Integers

In Python 3, there is effectively no limit to how long an integer value can be. Of course, it is constrained by the amount of memory your system has, as are all things, but beyond that an integer can be as long as you need it to be:

```
>>>
>>> print(12312312312312312312312312312312312312312312312312 + 1)
12312312312312312312312312312312312312312312312312312313
```

Python interprets a sequence of decimal digits without any prefix to be a decimal number:

```
>>>
>>> print(10)
10
```

Floating-Point Numbers

The `float` type in Python designates a floating-point number. `float` values are specified with a decimal point. Optionally, the character `e` or `E` followed by a positive or negative integer may be appended to specify scientific notation:

```
>>>
>>> 4.2
4.2
>>> type(4.2)
<class 'float'>
>>> 4.
4.0
>>> .2
0.2

>>> .4e7
4000000.0
>>> type(.4e7)
<class 'float'>
>>> 4.2e-4
0.00042
```

Strings

Strings are sequences of character data. The string type in Python is called `str`.

String literals may be delimited using either single or double quotes. All the characters between the opening delimiter and matching closing delimiter are part of the string:

```
>>>
>>> print("I am a string.")
I am a string.
>>> type("I am a string.")
<class 'str'>

>>> print('I am too.')
I am too.
>>> type('I am too.')
<class 'str'>
```

A string in Python can contain as many characters as you wish. The only limit is your machine's memory resources. A string can also be empty:

```
>>>
>>> ''
''
```

Escape Sequences in Strings
Sometimes, you want Python to interpret a character or sequence of characters within a string differently. This may occur in one of two ways:

- You may want to suppress the special interpretation that certain characters are usually given within a string.
- You may want to apply special interpretation to characters in a string which would normally be taken literally.

You can accomplish this using a backslash (`\`) character. A backslash character in a string indicates that one or more characters that follow it should be treated specially. (This is referred to as an escape sequence, because the

backslash causes the subsequent character sequence to “escape” its usual meaning.)

Boolean Type, Boolean Context, and “Truthiness”

Python 3 provides a Boolean data type. Objects of Boolean type may have one of two values, `True` or `False`:

```
>>>
>>> type(True)
<class 'bool'>
>>> type(False)
<class 'bool'>
```

As you will see in upcoming tutorials, expressions in Python are often evaluated in Boolean context, meaning they are interpreted to represent truth or falsehood. A value that is true in Boolean context is sometimes said to be “truthy,” and one that is false in Boolean context is said to be “falsy.” (You may also see “falsy” spelled “falsey.”)

Math

Function	Description
`abs()`	Returns absolute value of a number
`divmod()`	Returns quotient and remainder of integer division
`max()`	Returns the largest of the given arguments or items in an iterable
`min()`	Returns the smallest of the given arguments or items in an iterable
`pow()`	Raises a number to a power
`round()`	Rounds a floating-point value
`sum()`	Sums the items of an iterable

Type Conversion

Function	Description
`bool()`	Converts an argument to a Boolean value
`chr()`	Returns string representation of character given by integer argument
`complex()`	Returns a complex number constructed from arguments
`float()`	Returns a floating-point object constructed from a number or string
`hex()`	Converts an integer to a hexadecimal string
`int()`	Returns an integer object constructed from a number or string
`ord()`	Returns integer representation of a character
`repr()`	Returns a string containing a printable representation of an object

Function	Description
`str()`	Returns a string version of an object
`type()`	Returns the type of an object or creates a new type object

Input/Output

Function	Description
`format()`	Converts a value to a formatted representation
`input()`	Reads input from the console
`open()`	Opens a file and returns a file object
`print()`	Prints to a text stream or the console

Variable Assignment

Assignment is done with a single equals sign (=)

```
n = 300
```

Later, if you change the value of `n` and use it again, the new value will be substituted instead:

```
>>> n = 1000
>>> print(n)
1000
>>> n
1000
```

Python also allows chained assignment, which makes it possible to assign the same value to several variables simultaneously:

```
>>> a = b = c = 300
>>> print(a, b, c)
300 300 300
```

Variable Types in Python

In many programming languages, variables are statically typed. That means a variable is initially declared to have a specific data type, and any value assigned to it during its lifetime must always have that type.

Variables in Python are not subject to this restriction. In Python, a variable may be assigned a value of one type and then later re-assigned a value of a different type:

```
>>>
>>> var = 23.5
>>> print(var)
23.5

>>> var = "Now I'm a string"
>>> print(var)
Now I'm a string
```

You can see that an integer object is created using the built-in `type()` function:

```
>>> type(300)
<class 'int'>
```

Now consider the following statement:

```
>>> m = n
```

What happens when it is executed? Python does not create another object. It simply creates a new symbolic name or reference, `m`, which points to the same object that `n` points to.

Next, suppose you do this:

```
>>> m = 400
```

Now Python creates a new integer object with the value `400`, and `m` becomes a reference to it.

When the number of references to an object drops to zero, it is no longer accessible. At that point, its lifetime is over. Python will eventually notice that it is inaccessible and reclaim the allocated memory so it can be used for something else. In computer lingo, this process is referred to as garbage collection.

Object Identity

In Python, every object that is created is given a number that uniquely identifies it. It is guaranteed that no two objects will have the same identifier during any period in which their lifetimes overlap. Once an object's reference count drops to zero and it is garbage collected, as happened to the `300` object above, then its identifying number becomes available and may be used again.

The built-in Python function `id()` returns an object's integer identifier. Using the `id()`function, you can verify that two variables indeed point to the same object:

```
>>> n = 300
>>> m = n
>>> id(n)
60127840
>>> id(m)
60127840

>>> m = 400
>>> id(m)
60127872
```

Variable Names

The examples you have seen so far have used short, terse variable names like `m` and `n`. But variable names can be more verbose. In fact, it is usually beneficial if they are because it makes the purpose of the variable more evident at first glance.

Officially, variable names in Python can be any length and can consist of uppercase and lowercase letters (`A–Z`, `a–z`), digits (`0–9`), and the underscore character (_). An additional restriction is that, although a variable name can contain digits, the first character of a variable name cannot be a digit.

But this one is not, because a variable name can't begin with a digit:

```
>>> 1099_filed = False
```

Note that case is significant. Lowercase and uppercase letters are not the same. Use of the underscore character is significant as well. Each of the following defines a different variable:

```
>>> age = 1
>>> Age = 2
>>> aGe = 3
>>> AGE = 4
>>> a_g_e = 5
>>> _age = 6
>>> age_ = 7
>>> _AGE_ = 8
```

- **Camel Case:** Second and subsequent words are capitalized, to make word boundaries easier to see. (Presumably, it struck someone at some point that the capital letters strewn throughout the variable name vaguely resemble camel humps.)
    - Example: `numberOfCollegeGraduates`
- **Pascal Case:** Identical to Camel Case, except the first word is also capitalized.
    - Example: `NumberOfCollegeGraduates`
- **Snake Case:** Words are separated by underscores.
    - Example: `number_of_college_graduates`
- In Python 3.6, there are 33 reserved keywords:

Python Keywords			
`False`	`def`	`if`	`raise`
`None`	`del`	`import`	`return`
`True`	`elif`	`in`	`try`
`and`	`else`	`is`	`while`
`as`	`except`	`lambda`	`with`
`assert`	`finally`	`nonlocal`	`yield`
`break`	`for`	`not`	
`class`	`from`	`or`	
`continue`	`global`	`pass`	

Arithmetic Operators

The following table lists the arithmetic operators supported by Python:

Operator	Example	Meaning	Result
+ (unary)	`+a`	**Unary Positive**	`a` In other words, it doesn't really do anything. It mostly exists for the sake of completeness, to complement **Unary Negation**.
+ (binary)	`a + b`	**Addition**	Sum of `a` and `b`
– (unary)	`-a`	**Unary Negation**	Value equal to `a` but opposite in sign
– (binary)	`a - b`	**Subtraction**	`b` subtracted from `a`
`*`	`a * b`	**Multiplication**	Product of `a` and `b`
`/`	`a / b`	**Division**	Quotient when `a` is divided by `b`. The result always has type `float`.
`%`	`a % b`	**Modulo**	Remainder when `a` is divided by `b`
`//`	`a // b`	**Floor Division**(also called **Integer Division**)	Quotient when `a` is divided by `b`, rounded to the next smallest whole number
`**`	`a ** b`	**Exponentiation**	`a` raised to the power of `b`

Comparison Operators

Operator	Example	Meaning	Result
`==`	`a == b`	**Equal to**	`True` if the value of `a` is equal to the value of `b` `False` otherwise
`!=`	`a != b`	**Not equal to**	`True` if `a` is not equal to `b` `False` otherwise
`<`	`a < b`	**Less than**	`True` if `a` is less than `b` `False` otherwise
`<=`	`a <= b`	**Less than or equal to**	`True` if `a` is less than or equal to `b` `False` otherwise
`>`	`a > b`	**Greater than**	`True` if `a` is greater than `b` `False` otherwise
`>=`	`a >= b`	**Greater than or equal to**	`True` if `a` is greater than or equal to `b` `False` otherwise

Equality Comparison on Floating-Point Values

Recall from the earlier discussion of floating-point numbers that the value stored internally for a `float` object may not be precisely what you'd think it would be. For that reason, it is poor practice to compare floating-point values for exact equality. Consider this example:

```
>>> x = 1.1 + 2.2
```

```
>>> x == 3.3
False
```

Yikes! The internal representations of the addition operands are not exactly equal to `1.1` and `2.2`, so you cannot rely on `x` to compare exactly to `3.3`.

Interpretation of logical expressions involving `not`, `or`, and `and` is straightforward when the operands are Boolean:

Operator	Example	Meaning

`not`	`not x`	`True` if `x` is `False`
		`False` if `x` is `True`
		(Logically reverses the sense of `x`)

`or`	`x or y`	`True` if either `x` or `y` is `True` `False` otherwise
`and`	`x and y`	`True` if both `x` and `y` are `True` `False` otherwise

The "`None`" Keyword

`None` is always false:

```
>>> bool(None)
False
```

Bitwise Operators

Bitwise operators treat operands as sequences of binary digits and operate on them bit by bit. The following operators are supported:

Operator	Example	Meaning	Result
`&`	`a & b`	bitwise **AND**	Each bit position in the result is the logical **AND** of the bits in the corresponding position of the operands. (`1` if both are `1`, otherwise `0`.)
`|`	`a | b`	bitwise **OR**	Each bit position in the result is the logical **OR** of the bits in the corresponding position of the operands. (`1` if either is `1`, otherwise `0`.)
`~`	`~a`	bitwise **negation**	Each bit position in the result is the logical negation of the bit in the corresponding position of the operand. (`1` if `0`, `0` if `1`.)

Operator	Example	Meaning	Result
^	a ^ b	bitwise **XOR (exclusive OR)**	Each bit position in the result is the logical **XOR** of the bits in the corresponding position of the operands. (`1` if the bits in the operands are different, `0` if they are the same.)
>>	a >> n	**Shift right** n **places**	Each bit is shifted right n places.
<<	a << n	**Shift left** n **places**	Each bit is shifted left n places.

Identity Operators

Python provides two operators, `is` and `is not`, that determine whether the given operands have the same identity—that is, refer to the same object. This is not the same thing as equality, which means the two operands refer to objects that contain the same data but are not necessarily the same object.

```
>>> x = 1001
>>> y = 1000 + 1
>>> print(x, y)
1001 1001

>>> x == y
True
>>> x is y
False
```

Operator Precedence

Consider this expression:

```
>>> 20 + 4 * 10
60
```

Operator	Description	
lowest precedence	or	Boolean OR
	and	Boolean AND
	not	Boolean NOT

Operator	Description	
	`==`, `!=`, `<`, `<=`, `>`, `>=`, `is`, `is not`	comparisons, identity
	`|`	bitwise OR
	`^`	bitwise XOR
	`&`	bitwise AND
	`<<`, `>>`	bit shifts
	`+`, `-`	addition, subtraction
	`*`, `/`, `//`, `%`	multiplication, division, floor division, modulo
	`+x`, `-x`, `~x`	unary positive, unary negation, bitwise negation
highest precedence	`**`	exponentiation

Augmented Assignment		Standard Assignment
`a += 5`	is equivalent to	`a = a + 5`
`a /= 10`	is equivalent to	`a = a / 10`
`a ^= b`	is equivalent to	`a = a ^ b`

String Manipulation

The sections below highlight the operators, methods, and functions that are available for working with strings.

String Operators

You have already seen the operators `+` and `*` applied to numeric operands in the tutorial on Operators and Expressions in Python. These two operators can be applied to strings as well.

The + Operator

The `+` operator concatenates strings. It returns a string consisting of the operands joined together, as shown here:

```
>>>
>>> s = 'foo'
```

```
>>> t = 'bar'
>>> u = 'baz'

>>> s + t
'foobar'
>>> s + t + u
'foobarbaz'
```

*The * Operator*

The `*` operator creates multiple copies of a string. If `s` is a string and `n` is an integer, either of the following expressions returns a string consisting of `n` concatenated copies of `s`:

```
s * n
n * s
```

Here are examples of both forms:

```
>>>
>>> s = 'foo.'

>>> s * 4
'foo.foo.foo.foo.'
>>> 4 * s
'foo.foo.foo.foo.'
```

The `in` Operator

Python also provides a membership operator that can be used with strings. The `in` operator returns `True` if the first operand is contained within the second, and `False` otherwise:

```
>>>
>>> s = 'foo'

>>> s in 'That\'s food for thought.'
True
>>> s in 'That\'s good for now.'
False
```

Function	Description
`chr()`	Converts an integer to a character
`ord()`	Converts a character to an integer
`len()`	Returns the length of a string
`str()`	Returns a string representation of an object

String Indexing
Often in programming languages, individual items in an ordered set of data can be accessed directly using a numeric index or key value. This process is referred to as indexing.

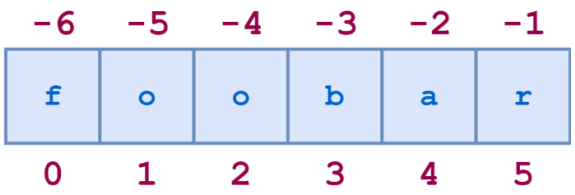
In Python, strings are ordered sequences of character data, and thus can be indexed in this way. Individual characters in a string can be accessed by specifying the string name followed by a number in square brackets (`[]`).

The individual characters can be accessed by index as follows:

```
>>> s = 'foobar'

>>> s[0]
'f'
>>> s[1]
'o'
>>> s[3]
'b'
>>> len(s)
6
>>> s[len(s)-1]
'r'
```

String indices can also be specified with negative numbers, in which case indexing occurs from the end of the string backward: `-1` refers to the last character, `-2` the second-to-last character, and so on. Here is the same diagram showing both the positive and negative indices into the string `'foobar'`:



String Slicing
Python also allows a form of indexing syntax that extracts substrings from a string, known as string slicing. If `s` is a string, an expression of the form `s[m:n]` returns the portion of `s`starting with position `m`, and up to but not including position `n`:

```
>>> s = 'foobar'
>>> s[2:5]
'oba'
```

Remember: String indices are zero-based. The first character in a string has index `0`. This applies to both standard indexing and slicing.

If you omit the first index, the slice starts at the beginning of the string. Thus, `s[:m]` and `s[0:m]` are equivalent:

```
>>> s = 'foobar'

>>> s[:4]
'foob'
>>> s[0:4]
'foob'
```

Specifying a Stride in a String Slice
There is one more variant of the slicing syntax to discuss. Adding an additional `:` and a third index designates a stride (also called a step), which indicates how many characters to jump after retrieving each character in the slice.

For example, for the string `'foobar'`, the slice `0:6:2` starts with the first character and ends with the last character (the whole string), and every second character is skipped.

```
>>> s = '12345' * 5
>>> s
```

```
'12345123451234512345123 45'
>>> s[::5]
'11111'
>>> s[4::5]
'55555'
```

You can specify a negative stride value as well, in which case Python steps backward through the string. In that case, the starting/first index should be greater than the ending/second index:

```
>>> s = 'foobar'
>>> s[5:0:-2]
'rbo'
```

Interpolating Variables Into a String

```
>>> n = 20
>>> m = 25
>>> prod = n * m
>>> print('The product of', n, 'and', m, 'is', prod)
The product of 20 and 25 is 500
```

Modifying Strings
In a nutshell, you can't. Strings are one of the data types Python considers immutable, meaning not able to be changed.

https://realpython.com/python-strings/#built-in-string-methods

Python Lists

In short, a list is a collection of arbitrary objects, somewhat akin to an array in many other programming languages but more flexible. Lists are defined in Python by enclosing a comma-separated sequence of objects in square brackets (`[]`), as shown below:

```
>>>
>>> a = ['foo', 'bar', 'baz', 'qux']
```

```
>>> print(a)
['foo', 'bar', 'baz', 'qux']
>>> a
['foo', 'bar', 'baz', 'qux']
```

The important characteristics of Python lists are as follows:

- Lists are ordered.
- Lists can contain any arbitrary objects.
- List elements can be accessed by index.
- Lists can be nested to arbitrary depth.
- Lists are mutable.
- Lists are dynamic.

Lists Are Ordered

A list is not merely a collection of objects. It is an ordered collection of objects. The order in which you specify the elements when you define a list is an innate characteristic of that list and is maintained for that list's lifetime. (You will see a Python data type that is not ordered in the next tutorial on dictionaries.)

Lists that have the same elements in a different order are not the same:

```
>>> a = ['foo', 'bar', 'baz', 'qux']
>>> b = ['baz', 'qux', 'bar', 'foo']
>>> a == b
False
>>> a is b
False

>>> [1, 2, 3, 4] == [4, 1, 3, 2]
False
```

Lists Can Contain Arbitrary Objects

A list can contain any assortment of objects. The elements of a list can all be the same type:

```
>>> a = [2, 4, 6, 8]
>>> a
[2, 4, 6, 8]
```

Or the elements can be of varying types:

```
>>> a = [21.42, 'foobar', 3, 4, 'bark', False, 3.14159]
>>> a
[21.42, 'foobar', 3, 4, 'bark', False, 3.14159]
```

List Elements Can Be Accessed by Index
Individual elements in a list can be accessed using an index in square brackets. This is exactly analogous to accessing individual characters in a string. List indexing is zero-based as it is with strings.

Consider the following list:

```
>>> a = ['foo', 'bar', 'baz', 'qux', 'quux', 'corge']
```

The syntax for reversing a list works the same way it does for strings:

```
>>> a[::-1]
['corge', 'quux', 'qux', 'baz', 'bar', 'foo']
```

Several Python operators and built-in functions can also be used with lists in ways that are analogous to strings:

- The `in` and `not in` operators:

  ```
  >>> a
  ['foo', 'bar', 'baz', 'qux', 'quux', 'corge']

  >>> 'qux' in a
  True
  >>> 'thud' not in a
  True
  ```

- The concatenation (+) and replication (*) operators:

  ```
  >>> a
  ['foo', 'bar', 'baz', 'qux', 'quux', 'corge']

  >>> a + ['grault', 'garply']
  ['foo', 'bar', 'baz', 'qux', 'quux', 'corge', 'grault', 'garply']
  ```

```
>>> a * 2
['foo', 'bar', 'baz', 'qux', 'quux', 'corge', 'foo', 'bar',
'baz',
'qux', 'quux', 'corge']
```

- The `len()`, `min()`, and `max()` functions:

```
>>> a
['foo', 'bar', 'baz', 'qux', 'quux', 'corge']

>>> len(a)
6
>>> min(a)
'bar'
>>> max(a)
'qux'
```

It's not an accident that strings and lists behave so similarly. They are both special cases of a more general object type called an iterable, which you will encounter in more detail in the upcoming tutorial on definite iteration.

Lists Can Be Nested

You have seen that an element in a list can be any sort of object. That includes another list. A list can contain sublists, which in turn can contain sublists themselves, and so on to arbitrary depth.

Consider this (admittedly contrived) example:

```
>>> x = ['a', ['bb', ['ccc', 'ddd'], 'ee', 'ff'], 'g', ['hh', 'ii'],
'j']
>>> x
['a', ['bb', ['ccc', 'ddd'], 'ee', 'ff'], 'g', ['hh', 'ii'], 'j']
```

To access the items in a sublist, simply append an additional index:

```
>>> x[1]
['bb', ['ccc', 'ddd'], 'ee', 'ff']

>>> x[1][0]
```

```
'bb'
>>> x[1][1]
['ccc', 'ddd']
>>> x[1][2]
'ee'
>>> x[1][3]
'ff'

>>> x[3]
['hh', 'ii']
>>> print(x[3][0], x[3][1])
hh ii
```

Lists Are Mutable

Most of the data types you have encountered so far have been atomic types. Integer or float objects, for example, are primitive units that can't be further broken down. These types are immutable, meaning that they can't be changed once they have been assigned. It doesn't make much sense to think of changing the value of an integer. If you want a different integer, you just assign a different one.

A list item can be deleted with the `del` command:

```
>>> a = ['foo', 'bar', 'baz', 'qux', 'quux', 'corge']

>>> del a[3]
>>> a
```

Python Tuples

Python provides another type that is an ordered collection of objects, called a tuple.

Pronunciation varies depending on whom you ask. Some pronounce it as though it were spelled "too-ple" (rhyming with "Mott the Hoople"), and others as though it were spelled "tup-ple" (rhyming with "supple"). My inclination is the latter, since it presumably derives from the same origin as "quintuple,"

“sextuple,” “octuple,” and so on, and everyone I know pronounces these latter as though they rhymed with “supple.”

Defining and Using Tuples
Tuples are identical to lists in all respects, except for the following properties:

- Tuples are defined by enclosing the elements in parentheses (`()`) instead of square brackets (`[]`).
- Tuples are immutable.
- Here is a short example showing a tuple definition, indexing, and slicing:

```
>>> t = ('foo', 'bar', 'baz', 'qux', 'quux', 'corge')
>>> t
('foo', 'bar', 'baz', 'qux', 'quux', 'corge')

>>> t[0]
'foo'
>>> t[-1]
'corge'
>>> t[1::2]
('bar', 'qux', 'corge')
```

Introduction to the `if` Statement

We’ll start by looking at the most basic type of `if` statement. In its simplest form, it looks like this:

```
if <expr>:
    <statement>
```

In the form shown above:

- `<expr>` is an expression evaluated in Boolean context, as discussed in the section on Logical Operators in the Operators and Expressions in Python tutorial.
- `<statement>` is a valid Python statement, which must be indented. (You will see why very soon.)

Python: It's All About the Indentation
Python follows a convention known as the off-side rule, a term coined by British computer scientist Peter J. Landin. (The term is taken from the offside law in association football.) Languages that adhere to the off-side rule define blocks by indentation. Python is one of a relatively small set of off-side rule languages.

Consider this script file `foo.py`:

```
1 if 'foo' in ['bar', 'baz', 'qux']:
2     print('Expression was true')
3     print('Executing statement in suite')
4     print('...')
5     print('Done.')
6 print('After conditional')
```

The `else` and `elif` Clauses

Now you know how to use an `if` statement to conditionally execute a single statement or a block of several statements. It's time to find out what else you can do.

Sometimes, you want to evaluate a condition and take one path if it is true but specify an alternative path if it is not. This is accomplished with an `else` clause:

```
if <expr>:
    <statement(s)>
else:
    <statement(s)>
```

Also

```
if <expr>:
    <statement(s)>
elif <expr>:
    <statement(s)>
elif <expr>:
    <statement(s)>
```

```
    ...
else:
    <statement(s)>
```

The `pass` Statement

Occasionally, you may find that you want to write what is called a code stub: a placeholder for where you will eventually put a block of code that you haven't implemented yet.

Classes in Python

Focusing first on the data, each thing or object is an instance of some *class*.

The primitive data structures available in Python, like numbers, strings, and lists are designed to represent simple things like the cost of something, the name of a poem, and your favorite colors, respectively.

What if you wanted to represent something much more complicated?

For example, let's say you wanted to track a number of different animals. If you used a list, the first element could be the animal's name while the second element could represent its age.

How would you know which element is supposed to be which? What if you had 100 different animals? Are you certain each animal has both a name and an age, and so forth? What if you wanted to add other properties to these animals? This lacks organization, and it's the exact need for *classes*.

Classes are used to create new user-defined data structures that contain arbitrary information about something. In the case of an animal, we could create an `Animal()` class to track properties about the Animal like the name and age.

Python Objects (Instances)

While the class is the blueprint, an *instance* is a copy of the class with *actual* values, literally an object belonging to a specific class. It's not an idea anymore; it's an actual animal, like a dog named Roger who's eight years old.

Instance Attributes
All classes create objects, and all objects contain characteristics called attributes (referred to as properties in the opening paragraph). Use the `__init__()` method to initialize (e.g., specify) an object's initial attributes by giving them their default value (or state). This method must have at least one argument as well as the `self` variable, which refers to the object itself (e.g., Dog).

Instance Methods

Instance methods are defined inside a class and are used to get the contents of an instance. They can also be used to perform operations with the attributes of our objects. Like the `__init__` method, the first argument is always `self`:

```
class Dog:

    # Class attribute
    species = 'mammal'

    # Initializer / Instance attributes
    def __init__(self, name, age):
        self.name = name
        self.age = age

    # instance method
    def description(self):
        return "{} is {} years old".format(self.name, self.age)

    # instance method
    def speak(self, sound):
        return "{} says {}".format(self.name, sound)


# Child class (inherits from Dog class)
class RussellTerrier(Dog):
    def run(self, speed):
        return "{} runs {}".format(self.name, speed)
```

```
# Child class (inherits from Dog class)
class Bulldog(Dog):
    def run(self, speed):
        return "{} runs {}".format(self.name, speed)
```

Docstring Types

Docstring conventions are described within PEP 257. Their purpose is to provide your users with a brief overview of the object. They should be kept concise enough to be easy to maintain but still be elaborate enough for new users to understand their purpose and how to use the documented object.

In all cases, the docstrings should use the triple-double quote (""") string format. This should be done whether the docstring is multi-lined or not. At a bare minimum, a docstring should be a quick summary of whatever is it you're describing and should be contained within a single line:

```
"""This is a quick summary line used as a description of the
object."""
```

Class Docstrings

Class Docstrings are created for the class itself, as well as any class methods. The docstrings are placed immediately following the class or class method indented by one level:

```
class SimpleClass:
    """Class docstrings go here."""

    def say_hello(self, name: str):
        """Class method docstrings go here."""

        print(f'Hello {name}')
```

Class docstrings should contain the following information:

- A brief summary of its purpose and behavior
- Any public methods, along with a brief description
- Any class properties (attributes)
- Anything related to the interface for subclassers, if the class is intended to be subclassed

Defining a Dictionary

Dictionaries are Python's implementation of a data structure that is more generally known as an associative array. A dictionary consists of a collection of key-value pairs. Each key-value pair maps the key to its associated value.

You can define a dictionary by enclosing a comma-separated list of key-value pairs in curly braces (`{}`). A colon (`:`) separates each key from its associated value:

```
d = {
    <key>: <value>,
    <key>: <value>,
      .
      .
      .
    <key>: <value>
}
```

The following defines a dictionary that maps a location to the name of its corresponding Major League Baseball team:

```
>>> MLB_team = {
...     'Colorado' : 'Rockies',
...     'Boston'   : 'Red Sox',
...     'Minnesota': 'Twins',
...     'Milwaukee': 'Brewers',
...     'Seattle'  : 'Mariners'
... }
```

You can also construct a dictionary with the built-in `dict()` function. The argument to `dict()` should be a sequence of key-value pairs. A list of tuples works well for this:

```
d = dict([
    (<key>, <value>),
    (<key>, <value),
      .
      .
      .
    (<key>, <value>)
])
```

Accessing Dictionary Values

Of course, dictionary elements must be accessible somehow. If you don't get them by index, then how do you get them?

A value is retrieved from a dictionary by specifying its corresponding key in square brackets (`[]`):

```
>>>
```

```
>>> MLB_team['Minnesota']
'Twins'
>>> MLB_team['Colorado']
'Rockies'
```

Operators and Built-in Functions

You have already become familiar with many of the operators and built-in functions that can be used with strings, lists, and tuples. Some of these work with dictionaries as well.

For example, the `in` and `not in` operators return `True` or `False` according to whether the specified operand occurs as a key in the dictionary:

```
>>> MLB_team = {
...     'Colorado' : 'Rockies',
...     'Boston'   : 'Red Sox',
...     'Minnesota': 'Twins',
...     'Milwaukee': 'Brewers',
...     'Seattle'  : 'Mariners'
```

```
... }

>>> 'Milwaukee' in MLB_team
True
>>> 'Toronto' in MLB_team
False
>>> 'Toronto' not in MLB_team
True
```

You can use the `in` operator together with short-circuit evaluation to avoid raising an error when trying to access a key that is not in the dictionary:

```
>>>
>>> MLB_team['Toronto']
Traceback (most recent call last):
  File "<pyshell#2>", line 1, in <module>
    MLB_team['Toronto']
KeyError: 'Toronto'

>>> 'Toronto' in MLB_team and MLB_team['Toronto']
False
```