been changed by the assignment `x[0] = 100`. Each entry in the list is an lvalue.

Python tuples are similar to lists, but they are immutable. Once you make a tuple, you cannot change it in-place. Let us imitate the list session. All of this looks the same.

```
>>> x = (1,2,3,4,5)
>>> x
(1, 2, 3, 4, 5)
>>> x[0]
```

```
1
>>> x[1]
2
>>> x[4]
5
>>> x[5]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: tuple index out of range
>>> type(x)
<class 'tuple'>
>>>
```

Now watch this.

```
>>> x[0] = 100
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'tuple' object does not support item assignment
```

Oops. You cannot change a tuple in-place. Tuple entries are *not* lvalues.

Both tuples and lists are heterogeneous. You can put objects of any types into them.

```
>>> motley = [True, "foo", 2, 2.0, ["cat", False, -1]]
>>> motley[4]
['cat', False, -1]
>>> motley[4][1]
False
>>>
```

You can do all of this stuff with a tuple, since you are only reading from it. Try it now!

The `+` operator works just as you might expect for tuples and lists.

```
>>> [1,2,3] + [4,5,6]
[1, 2, 3, 4, 5, 6]
>>> (1,2,3) + (4,5,6)
(1, 2, 3, 4, 5, 6)
```

The built-in `len` function will calculate the length of any sequence. For tuples and lists, this is the number of items present in the tuple or list. For strings, this is the number of characters in the string.

Here are common features for all sequences.

Operation	**List/Tuple**	**String**
`x in s`	`True` if `x` equals a member of `s`	`True` if `x` is a substring of `s`
`s + t`	concatenates `s` and `t`	
`s*n` or `n*s`	repeats `s` `n` times	
`len(s)`	number of objects in `s`	number of characters is `s`
`min(s)`	smallest object in `s`	character in `s` with smallest ASCII value
`max(s)`	largest object in `s`	character in `s` with largest ASCII value
`s.count(x)`	counts the number of times `x` is equal to an element of `s`	counts the number of times the sting `x` appears in `s`

When we refer to the "largest" item in a sequence, we need to do this operation on a list where it makes sense to compare the items. It is best to use this on sequences that are homogeneous, i.e., where all elements are of the same type.

6.1 Slicing of Sequences

Slicing is convenient way of obtaining a subset of a sequence. For strings and tuples, a slice returns a copied subset of the sequence. First we see slicing at an index for a list and a string. Notice that the slice goes all the way to the end.

```
>>> min(x)
'a'
>>> x = "abcdefg"
>>> x[:2]
'ab'
>>> x[2:]
'cdefg'
>>> foo = ["a", "bc", "defg", "hijk"]
>>> foo[2:]
['defg', 'hijk']
>>> foo[:2]
['a', 'bc']
```

You can specify both ends of a slice.

```
>>> foo[1:3]
['bc', 'defg']
>>> x[1:3]
'bc'
```

You can also specify a third "skip" parameter.

```
>>> alpha = "abcdefghijklmnpqrstuvwxyz"
>>> alpha[::3]
'adgjmqtwz'
>>> alpha[5::3]
'filpsvy'
>>> alpha[5:10:3]
'fi'
```

In the first case, we extracted every third character from the string. In the second we did so starting at index 5. In the third, we did so between indices 5 and 10. Be reminded: the indices of a sequence lurk between the sequence's elements.

Programming Exercises

1. How do you find the item in a list or tuple of strings that is first in asciicographical order?
2. How do you count the number of elements in a list of integers of even index that equal 5?
3. How do you count the number of times the letter A appears in a string, case insensitive?
4. If you have a sequence what does the slice [::-1] return? What happens if you put indices between the colons?

6.2 Slicing of Lists

Because lists are mutable, they have additional behavior when slicing occurs. Observe this.

```
>>> x = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12]
>>> x[:5] = []
>>> x
[5, 6, 7, 8, 9, 10, 11, 12]
```

A slice is an lvalue. We can, within bounds, assign to it. If you assign an empty list to a slice of a list consisting of consecutive elements, the slice is removed from the list. Beware of this.

```
>>> x = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12]
>>> x[::2] = []
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: attempt to assign sequence of size 0
to extended slice of size 7
```

There are limits. Here the slice you tried to assign to had non-consecutive elements. You can, however assign this slice to a list of equal length like so.

```
>>> x[::2] = [0, 10, 20, 30, 40, 50, 60]
>>> x
[0, 1, 10, 3, 20, 5, 30, 7, 40, 9, 50, 11, 60]
```

It is interesting to compare the action of `+=` on lists, strings and tuples. Compare these three sessions.

```
>>> x = [1,2,3,4,5]
>>> id(x)
4323953160
>>> x += [6,7, 8]
>>> x
[1, 2, 3, 4, 5, 6, 7, 8]
>>> id(x)
4323953160
```

Here we just modified an object in-place

```
>>> id(x)
4298553304
>>> x += (6,7,8)
>>> x
(1, 2, 3, 4, 5, 6, 7, 8)
>>> id(x)
4323935288
```

Here a new object got created because tuples are not mutable.

```
>>> x = "12345"
>>> id(x)
4323956024
>>> x += "678"
>>> x
'12345678'
>>> id(x)
4323953840
```

The same thing happened to the string and the tuple. New objects got created by `+=`.

7 Casting About

A *cast* is a temporary request to view an object of one type as being that of another. We show some examples here. Any Python object may be cast to a string.

```
>>> int("12345")
12345
>>> float("12345")
12345.0
>>> str(12345)
'12345'
>>> int("12345", 2)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: invalid literal for int() with base 2: '12345'
>>> int("10101110", 2)
174
>>> int("22123312", 4)
42742
```

Observe that you can pass a radix to cast a string to a number in that base. Lists can be cast to tuples and vice versa in the obvious way. You can also cast a string to a list.

```
>>> t = (1, 2, 3, 4)
>>> list(t)
[1, 2, 3, 4]
>>> l = [1,2,3,4]
>>> tuple(l)
(1, 2, 3, 4)
>>> list("caterwaul")
['c', 'a', 't', 'e', 'r', 'w', 'a', 'u', 'l']
```

Programming Exercises

1. Cast all of the types we have seen so far to a string and see what happens.
2. Strings have a method called `join` that takes a list or tuple as an input. If you cast a string to a list, how can you use `join` to undo the action?

8 List Behaviors

Because they are mutable, lists exhibit some behaviors not available to tuples and strings. We have noted already that they behave differently when slices are taken. Here is a table of some of the most important operations and methods.

Operation	Action
`s[x] = y`	reassigns the value held by `s` at index `x` to `y`. List entries are lvalues.
`s[m:n] = [], del s[m:n]`	deletes all values between the indices `m` and `n` provided `m < n`.
`del s[a:b:c]`	deletes all elements of the indicated slice
`s.append(x)`	appends item `x` to the list `s`
`s.extend(t)`, s += t	appends items in the sequence `t` to the list `s`
`s.clear()`	empties the list `s`
`s.insert(i, t)`, `s[i:i]=t`	splice in the sequence `t` into the list `s`
`s.reverse()`	reverses the list in place.

9 Hashed Types

Python has two unordered types, dictionary (`dict`) and (`set`). A set is a container that does not admit duplicate entries, as defined by `==`. A dictionary is a container holding key-value pairs. These are made very efficient via a means called *hashing*. We will begin by describing this very clever mechanism. It makes access to items in sets and dictionaries fast and efficient.

9.1 What hashing? Why do it?

A *hash function* is a mathematical function whose domain is some collection of Python objects (e.g. strings) and whose codomain is the integers. Here we show the hashing function at work on strings.

```
>>> hash("a")
-8506767599803020586
>>> hash("b")
-7609782221146829849
>>> hash("c")
-5419851217699219052
>>> hash("ab")
-2244008983755709986
```

Here it is on integers. “Small" integers hash as themselves; once they get to a certain outrageous size, a truncation process occurs.

```
>>> hash(0)
0
>>> hash(1)
1
>>> hash(1024)
1024
>>> hash(1024576)
1024576
>>> hash(331214214214241)
331214214214241
>>> hash(413908140942980249081902)
97417085330101598
>>>
```

A perfect hash function will give different values to different objects. Hash functions depend on the state of the object they are hashing. Because mutable objects can have their state changed, this hashing process is not possible for them. Were a mutable object hashable, its hash would change whenever its state did. And, for the purposes we are about to describe here, that is very bad news.

One way we could implement a set us just to use a list and to reject the addition of duplicate elements. This causes inefficiency. Searching a list for an item is an $O(n)$ process, because the amount of resources it takes is at worst proportional to n, the size of the list. As the set got big, adding new elements would become burdensome.

So what do we do? The hash function provides the key. First, we reserve a big chunk of memory, say of size M. When we add an element we hash it, mod out by M and get an nonnegative integer less than M. We then store that object at that index in the chunk of memory (I lied... we store its memory address so we have access to it). So, to check for the presence of that object, we hash it and know precisely where it is stored. Hey, this is an $O(1)$ (constant-time) procedure.

The Nasty Hairy Fly in the Sweet Ointment Even if you have a perfect hash function (in practice never), this process of modding by M can cause a new item to be placed into an occupied slot. Beezlebub! Defeat!

Nah, what we do is store a little list of objects; we can go to that location and check that list. This kind of thing happens if the chunk of memory gets too crowded.

At some level of crowding, the whole thing will be put in a new, bigger, chunk of memory and everything will be rehashed to relieve the crowding.

Both sets and dictionaries achieve quick access of elements in this manner. Next, we will learn about sets mathematically and learn how they are implemented in Python.

10 Sets

Informally, in mathematics, a set is a collection of objects with along with a notion of belonging. In computer science, we are concerned with finite collections of objects, so some of the hairier aspects of axiomatic set theory will not come our way. However, we will have a brief discussion of some basic ideas of set theory and we will see how they are implemented in Python.

No meaningful discussion is possible in the absence of context. So, when we discuss sets, we will discuss them withing a *universe of discourse* which we will generally denote by Ω. Such a thing is often infinite. Example: the set of all ASCII characters strings. Or, perhaps the set of all integers. When we discuss sets, we will always have some universe of discourse in mind.

Inside of our universe of discourse, we can define sets two ways. One is by making an explicit list of elements. For example if $\Omega = \mathbb{Z}$, the set of all integers,

$$A = \{1, 6, -5, 4, 3\}$$

is a legitimate way to define a set. For belonging, we use the symbol $\in$. So, here $6 \in A$. To negate belonging, we use $\notin$; for example $100 \notin A$.

A set in our universe is well-defined if we can tell if any given element in the universe is or is not in the set. This leaves us another way to define a set, by using a predicate. For example let

$$E = \{x \in \Omega | x\%2 = 0\}$$

is the set of even numbers. Note we are using mathematical notation here; the Python form of this predicate is `x % 2 == 0`.

If A and B are sets in a universe Ω, we will write $A \subseteq B$ to mean that every element of A belongs to B. We define $A = B$ to mean $A \subseteq B$ and $B \subseteq A$. In other words, sets are equal if they contain exactly the same elements. We write $A \subset B$ to indicate that $A \subseteq B$, but that B has some element not in A and we say that A is a *proper subset* of B.

One implication of this definition is that the order in which elements of a set are presented is immaterial, and if you list an element twice, it is no different from having the element in once.

Now it's time for some nitty-gritty with Python. Let's put some animals on our farm in a list. We will then cast the list to a set.

```
>>> farm = ["sheep", "sheep", "cow", "horse",
        "pig", "goose", "pig", "cow"]
>>> animals = set(farm)
>>> animals
{'pig', 'horse', 'goose', 'sheep', 'cow'}
```

Notice that all duplicates got removed.

Take note that only hashable elements can be placed in a set. In this book, we will only put immutable objects in a set; these are always hashable. This restriction makes access to items in a set very fast, as we described at the beginning of this section.

Bear witness to this punishment dished out by an irate python when we try to hash the unhashable.

```
>>> hash("cow")
7419535498109472991
>>> hash([1,2,3])
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unhashable type: 'list'
```

Also, notice that items in a set seem to be presented in no seemingly discernible order.

Now let us see how Python implements these concepts. We begin with $\in$.

```
>>> "horse" in animals
True
>>> "rhino" in animals
False
```

We see that $x \in A$ if `x in A` is `True`, and $x \notin A$ otherwise.

Python also implements $\notin$. If `x not in A` is true, then $x \notin A$.

Now let's go for $\subseteq$.

```
>>> sample = {"goose", "cow"}
>>> animals.issubset(sample)
False
>>> sample.issubset(animals)
True
```

You can also do this; its a nice mnemonic.

```
>>> animals <= sample
False
>>> sample <= animals
True
```

The relational operator `<=` is the subset relation on sets. It has a strict version to indicate the "is a proper subset of" relation $\subset$.

```
>>> sample < animals
True
>>> animals < animals
False
>>>
```

Now we will see how boolean operations can be used in set-world. If A and B are sets in a universe Ω, we define the *complement* of A by

$$A^c = \{x \in \Omega | x \notin A\}.$$

The *union* of A and B is defined by

$$A \cup B = \{x \in \Omega | x \in A \vee x \in B\}.$$

Note that math uses $\vee$ for the infix binary `or` operator, and $\wedge$ for the infix `and` operator. It is easy to see that $A \cup B = B \cup A$, since this set is everything belonging to at least one of A or B.

We define the *intersection* of A and B by

$$A \cap B = \{x \in \Omega | x \in A \wedge x \in B\}.$$

To say it quickly, $A \cap B$ contains exactly all elements common to A and B. Think: street intersection, that which belongs to both streets.

Let us now see what Python has for union and intersection. There is no surprise here.

```
>>> zoo = {"rhino", "zebra", "sheep", "horse", "elephant"}
>>> zoo.intersection(animals)
{'sheep', 'horse'}
>>> zoo.union(animals)
{'horse', 'elephant', 'pig', 'zebra', 'goose', 'rhino',
    'sheep', 'cow'}
>>> zoo | animals
{'horse', 'sheep', 'rhino', 'pig', 'elephant', 'goose', 'cow', 'zebra'}
>>> zoo & animals
{'horse', 'sheep'}
```

There are two other set-theoretic operations that come in handy when handling data. There is the *relative complement* defined by

$$A - B = A \cap B^c = \{x \in \Omega | x \notin B\}.$$

This is the set of all things present in A not present in B. And there is the *symmetric difference*

$$A \triangle B = (A - B) \cup (B - A),$$

which consists of all elements belong to exactly one of A or B. Note the

$$A \triangle B = \{x \in \Omega | x \in A \oplus x \in B\},$$

where $\oplus$ is the infix exclusive or operator. Python handles this with aplomb.

```
>>> animals.symmetric_difference(zoo)
{'elephant', 'pig', 'zebra', 'goose', 'rhino', 'cow'}
>>> animals.difference(zoo)
{'cow', 'goose', 'pig'}
>>> zoo.difference(animals)
{'zebra', 'rhino', 'elephant'}
>>> animals ^ zoo
{'goose', 'rhino', 'elephant', 'pig', 'cow', 'zebra'}
>>> animals - zoo
{'goose', 'pig', 'cow'}
```

A set is a mutable object. It supports the `len` function, which will tell you how many elements it has.

Here is how to add new elements to a set.

```
>>> new_set=set()
>>> ne_set.add(1)
>>> ne_set.add(False)
>>> ne_set.add("cows")
>>> ne_set
{False, 1, 'cows'}
```

Now let's add a duplicate.

```
>>> ne_set.add("cows")
>>> ne_set
{False, 1, 'cows'}
```

The addition of the duplicate is ignored. To get rid of an element, use `discard`

```
>>> new_set.discard("cows")
>>> new_set
{False, 1}
```

Here is how to chuck everything.

```
>>> new_set.clear()
>>> new_set
set()
```

Programming Exercises

1. What does `is_disjoint()` do?
2. What does `pop` do? What is maddening about it?
3. Spelunking exercise: What types can you cast a set to? What types can you cast as a set? What happens in each case?
4. What happens if you try to index into a set?

11 Dictionaries

Imagine that you might want to have a list indexed by something other than numbers. For this purpose, Python features a second hashed data structure, the `dictionary`. Hashing, as we discussed before, gives rapid access to dictionary entries. Here we create a dictionary that stores telephone extensions. First we show how to create an empty dictionary.

```
>>> phone = {}
```

Now we show how to pre–populate a dictionary with a couple of entries.

```
>>> phone = {"morrison":2746, "yeh": 2725}
>>> phone["morrison"]
2746
```

Each dictionary entry consists of two parts. The first part is called the *key* and the second part is called the *value*. For any key `k`, its corresponding value is `phone[k]`. Shortly, we shall see that the value `phone[k]` is an lvalue. Because dictionary entries are retrieved via their keys, the keys of the dictionary must be hashable objects.

Notice the action of the `in` operator in a dictionary; this checks for membership in the keys.

```
>>> "morrison" in phone
True
>>> "sarocco" in phone
False
```

We can also check for the presence of a value.

```
>>> 2746 in phone.values()
True
>>> 2020 in phone.values()
False
>>>
```

It is very easy to add a new entry. Just associate a value with a key not present in the dictionary.

```
>>> phone["sarocco"] = 2722
>>> phone
{'yeh': 2725, 'sarocco': 2722, 'morrison': 2746}
>>> phone["miller"] = 2741
>>> phone
{'miller': 2741, 'yeh': 2725, 'sarocco': 2722, 'morrison': 2746}
```

You can get all of the key values in a list by using the keys() method. You can do a similar thing for getting all of the values in the dictionary.

```
>>> phone.keys()
dict.keys('miller', 'yeh', 'sarocco', 'morrison')
>>> phone.values()
dict.values([2741, 2725, 2722, 2746])
>>>
```

Finally you can change the value for any key as follows.

```
>>> len(phone)
3
```

So if you make an assignment `phone[foo] = blah`, the dictionary checks itself for the presence of `foo`; if `foo` is present, the value attached to it is changed to `blah`. If not, then the value `foo` is added to the keys and `blah` is assigned as its value.

Also, dictionaries know their size; just use `len`.

```
>>> len(phone)
>>> 4
```

12 Terminology Roundup

- **cast** This is a temporary request to regard an object to be regarded as another type
- **complement**A is a set, then the complement of A is the set of all elements in the universe of discourse not belonging to A.
- **disjoint** Two sets are disjoint if the have no elements in common.
- **garbage-collected language** these languages manage memory for you. Unused objects are automatically deallocated by a background process called the *garbage collector*
- **f-string** This is a format string. It is preceded by an `f` and can contain expressions that are surrounded by curly braces. These expressions are evaluated, converted into strings, and placed in evaluation of the f-string.
- **hash function** This is a function whose domain is a set of Python objects and whose codomain is the integers. A hash function is perfect if different objects always return different values.
- **intersection** The intersection of two sets is the set of all elements belonging to both of the sets.
- **keyword arguments** These are named arguments which are optional and which go at the end of a function's argument list.
- **lvalue** This is a symbol representing addressable memory. Variables, list items, and list slices are all lvalues. The term **object** simply refers to a datum stored in memory along with its associated code. Objects have three attributes, state (what an object knows), identity (an object's presence in memory), and behavior (what an object does).
- **proper subset** We say that A is a proper subset of B if every element of A belongs to B as well, and B contains at least one element not belonging to A.
- **relative complement** This is the set of all elements lying in one set but not another.
- **string** This refers to a contiguous piece of a sequence type. You can use the postfix [:::] operator to obtain a slice of a sequence
- **string pool** This is an area of memory in which small strings are kept and which is not garbage collected.
- **sentinel value** This is a return value for function that tells you something has gone wrong.
- **string** This is a character sequence.
- **subset** We say that A is a subset of B if every element of A belongs to B as well.
- **symmetric difference** The symmetric difference of two sets is the set of all element belonging to exactly one of the two sets.

- **type** This refers to the species of an object. Examples include integer, string, and boolean. The **union** The union of two sets is the set of all elements belonging to at least one of the sets.
- **universe of discourse** In set theory, this is the contextual bounds of the discussion; to wit, it is the set of all objects we speak of.

References

[1] I. Anaconda, *Anaconda download site*. `https://www.anaconda.com/products/individual`.

[2] N. Ninja, *The net ninja python 3*. `https://www.youtube.com/playlist?list=PL4cUxeGkcC9idu6GZ8EU_5B6WpKTdYZbK`.

[3] I. Python Foundation, *Python built-in types*. `https://docs.python.org/3/library/functions.html`.

[4] C. Schaefer, *Python tutorials*. `https://www.youtube.com/watch?v=YYXdXT2l-Gg&list=PL-osiE80TeTt2d9bfVyTiXJA-UTHn6WwU&ab_channel=CoreySchafer`.

[5] Wikipedia, *Duck typing*. `https://en.wikipedia.org/wiki/Duck_typing`.

[6] ——, *Two's complement*. `http://en.wikipedia.org/wiki/IEEE_754`.