```
# Single line comments start with a number symbol.

""" Multiline strings can be written
    using three "s, and are often used
    as documentation.
"""

####################################################
## 1. Primitive Datatypes and Operators
####################################################

# You have numbers
3  # => 3

# Math is what you would expect
1 + 1   # => 2
8 - 1   # => 7
10 * 2  # => 20
35 / 5  # => 7.0

# Integer division rounds down for both positive and negative numbers.
5 // 3       # => 1
-5 // 3      # => -2
5.0 // 3.0   # => 1.0 # works on floats too
-5.0 // 3.0  # => -2.0

# The result of division is always a float
10.0 / 3  # => 3.3333333333333335

# Modulo operation
7 % 3  # => 1

# Exponentiation (x**y, x to the yth power)
2**3  # => 8

# Enforce precedence with parentheses
1 + 3 * 2  # => 7
(1 + 3) * 2  # => 8

# Boolean values are primitives (Note: the capitalization)
True  # => True
False  # => False

# negate with not
not True   # => False
not False  # => True

# Boolean Operators
# Note "and" and "or" are case-sensitive
True and False  # => False
False or True   # => True

# True and False are actually 1 and 0 but with different keywords
True + True # => 2
True * 8    # => 8
False - 5   # => -5

# Comparison operators look at the numerical value of True and False
0 == False  # => True
1 == True   # => True
```

```
2 == True   # => False
-5 != False # => True

# Using boolean logical operators on ints casts them to booleans for evaluation, but their non-cast value is returned
# Don't mix up with bool(ints) and bitwise and/or (&,|)
bool(0)     # => False
bool(4)     # => True
bool(-6)    # => True
0 and 2     # => 0
-5 or 0     # => -5

# Equality is ==
1 == 1  # => True
2 == 1  # => False

# Inequality is !=
1 != 1  # => False
2 != 1  # => True

# More comparisons
1 < 10  # => True
1 > 10  # => False
2 <= 2  # => True
2 >= 2  # => True

# Seeing whether a value is in a range
1 < 2 and 2 < 3  # => True
2 < 3 and 3 < 2  # => False
# Chaining makes this look nicer
1 < 2 < 3  # => True
2 < 3 < 2  # => False

# (is vs. ==) is checks if two variables refer to the same object, but == checks
# if the objects pointed to have the same values.
a = [1, 2, 3, 4]  # Point a at a new list, [1, 2, 3, 4]
b = a             # Point b at what a is pointing to
b is a            # => True, a and b refer to the same object
b == a            # => True, a's and b's objects are equal
b = [1, 2, 3, 4]  # Point b at a new list, [1, 2, 3, 4]
b is a            # => False, a and b do not refer to the same object
b == a            # => True, a's and b's objects are equal

# Strings are created with " or '
"This is a string."
'This is also a string.'

# Strings can be added too! But try not to do this.
"Hello " + "world!"  # => "Hello world!"
# String literals (but not variables) can be concatenated without using '+'
"Hello " "world!"    # => "Hello world!"

# A string can be treated like a list of characters
"This is a string"[0]  # => 'T'

# You can find the length of a string
len("This is a string")  # => 16

# You can also format using f-strings or formatted string literals (in Python 3.6+)
name = "Reiko"
f"She said her name is {name}." # => "She said her name is Reiko"
```

```
# You can basically put any Python statement inside the braces and it will be output in the string.
f"{name} is {len(name)} characters long." # => "Reiko is 5 characters long."


# None is an object
None  # => None

# Don't use the equality "==" symbol to compare objects to None
# Use "is" instead. This checks for equality of object identity.
"etc" is None  # => False
None is None   # => True

# None, 0, and empty strings/lists/dicts/tuples all evaluate to False.
# All other values are True
bool(0)   # => False
bool("")  # => False
bool([])  # => False
bool({})  # => False
bool(())  # => False

####################################################
## 2. Variables and Collections
####################################################

# Python has a print function
print("I'm Python. Nice to meet you!")  # => I'm Python. Nice to meet you!

# By default the print function also prints out a newline at the end.
# Use the optional argument end to change the end string.
print("Hello, World", end="!")  # => Hello, World!

# Simple way to get input data from console
input_string_var = input("Enter some data: ") # Returns the data as a string
# Note: In earlier versions of Python, input() method was named as raw_input()

# There are no declarations, only assignments.
# Convention is to use lower_case_with_underscores
some_var = 5
some_var  # => 5

# Accessing a previously unassigned variable is an exception.
# See Control Flow to learn more about exception handling.
some_unknown_var  # Raises a NameError

# if can be used as an expression
# Equivalent of C's '?:' ternary operator
"yahoo!" if 3 > 2 else 2  # => "yahoo!"

# Lists store sequences
li = []
# You can start with a prefilled list
other_li = [4, 5, 6]

# Add stuff to the end of a list with append
li.append(1)    # li is now [1]
li.append(2)    # li is now [1, 2]
li.append(4)    # li is now [1, 2, 4]
li.append(3)    # li is now [1, 2, 4, 3]
# Remove from the end with pop
li.pop()        # => 3 and li is now [1, 2, 4]
```

```
# Let's put it back
li.append(3)    # li is now [1, 2, 4, 3] again.

# Access a list like you would any array
li[0]   # => 1
# Look at the last element
li[-1]  # => 3

# Looking out of bounds is an IndexError
li[4]  # Raises an IndexError

# You can look at ranges with slice syntax.
# The start index is included, the end index is not
# (It's a closed/open range for you mathy types.)
li[1:3]   # Return list from index 1 to 3 => [2, 4]
li[2:]    # Return list starting from index 2 => [4, 3]
li[:3]    # Return list from beginning until index 3  => [1, 2, 4]
li[::2]   # Return list selecting every second entry => [1, 4]
li[::-1]  # Return list in reverse order => [3, 4, 2, 1]
# Use any combination of these to make advanced slices
# li[start:end:step]

# Make a one layer deep copy using slices
li2 = li[:]  # => li2 = [1, 2, 4, 3] but (li2 is li) will result in false.

# Remove arbitrary elements from a list with "del"
del li[2]  # li is now [1, 2, 3]

# Remove first occurrence of a value
li.remove(2)  # li is now [1, 3]
li.remove(2)  # Raises a ValueError as 2 is not in the list

# Insert an element at a specific index
li.insert(1, 2)  # li is now [1, 2, 3] again

# Get the index of the first item found matching the argument
li.index(2)  # => 1
li.index(4)  # Raises a ValueError as 4 is not in the list

# You can add lists
# Note: values for li and for other_li are not modified.
li + other_li  # => [1, 2, 3, 4, 5, 6]

# Concatenate lists with "extend()"
li.extend(other_li)  # Now li is [1, 2, 3, 4, 5, 6]

# Check for existence in a list with "in"
1 in li  # => True

# Examine the length with "len()"
len(li)  # => 6


# Tuples are like lists but are immutable.
tup = (1, 2, 3)
tup[0]      # => 1
tup[0] = 3  # Raises a TypeError

# Note that a tuple of length one has to have a comma after the last element but
# tuples of other lengths, even zero, do not.
```

```
type((1))   # => <class 'int'>
type((1,))  # => <class 'tuple'>
type(())    # => <class 'tuple'>

# You can do most of the list operations on tuples too
len(tup)         # => 3
tup + (4, 5, 6)  # => (1, 2, 3, 4, 5, 6)
tup[:2]          # => (1, 2)
2 in tup         # => True

# You can unpack tuples (or lists) into variables
a, b, c = (1, 2, 3)  # a is now 1, b is now 2 and c is now 3
# You can also do extended unpacking
a, *b, c = (1, 2, 3, 4)  # a is now 1, b is now [2, 3] and c is now 4
# Tuples are created by default if you leave out the parentheses
d, e, f = 4, 5, 6  # tuple 4, 5, 6 is unpacked into variables d, e and f
# respectively such that d = 4, e = 5 and f = 6
# Now look how easy it is to swap two values
e, d = d, e  # d is now 5 and e is now 4


# Dictionaries store mappings from keys to values
empty_dict = {}
# Here is a prefilled dictionary
filled_dict = {"one": 1, "two": 2, "three": 3}

# Note keys for dictionaries have to be immutable types. This is to ensure that
# the key can be converted to a constant hash value for quick look-ups.
# Immutable types include ints, floats, strings, tuples.
invalid_dict = {[1,2,3]: "123"}  # => Raises a TypeError: unhashable type: 'list'
valid_dict = {(1,2,3):[1,2,3]}   # Values can be of any type, however.

# Look up values with []
filled_dict["one"]  # => 1

# Get all keys as an iterable with "keys()". We need to wrap the call in list()
# to turn it into a list. We'll talk about those later.  Note - for Python
# versions <3.7, dictionary key ordering is not guaranteed. Your results might
# not match the example below exactly. However, as of Python 3.7, dictionary
# items maintain the order at which they are inserted into the dictionary.
list(filled_dict.keys())  # => ["three", "two", "one"] in Python <3.7
list(filled_dict.keys())  # => ["one", "two", "three"] in Python 3.7+


# Get all values as an iterable with "values()". Once again we need to wrap it
# in list() to get it out of the iterable. Note - Same as above regarding key
# ordering.
list(filled_dict.values())  # => [3, 2, 1]  in Python <3.7
list(filled_dict.values())  # => [1, 2, 3] in Python 3.7+

# Check for existence of keys in a dictionary with "in"
"one" in filled_dict  # => True
1 in filled_dict      # => False

# Looking up a non-existing key is a KeyError
filled_dict["four"]  # KeyError

# Use "get()" method to avoid the KeyError
filled_dict.get("one")      # => 1
filled_dict.get("four")     # => None
```

```
# The get method supports a default argument when the value is missing
filled_dict.get("one", 4)   # => 1
filled_dict.get("four", 4)  # => 4

# "setdefault()" inserts into a dictionary only if the given key isn't present
filled_dict.setdefault("five", 5)  # filled_dict["five"] is set to 5
filled_dict.setdefault("five", 6)  # filled_dict["five"] is still 5

# Adding to a dictionary
filled_dict.update({"four":4})  # => {"one": 1, "two": 2, "three": 3, "four": 4}
filled_dict["four"] = 4         # another way to add to dict

# Remove keys from a dictionary with del
del filled_dict["one"]  # Removes the key "one" from filled dict

# From Python 3.5 you can also use the additional unpacking options
{'a': 1, **{'b': 2}}  # => {'a': 1, 'b': 2}
{'a': 1, **{'a': 2}}  # => {'a': 2}



# Sets store ... well sets
empty_set = set()
# Initialize a set with a bunch of values. Yeah, it looks a bit like a dict. Sorry.
some_set = {1, 1, 2, 2, 3, 4}  # some_set is now {1, 2, 3, 4}

# Similar to keys of a dictionary, elements of a set have to be immutable.
invalid_set = {[1], 1}  # => Raises a TypeError: unhashable type: 'list'
valid_set = {(1,), 1}

# Add one more item to the set
filled_set = some_set
filled_set.add(5)  # filled_set is now {1, 2, 3, 4, 5}
# Sets do not have duplicate elements
filled_set.add(5)  # it remains as before {1, 2, 3, 4, 5}

# Do set intersection with &
other_set = {3, 4, 5, 6}
filled_set & other_set  # => {3, 4, 5}

# Do set union with |
filled_set | other_set  # => {1, 2, 3, 4, 5, 6}

# Do set difference with -
{1, 2, 3, 4} - {2, 3, 5}  # => {1, 4}

# Do set symmetric difference with ^
{1, 2, 3, 4} ^ {2, 3, 5}  # => {1, 4, 5}

# Check if set on the left is a superset of set on the right
{1, 2} >= {1, 2, 3} # => False

# Check if set on the left is a subset of set on the right
{1, 2} <= {1, 2, 3} # => True

# Check for existence in a set with in
2 in filled_set   # => True
10 in filled_set  # => False

# Make a one layer deep copy
```

```
filled_set = some_set.copy()  # filled_set is {1, 2, 3, 4, 5}
filled_set is some_set        # => False


####################################################
## 3. Control Flow and Iterables
####################################################

# Let's just make a variable
some_var = 5

# Here is an if statement. Indentation is significant in Python!
# Convention is to use four spaces, not tabs.
# This prints "some_var is smaller than 10"
if some_var > 10:
    print("some_var is totally bigger than 10.")
elif some_var < 10:    # This elif clause is optional.
    print("some_var is smaller than 10.")
else:                  # This is optional too.
    print("some_var is indeed 10.")


"""
For loops iterate over lists
prints:
    dog is a mammal
    cat is a mammal
    mouse is a mammal
"""
for animal in ["dog", "cat", "mouse"]:
    # You can use format() to interpolate formatted strings
    print("{} is a mammal".format(animal))

"""
"range(number)" returns an iterable of numbers
from zero to the given number
prints:
    0
    1
    2
    3
"""
for i in range(4):
    print(i)

"""
"range(lower, upper)" returns an iterable of numbers
from the lower number to the upper number
prints:
    4
    5
    6
    7
"""
for i in range(4, 8):
    print(i)

"""
"range(lower, upper, step)" returns an iterable of numbers
from the lower number to the upper number, while incrementing
```

```
by step. If step is not indicated, the default value is 1.
prints:
   4
   6
"""
for i in range(4, 8, 2):
   print(i)

"""
To loop over a list, and retrieve both the index and the value of each item in the list
prints:
   0 dog
   1 cat
   2 mouse
"""
animals = ["dog", "cat", "mouse"]
for i, value in enumerate(animals):
   print(i, value)

"""
While loops go until a condition is no longer met.
prints:
   0
   1
   2
   3
"""
x = 0
while x < 4:
   print(x)
   x += 1  # Shorthand for x = x + 1

# Handle exceptions with a try/except block
try:
   # Use "raise" to raise an error
   raise IndexError("This is an index error")
except IndexError as e:
   pass              # Pass is just a no-op. Usually you would do recovery here.
except (TypeError, NameError):
   pass              # Multiple exceptions can be handled together, if required.
else:                # Optional clause to the try/except block. Must follow all except blocks
   print("All good!")   # Runs only if the code in try raises no exceptions
finally:             #  Execute under all circumstances
   print("We can clean up resources here")

# Instead of try/finally to cleanup resources you can use a with statement
with open("myfile.txt") as f:
   for line in f:
      print(line)

# Writing to a file
contents = {"aa": 12, "bb": 21}
with open("myfile1.txt", "w+") as file:
   file.write(str(contents))       # writes a string to a file

with open("myfile2.txt", "w+") as file:
   file.write(json.dumps(contents)) # writes an object to a file

# Reading from a file
with open('myfile1.txt', "r+") as file:
```

```
    contents = file.read()          # reads a string from a file
print(contents)
# print: {"aa": 12, "bb": 21}

with open('myfile2.txt', "r+") as file:
    contents = json.load(file)      # reads a json object from a file
print(contents)
# print: {"aa": 12, "bb": 21}


# Python offers a fundamental abstraction called the Iterable.
# An iterable is an object that can be treated as a sequence.
# The object returned by the range function, is an iterable.

filled_dict = {"one": 1, "two": 2, "three": 3}
our_iterable = filled_dict.keys()
print(our_iterable)  # => dict_keys(['one', 'two', 'three']). This is an object that implements our Iterable interface.

# We can loop over it.
for i in our_iterable:
    print(i)  # Prints one, two, three

# However we cannot address elements by index.
our_iterable[1]  # Raises a TypeError

# An iterable is an object that knows how to create an iterator.
our_iterator = iter(our_iterable)

# Our iterator is an object that can remember the state as we traverse through it.
# We get the next object with "next()".
next(our_iterator)  # => "one"

# It maintains state as we iterate.
next(our_iterator)  # => "two"
next(our_iterator)  # => "three"

# After the iterator has returned all of its data, it raises a StopIteration exception
next(our_iterator)  # Raises StopIteration

# We can also loop over it, in fact, "for" does this implicitly!
our_iterator = iter(our_iterable)
for i in our_iterator:
    print(i)  # Prints one, two, three

# You can grab all the elements of an iterable or iterator by calling list() on it.
list(our_iterable)  # => Returns ["one", "two", "three"]
list(our_iterator)  # => Returns [] because state is saved


####################################################
## 4. Functions
####################################################

# Use "def" to create new functions
def add(x, y):
    print("x is {} and y is {}".format(x, y))
    return x + y  # Return values with a return statement

# Calling functions with parameters
add(5, 6)  # => prints out "x is 5 and y is 6" and returns 11
```

```
# Another way to call functions is with keyword arguments
add(y=6, x=5)  # Keyword arguments can arrive in any order.

# You can define functions that take a variable number of
# positional arguments
def varargs(*args):
    return args

varargs(1, 2, 3)  # => (1, 2, 3)

# You can define functions that take a variable number of
# keyword arguments, as well
def keyword_args(**kwargs):
    return kwargs

# Let's call it to see what happens
keyword_args(big="foot", loch="ness")  # => {"big": "foot", "loch": "ness"}


# You can do both at once, if you like
def all_the_args(*args, **kwargs):
    print(args)
    print(kwargs)
"""
all_the_args(1, 2, a=3, b=4) prints:
    (1, 2)
    {"a": 3, "b": 4}
"""

# When calling functions, you can do the opposite of args/kwargs!
# Use * to expand tuples and use ** to expand kwargs.
args = (1, 2, 3, 4)
kwargs = {"a": 3, "b": 4}
all_the_args(*args)            # equivalent to all_the_args(1, 2, 3, 4)
all_the_args(**kwargs)         # equivalent to all_the_args(a=3, b=4)
all_the_args(*args, **kwargs)  # equivalent to all_the_args(1, 2, 3, 4, a=3, b=4)

# Returning multiple values (with tuple assignments)
def swap(x, y):
    return y, x  # Return multiple values as a tuple without the parenthesis.
                 # (Note: parenthesis have been excluded but can be included)

x = 1
y = 2
x, y = swap(x, y)     # => x = 2, y = 1
# (x, y) = swap(x,y)  # Again parenthesis have been excluded but can be included.

# Function Scope
x = 5

def set_x(num):
    # Local var x not the same as global variable x
    x = num    # => 43
    print(x)   # => 43

def set_global_x(num):
    global x
    print(x)   # => 5
    x = num    # global var x is now set to 6
```

```
    print(x)   # => 6

set_x(43)
set_global_x(6)


# Python has first class functions
def create_adder(x):
    def adder(y):
        return x + y
    return adder

add_10 = create_adder(10)
add_10(3)   # => 13

# There are also anonymous functions
(lambda x: x > 2)(3)                # => True
(lambda x, y: x ** 2 + y ** 2)(2, 1)  # => 5

# There are built-in higher order functions
list(map(add_10, [1, 2, 3]))         # => [11, 12, 13]
list(map(max, [1, 2, 3], [4, 2, 1]))  # => [4, 2, 3]

list(filter(lambda x: x > 5, [3, 4, 5, 6, 7]))  # => [6, 7]

# We can use list comprehensions for nice maps and filters
# List comprehension stores the output as a list which can itself be a nested list
[add_10(i) for i in [1, 2, 3]]         # => [11, 12, 13]
[x for x in [3, 4, 5, 6, 7] if x > 5]  # => [6, 7]

# You can construct set and dict comprehensions as well.
{x for x in 'abcddeef' if x not in 'abc'}  # => {'d', 'e', 'f'}
{x: x**2 for x in range(5)}  # => {0: 0, 1: 1, 2: 4, 3: 9, 4: 16}


####################################################
## 5. Modules
####################################################

# You can import modules
import math
print(math.sqrt(16))  # => 4.0

# You can get specific functions from a module
from math import ceil, floor
print(ceil(3.7))   # => 4.0
print(floor(3.7))  # => 3.0

# You can import all functions from a module.
# Warning: this is not recommended
from math import *

# You can shorten module names
import math as m
math.sqrt(16) == m.sqrt(16)  # => True

# Python modules are just ordinary Python files. You
# can write your own, and import them. The name of the
# module is the same as the name of the file.
```

```
# You can find out which functions and attributes
# are defined in a module.
import math
dir(math)

# If you have a Python script named math.py in the same
# folder as your current script, the file math.py will
# be loaded instead of the built-in Python module.
# This happens because the local folder has priority
# over Python's built-in libraries.
```