Python

Python

- Python is a high-level, dynamically typed multiparadigm programming language, created by Guido van Rossum in the early 90s. It is now one of the most popular languages in existence.
- Python's syntactic clarity allows you to express very powerful ideas in very few lines of code while being very readable. It's basically **executable pseudocode!**
- As an example, here is an implementation of the classic Quicksort algorithm in Python:

```
def quicksort(arr):
    if len(arr) <= 1:
        return arr
    pivot = arr[len(arr) // 2]
    left = [x for x in arr if x < pivot]
    middle = [x for x in arr if x == pivot]
    right = [x for x in arr if x > pivot]
    return quicksort(left) + middle + quicksort(right)

quicksort([3, 6, 8, 10, 1, 2, 1]) # Returns "[1, 1, 2, 3, 6, 8, 10]"
```

> Note: This primer applies to Python 3 specifically. Check out the Python 2 primer if you want to learn about the (now old) Python 2.7.

Python Versions

- If you haven't experimented with Python at all and are just starting off, we recommend you begin with the latest version of Python 3.
- You can double-check your Python version at the command line after activating your environment by running `python --version`.

Python Style Guide

- Google’s Python Style Guide is a fantastic resource with a list of dos and don’ts for formatting Python code that is commonly followed in the industry.

Python Notebooks: Jupyter and Colab

- Before we dive into Python, we’d like to briefly talk about **notebooks** and **creating virtual environments**.
- If you’re looking to work on different projects, you’ll likely be utilizing different versions of Python modules. In this case, it is a good practice to have multiple virtual environments to work on different projects.
- Python Setup: Remote vs. Local offers an in-depth coverage of the various remote and local options available.

The Zen of Python

- The Zen of Python by Tim Peters are 19 guidelines for the design of the Python language. Your Python code doesn’t necessarily have to follow these guidelines, but they’re good to keep in mind. The Zen of Python is an Easter egg, or hidden joke, that appears if you run:
- which outputs:

```
The Zen of Python, by Tim Peters

Beautiful is better than ugly.
Explicit is better than implicit.
Simple is better than complex.
Complex is better than complicated.
Flat is better than nested.
Sparse is better than dense.
Readability counts.
Special cases aren't special enough to break the rules.
Although practicality beats purity.
Errors should never pass silently.
Unless explicitly silenced.
In the face of ambiguity, refuse the temptation to guess.
There should be one-- and preferably only one --obvious way to do it.
Although that way may not be obvious at first unless you're Dutch.
Now is better than never.
Although never is often better than *right* now.
If the implementation is hard to explain, it's a bad idea.
If the implementation is easy to explain, it may be a good idea.
Namespaces are one honking great idea -- let's do more of those!
```

- For more on the coding principles above, refer to The Zen of Python, Explained.

Indentation

- Python is big on indentation! Where in other programming languages the indentation in code is to improve readability, Python uses indentation to indicate a block of code.
- The convention is to use **four spaces**, **not tabs**.

Code Comments

- Python supports two types of comments: single-line and multi-line, as detailed below:

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

""" Multiline strings can be written using three double quotes,
    and are often used for documentation (hence called "docstrings").
    They are also the closest concept to multi-line comments in other languages.
"""
```

Variables

- In Python, there are no declarations unlike C/C++; only assignments:
- Accessing a previously unassigned variable leads to a `NameError` exception:

```
some_unknown_var # Raises a NameError
```

- Variables are “names” in Python that simply refer to objects. This implies that you can make another variable point to an object by assigning it the original variable that was pointing to the same object.

```
a = [1, 2, 3] # Point "a" at a new list, [1, 2, 3, 4]
b = a         # Point "b" at what "a" is pointing to
b += [4]      # Extend the list pointed by "a" and "b" by adding "4" to it
```

```
a             # Returns "[1, 2, 3, 4]"
b             # Also returns "[1, 2, 3, 4]"
```

- `is` checks if two variables refer to the same object, while `==` checks if the objects that the variables point 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           # Returns True, "a" and "b" refer to the *same* object
b == a           # Returns True, the objects that "a" and "b" are pointing to are *equal*

b = [1, 2, 3, 4] # Point b at a new list, [1, 2, 3, 4]
b is a           # Returns False, "a" and "b" do not refer to the *same* object
b == a           # Returns True, the objects that "a" and "b" are pointing to are *equal*
```

- In Python, everything is an object. This means that even `None` (which is used to denote that the variable doesn't point to an object yet) is also an object!

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

- Python has local and global variables. Here's an example of local vs. global variable scope:

```
x = 5

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

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

set_x(43)
set_global_x(6)
```

Print Function

- Python has a `print` function:

```
print("I'm Python. Nice to meet you!") # Returns I'm Python. Nice to meet you!
```

- By default, the print function also prints out a newline at the end. Override the optional argument `end` to modify this behavior:

```
print("Hello, World", end="!") # Returns Hello, World!
```

Input Function

- Python offers a 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()` was named as `raw_input()`.

Order of Operations

- Just like mathematical operations in other languages, Python uses the BODMAS rule (also called the PEMDAS rule) to ascertain operator precedence. BODMAS is an acronym and it stands for Bracket, Of, Division, Multiplication, Addition, and Subtraction.
- To bypass BODMAS, enforce precedence with parentheses:

Basic Data Types

- Like most languages, Python has a number of basic types including integers, floats, booleans, and strings.

Numbers

- Integers work as you would expect from other languages:

```
x = 3
x       # Prints "3"
type(x) # Prints "<class 'int'>"
x + 1   # Addition; returns "4"
x - 1   # Subtraction; returns "2"
x * 2   # Multiplication; returns "6"
x ** 2  # Exponentiation; returns "9"
x += 1  # Returns "4"
x *= 2  # Returns "8"
x % 4   # Modulo operation; returns "3"
```

- Floats also behave similar to other languages:

```
y = 2.5
type(y)                 # Returns "<class 'float'>"
y, y + 1, y * 2, y ** 2 # Returns "(2.5, 3.5, 5.0, 6.25)"
```

- Some nuances in integer/float division that you should take note of:

```
3 / 2       # Float division in Python 3, returns "1.5"; integer division in Python 2, ret
urns "1"
3 // 2      # Integer division in both Python 2 and 3, returns "1"
10.0 / 3    # Float division in both Python 2 and 3, returns "3.33.."

# Integer division rounds down for both positive and negative numbers
-5 // 3     # -2
5.0 // 3.0  # 1.0
-5.0 // 3.0 # -2.0
```

- Note that unlike many languages, Python does not have unary increment (`x++`) or decrement (`x--`) operators, but accepts the `+=` and `=` operators.
- Python also has built-in types for complex numbers; you can find all of the details in the Python documentation.

Use Underscores to Format Large Numbers

- When working with a large number in Python, it can be difficult to figure out how many digits that number has. Python 3.6 and above allows you to use underscores as visual separators to group digits.

- In the example below, underscores are used to group decimal numbers by thousands.

```
large_num = 1_000_000
large_num # Returns 1000000
```

Booleans

- Python implements all of the usual operators for Boolean logic, but uses English words rather than symbols (`&&`, `||`, etc.):

```
t = True
f = False
type(t) # Returns "<class 'bool'>"
t and f # Logical AND; returns "False"
t or f  # Logical OR; returns "True"
not t   # Logical NOT; returns "False"
t != f  # Logical XOR; returns "True"
```

- The numerical value of `True` and `False` is `1` and `0`:

```
True + True # Returns 2
True * 8    # Returns 8
False - 5   # Returns -5
```

- Comparison operators look at the numerical value of `True` and `False`:

```
0 == False  # Returns True
1 == True   # Returns True
2 == True   # Returns False
-5 != False # Returns True
```

- `None`, `0`, and empty strings/lists/dicts/tuples all evaluate to `False`. All other values are `True`.

```
bool(0)  # Returns False
bool("") # Returns False
bool([]) # Returns False
```

```
bool({}) # Returns False
bool(()) # Returns False
```

- Equality comparisons yield boolean outputs:

```
# Equality is ==
1 == 1 # Returns True
2 == 1 # Returns False

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

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

# Seeing whether a value is in a range
1 < 2 and 2 < 3 # Returns True
2 < 3 and 3 < 2 # Returns False

# Chaining makes the above look nicer
1 < 2 < 3 # Returns True
2 < 3 < 2 # Returns False
```

- Casting integers as booleans transforms a non-zero integer to `True`, while zeros get transformed to `False`:

```
bool(0)  # Returns False
bool(4)  # Returns True
bool(-6) # Returns True
```

- Using logical operators with integers casts them to booleans for evaluation, using the same rules as mentioned above. However, note that the original pre-cast value is returned.

Strings

- Python has great support for strings:

```
hello = 'hello'                 # String literals can use single quotes
world = "world"                 # or double quotes; it does not matter.
                                # But note that you can nest one in another, for e.g.,
 'a"x"b' and "a'x'b"
print(hello)                    # Prints "hello"
len(hello)                      # String length; returns "5"
hello[0]                        # A string can be treated like a list of characters, retur
ns 'h'
hello + ' ' + world             # String concatenation using '+', returns "hello world"
"hello " "world"                # String literals (but not variables) can be concatenated
 without using '+', returns "hello world"
'%s %s %d' % (hello, world, 12) # sprintf style string formatting, returns "hello world 1
2"
```

- String objects have a bunch of useful methods; for example:

```
s = "hello"
s.capitalize()          # Capitalize a string; returns "Hello"
s.upper()               # Convert a string to uppercase; prints "HELLO"
s.rjust(7)              # Right-justify a string, padding with spaces; returns "  hello"
s.center(7)             # Center a string, padding with spaces; returns " hello "
s.replace('l', '(ell)') # Replace all instances of one substring with another;
                        # returns "he(ell)(ell)o"
'  world '.strip()      # Strip leading and trailing whitespace; returns "world"
```

- You can find a list of all string methods in the Python documentation.

String Formatting

- Python has several different ways of formatting strings. Simple positional formatting is probably the most common use-case. Use it if the order of your arguments is not likely to change and you only have very few elements you want to concatenate. Since the elements are not represented by something as descriptive as a name this simple style should only be used to format a relatively small number of elements.

Old Style/pre-Python 2.6

- The old style uses `'' % <tuple>` as follows:

```
'%s %s' % ('one', 'two') # Returns 'one two'
'%d %d' % (1, 2)         # Returns '1 2'
```

New Style/Python 2.6

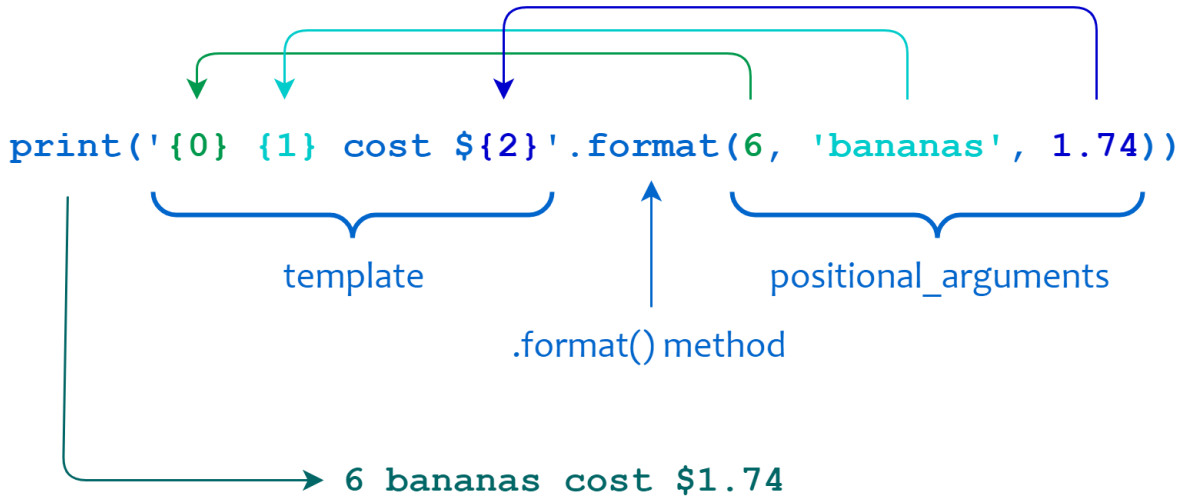
- The new style uses `''.format()` as follows:

```
'{} {}'.format('one', 'two') # Returns 'one two'
'{} {}'.format(1, 2)         # Returns '1 2'
```

- Note that both the old and new style of formatting are still compatible with the newest releases of Python, which is version 3.8 at the time of writing.
- With the new style formatting, you can give placeholders an explicit positional index (called positional arguments). This allows for re-arranging the order of display without changing the arguments. This operation is not available with old-style formatting.

```
'{1} {0}'.format('one', 'two') # Returns 'two one'
```

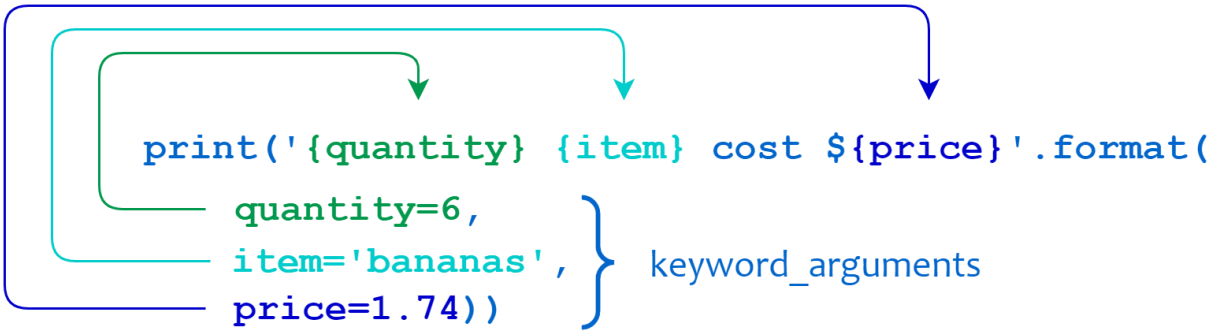
- For the example `print('{0} {1} cost ${2}'.format(6, 'bananas', 1.74))`, the output is `6 bananas cost $1.74`, as explained below:



- You can also use keyword arguments instead of positional parameters to produce the same result. This is called keyword arguments.

```
'{first} {second}'.format(first='one', second='two') # Returns 'one two'
```

- For the example `print('{quantity} {item} cost ${price}'.format(quantity=6, item='bananas', price=1.74))`, the output is `6 bananas cost $1.74`, as explained below:



F-strings

- Starting Python 3.6, you can also format strings using f-string literals, which are much more powerful than the old/new string formatters we discussed earlier:

```
name = "Reiko"
f"She said her name is {name}."           # Returns "She said her name is Reiko."

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

Padding and Aligning Strings

- By default, values are formatted to take up only as many characters as needed to represent the content. It is however also possible to define that a value should be padded to a specific length.
- Unfortunately the default alignment differs between old and new style formatting. The old style defaults to right aligned while the new style is left aligned.
- To align text right:

```
'%10s' % ('test',)      # Returns "      test"
'{:>10}'.format('test') # Returns "      test"
```

- To align text left:

```
'%-10s' % ('test',)     # Returns "test      "
'{:10}'.format('test')  # Returns "test      "
'{:<10}'.format('test') # Returns "test      "
```

- Again, the new style formatting surpasses the old variant by providing more control over how values are padded and aligned. You are able to choose the padding character and override the default space character for padding. This operation is not available with old-style formatting.

```
'{:_<10}'.format('test') # Returns "test______"
'{:0<10}'.format('test') # Returns "test000000"
```

- And also center align values. This operation is not available with old-style formatting.
- When using center alignment where the length of the string leads to an uneven split of the padding characters the extra character will be placed on the right side. This operation is not available with old-style formatting.
- You can also combine the field numbering (say, `{0}` for the first argument) specification with the format type (say, `{:s}` for strings):

```
'{2:s}, {1:s}, {0:s}'.format('test1', 'test2', 'test3') # Returns "test3, test2, test1"
'{2}, {1}, {0}'.format('test1', 'test2', 'test3')       # Returns "test3, test2, test1"
```

- Unpacking arguments:
- Unpacking arguments by name:

```
'Coordinates: {latitude}, {longitude}'.format(latitude='37.24N', longitude='-115.81W') # R
eturns "Coordinates: 37.24N, -115.81W"

coord = {'latitude': '37.24N', 'longitude': '-115.81W'}
'Coordinates: {latitude}, {longitude}'.format(**coord)                                  # R
eturns 'Coordinates: 37.24N, -115.81W'
```

- f-strings can also be formatted similarly:
  - As an example:

```
test2 = "test2"
test1 = "test1"
test0 = "test0"

f'{test2:10.3}, {test1:5.2}, {test0:2.1}' # Returns "tes       , te   , t "
```

  - Yet another one:

```
s1 = 'a'
s2 = 'ab'
s3 = 'abc'
s4 = 'abcd'
print(f'{s1:>10}') # Prints          a
print(f'{s2:>10}') # Prints         ab
print(f'{s3:>10}') # Prints        abc
print(f'{s4:>10}') # Prints       abcd
```

Truncating Long Strings

- Inverse to padding it is also possible to truncate overly long values to a specific number of characters. The number behind the `.` in the format specifies the precision of the output. For strings that means that the output is truncated to the specified length. In our example this would be 5 characters.

```
'%.5s' % ('xylophone',)     # Returns "xylop"
'{:.5}'.format('xylophone') # Returns "xylop"
```

Combining Truncating and Padding

- It is also possible to combine truncating and padding:

```
'%-10.5s' % ('xylophone',)    # Returns "xylop     "
'{:10.5}'.format('xylophone') # Returns "xylop     "
```

Numbers

- Of course it is also possible to format numbers.
- Integers:
- Floats:

```
'%f' % (3.141592653589793,)      # Returns "3.141593"
'{:f}'.format(3.141592653589793) # Returns "3.141593"
```

Padding Numbers

- Similar to strings numbers can also be constrained to a specific width.

```
'%4d' % (42,)      # Returns "  42"
'{:4d}'.format(42) # Returns "  42"
```

- Again similar to truncating strings the precision for floating point numbers limits the number of positions after the decimal point. For floating points, the padding value represents the length of the complete output (including the decimal). In the example below we want our output to have at least 6 characters with 2 after the decimal point.

```
'%06.2f' % (3.141592653589793,)      # Returns "003.14"
'{:06.2f}'.format(3.141592653589793) # Returns "003.14"
```

- For integer values providing a precision doesn't make much sense and is actually forbidden in the new style (it will result in a `ValueError`).

```
'%04d' % (42,)      # Returns "0042"
'{:04d}'.format(42) # Returns "0042"
```

- Some examples:
  - Specify a sign for floats:
    - Show a space for positive numbers, but a sign for negative numbers:
    - Show only the minus – same as `{:f}; {:f}`:

```
'{:-f}; {:-f}'.format(3.14, -3.14) # Returns "3.140000; -3.140000"
```

- Same for ints:
- Converting the value to different bases using replacing `{:d}`, `{:x}` and `{:o}`:
  - Note that format also supports binary numbers:

```
"int: {0:d};  hex: {0:x};  oct: {0:o};  bin: {0:b}".format(42) # Returns "i
nt: 42;  hex: 2a;  oct: 52;  bin: 101010'"
```

  - With `0x`, `0o`, or `0b` as prefix:

```
"int: {0:d};  hex: {0:#x};  oct: {0:#o};  bin: {0:#b}".format(42) # Returns
"int: 42;  hex: 0x2a;  oct: 0o52;  bin: 0b101010"
```

- Expressing a percentage:
- Using type-specific formatting:

```
import datetime
d = datetime.datetime(2010, 7, 4, 12, 15, 58)
'{:%Y-%m-%d %H:%M:%S}'.format(d) # Returns "2010-07-04 12:15:58"
```

- Nesting arguments and more complex examples:

```
for align, text in zip('<^>', ['left', 'center', 'right']):
    '{0:{fill}{align}16}'.format(text, fill=align, align=align)
# Returns:
# 'left<<<<<<<<<<<<'
# '^^^^^center^^^^^'
# '>>>>>>>>>>>right'

width = 5
for num in range(5,12):
    for base in 'dXob':
        print('{0:{width}{base}}'.format(num, base=base, width=width), end=' ')
    print()
# Prints:
#    5     5     5   101
#    6     6     6   110
```

```
#    7     7     7   111
#    8     8    10  1000
#    9     9    11  1001
#   10     A    12  1010
#   11     B    13  1011
```

- f-strings can also be formatted similarly:
  - Format width:

```
for x in range(1, 11):
    print(f'{x:02} {x*x:3} {x*x*x:4}')
# Prints:
# 01   1    1
# 02   4    8
# 03   9   27
# 04  16   64
# 05  25  125
# 06  36  216
# 07  49  343
# 08  64  512
# 09  81  729
# 10 100 1000
```

Index of a Substring in a Python String

- If you want to find the index of a substring in a string, use the `str.find()` method which returns the index of the first occurrence of the substring if found and `1` otherwise.
- Find the index of first occurrence of the substring:

```
sentence.find("day") # Returns 2
sentence.find("nice") # Returns -1
```

- You can also provide the starting and stopping position of the search:

```
# Start searching for the substring at index 3
sentence.find("day", 3) # Returns 15
```

- Note that you can also use `str.index()` to accomplish the same end result.

Replace One String with Another String Using Regular Expressions

- If you want to either replace one string with another string or to change the order of characters in a string, use `re.sub()`.
- `re.sub()` allows you to use a regular expression to specify the pattern of the string you want to swap.
- In the code below, we replace 3/7/2021 with Sunday and replace 3/7/2021 with 2021/3/7.

```
import re

text = "Today is 3/7/2021"
match_pattern = r"(\d+)/(\d+)/(\d+)"

re.sub(match_pattern, "Sunday", text) # Returns 'Today is Sunday'
re.sub(match_pattern, r"\3-\1-\2", text) # Returns 'Today is 2021-3-7'
```

Containers

- Containers are any object that holds an arbitrary number of other objects. Generally, containers provide a way to access the contained objects and to iterate over them.
- Python includes several built-in container types: lists, dictionaries, sets, and tuples:

```
from collections import Container # Can also use "from typing import Sequence"
isinstance(list(), Container)  # Prints True
isinstance(tuple(), Container) # Prints True
isinstance(set(), Container)   # Prints True
isinstance(dict(), Container)  # Prints True

# Note that the "dict" datatype is also a mapping datatype (along with being a container):
isinstance(dict(), collections.Mapping) # Prints True
```

Lists

- A list is the Python equivalent of an array, but is resizable and can contain elements of different types:

```
l = [3, 1, 2]   # Create a list
l[0]            # Access a list like you would any array; returns "1"
l[4]            # Looking out-of-bounds is an IndexError; Raises an "IndexError"
l[::-1]         # Return list in reverse order "[2, 1, 3]"
l, l[2]         # Returns "([3, 1, 2] 2)"
l[-1]           # Negative indices count from the end of the list; prints "2"

l[2] = 'foo'    # Lists can contain elements of different types
l               # Prints "[3, 1, 'foo']"

l.append('bar') # Add a new element to the end of the list
l               # Prints "[3, 1, 'foo', 'bar']"
x = l.pop()     # Remove and return the last element of the list
x, l            # Prints "bar [3, 1, 'foo']"
```

- Lists be “unpacked” 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

# Now look how easy it is to swap two values
b, a = [a, b            # a is now [2, 3] and b is now 1
```

- As usual, you can find all the gory details about lists in the Python documentation.

Iterate Over a List

- You can loop over the elements of a list like this:

```
animals = ['cat', 'dog', 'monkey']
for animal in animals:
    print(animal)
# Prints "cat", "dog", "monkey", each on its own line.
```

- If you want access to the index of each element within the body of a loop, use the built-in `enumerate` function:

```
animals = ['cat', 'dog', 'monkey']
for idx, animal in enumerate(animals):
    print('#%d: %s' % (idx + 1, animal))
# Prints "#1: cat", "#2: dog", "#3: monkey", each on its own line
```

List Comprehensions

- List comprehensions are a tool for transforming one list (any iterable actually) into another list. During this transformation, elements can be conditionally included in the new list and each element can be transformed as needed.

From Loops to Comprehensions

- Every list comprehension can be rewritten as a `for` loop but not every `for` loop can be rewritten as a list comprehension.
- The key to understanding when to use list comprehensions is to practice identifying problems that smell like list comprehensions.
- If you can rewrite your code to look just like this `for` loop, you can also rewrite it as a list comprehension:

```
new_things = []
for item in old_things:
    if condition_based_on(item):
        new_things.append("something with " + item)
```

- You can rewrite the above `for` loop as a list comprehension like this:

```
new_things = ["something with " + item for item in old_things if condition_based_on(item)]
```

- Thus, we can go from a `for` loop into a list comprehension by simply:
  - Copying the variable assignment for our new empty list.
  - Copying the expression that we've been `append`ing into this new list.
  - Copying the `for` loop line, excluding the final `:`.
  - Copying the `if` statement line, also without the `:`.
- As a simple example, consider the following code that computes square numbers:

```
nums = [0, 1, 2, 3, 4]
squares = []
```

```
for x in nums:
    if x % 2 == 0
        squares.append(x ** 2)
squares # Returns [0, 4, 16]
```

- You can make this code simpler using a **list comprehension**:

```
nums = [0, 1, 2, 3, 4]
[x ** 2 for x in nums if x % 2 == 0] # Returns [0, 4, 16]
```

- List comprehensions **do not** necessarily need to contain the `if` conditional clause:

```
nums = [0, 1, 2, 3, 4]
[x ** 2 for x in nums] # Returns [0, 1, 4, 9, 16]
```

- You can also use `if`/`else` in a list comprehension. Note that this actually uses a different language construct, a conditional expression, which itself is not part of the comprehension syntax, while the `if` after the `for` … `in` is part of the list comprehension syntax.

```
nums = [0, 1, 2, 3, 4]

# Create a list with x ** 2 if x is odd else 0, for each element in x
[x ** 2 if x % 2 else 0 for x in nums] # Returns [0, 1, 0, 9, 0]
```

- List comprehensions can emulate `map` and/or `filter` as follows:

```
[someFunc(i) for i in [1, 2, 3]]      # Returns [11, 12, 13]
[x for x in [3, 4, 5, 6, 7] if x > 5] # Returns [6, 7]
```

- On the other hand, list comprehensions can be equivalently written using a combination of the `list` constructor, and/or `map` and/or `filter`:

```
list(map(lambda x: x + 10, [1, 2, 3]))         # Returns [11, 12, 13]
list(map(max, [1, 2, 3], [4, 2, 1]))           # Returns [4, 2, 3]
list(filter(lambda x: x > 5, [3, 4, 5, 6, 7])) # Returns [6, 7]
```

Nested Loops

- In this section, we'll tackle list comprehensions with nested looping.
- Here's a `for` loop that flattens a matrix (a list of lists):

```
flattened = []
for row in matrix:
    for n in row:
        flattened.append(n)
```

- And here's a list comprehension that does the same thing:

```
flattened = [n for row in matrix for n in row]
```

- Nested loops in list comprehensions do not read like English prose. A common pitfalls is to read this list comprehension as:

```
flattened = [n for n in row for row in matrix]
```

- **But that's not right!** We've mistakenly flipped the `for` loops here. The correct version is the one above.
- When working with nested loops in list comprehensions remember that the `for` **clauses remain in the same order** as in our original `for` loops.

Slicing

- In addition to accessing list elements one at a time, Python provides concise syntax to access sublists; this is known as *slicing*:

```
nums = list(range(5)) # range is a built-in function that creates a list of integers
nums                  # Returns "[0, 1, 2, 3, 4]"
nums[2:4]             # Get a slice from index 2 to 4 (exclusive); returns "[2, 3]"
nums[2:]              # Get a slice from index 2 to the end; returns "[2, 3, 4]"
nums[:2]              # Get a slice from the start to index 2 (exclusive); returns "[0,
 1]"
nums[:]               # Get a slice of the whole list; returns "[0, 1, 2, 3, 4]"
nums[:-1]             # Slice indices can be negative; returns "[0, 1, 2, 3]"
```

- Assigning to a slice (even with a source of different length) is possible since lists are mutable:

```
# Case 1: source of the same length
nums1 = [1, 2, 3]
nums1[1:] = [4, 5]     # Assign a new sublist to a slice
nums1                  # Returns "[1, 4, 5]"

# Case 2: source of different length
nums2 = nums1
nums2[1:] = [6]        # Assign a new sublist to a slice
nums2                  # Returns "[1, 6]"
id(nums1) == id(nums2) # Returns True since lists are mutable, i.e., can be changed in-pla
ce
```

List Functions

```
l = [1, 2, 3]

l_copy = l[:]    # Make a one layer deep copy of l into l_copy
                 # Note that "l_copy is l" will result in False after this operation.
                 # This is similar to using the "copy()" method, i.e., l_copy = l.copy()

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

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

l.insert(2, 3)   # Insert an element at a specific index; l is now [1, 2, 3].
                 # Note that l.insert(n, 3) would return the same output, where n >= len
(l),
                 # for example, l.insert(3, 3).

l.index(3)       # Get the index of the first item found matching the argument; returns 3
# l.index(4)     # Raises a ValueError as 4 is not in the list

l_copy += [3]    # This is similar to using the "extend()" method

l + l_copy       # Concatenate two lists; returns [1, 2, 3, 1, 2, 3]
                 # Again, this is similar to using the "extend()" method; with the only
                 # difference being that "list.extend()" carries out the operation in plac
e,
                 # while '+' creates a new list object (and doesn't modify "l" and "l_cop
y").

l.append(l_copy) # You can append lists using the "append()" method; returns [1, 2, 3, [1,
2, 3]]
```

```
1 in l          # Check for existence (also called "membership check") in a list with "i
n"; returns True

len(l)          # Examine the length with "len()"; returns 4
```

- List concatenation using `.extend()` can be achieved using the in-place addition operator, `+=`.

```
# Extending a list with another iterable (in this case, a list)
l = [1, 2, 3]
l += [4, 5, 6]   # Returns "[1, 2, 3, 4, 5, 6]"
# Note that l += 4, 5, 6 works as well since the source argument on the right is already a
n iterable (in this case, a tuple)

l += [[7, 8, 9]] # Returns "[1, 2, 3, 4, 5, 6, [7, 8, 9]]"

# For slicing use-cases
l[1:] = [10]     # Returns "[1, 10]"
```

- Instead of needing to create an explicit list using the source argument (on the right), as a hack, you can simply use a trailing comma to create a tuple out of the source argument (and thus imitate the above functionality):

```
# Extending a list with another iterable (in this case, a list)
l = [1, 2, 3]
l += 4           # TypeError: 'int' object is not iterable
l += 4,          # Equivalent to l += (4,); same effect as l += [4]; returns "[1, 2, 3,
 4]"
l += [5, 6, 7],  # Returns "[1, 2, 3, 4, [5, 6, 7]]"

# For slicing use-cases
l[1:] = 10,      # Equivalent to l[1:] = (10,); same effect as l[1:] = [10]; returns "[1,
 10]"
```

Dictionaries

- A dictionary stores `(key, value)` pairs, similar to a `Map` in Java or an object in Javascript. In other words, dictionaries store mappings from keys to values.
- You can use it like this:

```
d = {"one": 1, "two": 2, "three": 3} # Create a new dictionary with some data
d['one']                             # Lookup values in the dictionary using "[]"; returns
```

```
"1"
'two' in d                          # Check if a dictionary has a given key; returns "Tru
e"
d['four'] = 4                       # Set an entry in a dictionary
d['four']                           # Returns "4"
```

- You can find all you need to know about dictionaries in the Python documentation.

Accessing a Non-existent Key

```
d = {"one": 1, "two": 2, "three": 3} # Create a new dictionary with some data
print(d["four"])                    # Returns "KeyError: 'four' not a key of d"
```

- In the above snippet, `four` does not exist in `d`. We get a `KeyError` when we try to access `d[four]`. As a result, in many situations, we need to check if the key exists in a dictionary before we try to access it.

`<dict>.get()`

- Use the `get()` method to avoid the KeyError:

```
d.get("four") # Returns "4"
d.get("five") # Returns None
```

- The `get()` method supports a default argument which is returned when the key being queried is missing:

```
d.get("five", 'N/A') # Get an element with a default; returns "N/A"
d.get("four", 'N/A') # Get an element with a default; returns "4"
```

- A good use-case for `get()` is getting values in a nested dictionary with missing keys where it can be challenging to use a conditional statement:

```
fruits = [
    {"name": "apple", "attr": {"color": "red", "taste": "sweet"}},
    {"name": "orange", "attr": {"taste": "sour"}},
    {"name": "grape", "attr": {"color": "purple"}},
    {"name": "banana"},
]
```

```
colors = [fruit["attr"]["color"]
          if "attr" in fruit and "color" in fruit["attr"] else "unknown"
          for fruit in fruits]
colors # Returns ['red', 'unknown', 'purple', 'unknown']
```

- In contrast, a better way is to use the `get()` method twice like below. The first get method will return an empty dictionary if the key `attr` doesn't exist. The second `get()` method will return unknown if the key color doesn't exist.

```
colors = [fruit.get("attr", {}).get("color", "unknown") for fruit in fruits]
colors # Returns ['red', 'unknown', 'purple', 'unknown']
```

`defaultdict`

- We can also use `collections.defaultdict` which returns a default value without having to specify one during every dictionary lookup. `defaultdict` operates in the same way as a dictionary in nearly all aspects except for handling missing keys. When accessing keys that don't exist, `defaultdict` automatically sets a default value for them. The factory function to create the default value can be passed in the constructor of `defaultdict`.

```
from collections import defaultdict

# create an empty defaultdict with default value equal to []
default_list_d = defaultdict(list)
default_list_d['a'].append(1) # default_list_d is now {"a": [1]}

# create a defaultdict with default value equal to 0
default_int_d = defaultdict(int)
default_int_d['c'] += 1       # default_int_d is now {"c": 1}
```

- We can also pass in a lambda as the factory function to return custom default values. Let's say for our default value we return the tuple `(0, 0)`.

```
pair_dict = defaultdict(lambda: (0,0))
print(pair_dict['point'] == (0,0)) # prints True
```

- Using a `defaultdict` can help reduce the clutter in your code, speeding up your implementation.

Key Membership Check

- We can easily check if a key is in a dictionary with `in`:

```
d = {"one": 1, "two": 2, "three": 3} # Create a new dictionary with some data
doesOneExist = "one" in d            # Return True
```

Iterating Over Keys

- Python also provides a nice way of iterating over the keys inside a dictionary. However, when you are iterating over keys in this manner, remember that you cannot add new keys or delete any existing keys, as this will result in an `RuntimeError`.

```
d = {"a": 0, "b": 5, "c": 6, "d": 7, "e": 11, "f": 19}
# iterate over each key in d
for key in d:
    print(d[key]) # This is OK
    del d[key]    # Raises a "RuntimeError: dictionary changed size during iteration"
    d[1] = 0      # Raises a "RuntimeError: dictionary changed size during iteration"
```

`del`

- Remove keys from a dictionary with the `del` operator:

```
d = {"one": 1, "two": 2, "three": 3} # Create a new dictionary with some data
del d["one"]                         # Delete key "numbers" of d
d.get("four", "N/A")                 # "four" is no longer a key; returns "N/A"
```

Key Datatypes

- Note that as we saw in the section on tuples, keys for dictionaries have to be immutable datatypes, such as ints, floats, strings, tuples, etc. This is to ensure that the key can be converted to a constant hash value for quick look-ups.

```
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.
```

- Get all keys as an iterable with `keys()`. Note that we need to wrap the call in `list()` to turn it into a list, as seen in the putting it all together section on iterators. Note that for Python versions <3.7, dictionary key ordering is not guaranteed, which is why your results might not match the example below exactly. However, as of Python 3.7, dictionary items maintain the order with which they are inserted into the dictionary.

```
list(filled_dict.keys()) # Can returns ["three", "two", "one"] in Python <3.7
list(filled_dict.keys()) # Returns ["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 convert the iterable into a list by generating the entire list at once. Note that the discussion above regarding key ordering holds below as well.

```
list(filled_dict.values()) # Returns [3, 2, 1] in Python <3.7
list(filled_dict.values()) # Returns [1, 2, 3] in Python 3.7+
```

Iterate Over a Dictionary

- It is easy to iterate over the keys in a dictionary:

```
d = {'person': 2, 'cat': 4, 'spider': 8}
for animal in d:
    legs = d[animal]
    print('A %s has %d legs' % (animal, legs))
# Prints "A person has 2 legs", "A cat has 4 legs", "A spider has 8 legs"
```

- If you want access to keys and their corresponding values, use the `items` method:

```
d = {'person': 2, 'cat': 4, 'spider': 8}
for animal, legs in d.items():
    print('A %s has %d legs' % (animal, legs))
# Prints "A person has 2 legs", "A cat has 4 legs", "A spider has 8 legs"
```

Dictionary Comprehensions

- These are similar to list comprehensions, but allow you to easily construct dictionaries.
- As an example, consider a `for` loop that makes a new dictionary by swapping the keys and values of the original one:

```
flipped = {}
for key, value in original.items():
    flipped[value] = key
```

- That same code written as a dictionary comprehension:

```
flipped = {value: key for key, value in original.items()}
```

- As another example:

```
nums = [0, 1, 2, 3, 4]
even_num_to_square = {x: x ** 2 for x in nums if x % 2 == 0}
print(even_num_to_square) # Prints "{0: 0, 2: 4, 4: 16}"
```

- Yet another example:

```
{x: x**2 for x in range(5)} # Returns {0: 0, 1: 1, 2: 4, 3: 9, 4: 16}
```

Sets

- A set is an **unordered** collection of distinct elements. In other words, sets do not allows duplicates and thus lend themselves for uses-cases involving retaining unique elements (and removing duplicates) canonically. As a simple example, consider the following:

```
animals = {'cat', 'dog'} # Note the syntax similarity to a dict.
'cat' in animals         # Check if an element is in a set; prints "True"
'fish' in animals        # Returns "False"
animals.add('fish')      # Add an element to a set
'fish' in animals        # Returns "True"
```

```
len(animals)             # Number of elements in a set; returns "3"
animals.add('cat')       # Adding an element that is already in the set does nothing
len(animals)             # Returns "3"
animals.remove('cat')    # Remove an element from a set
len(animals)             # Returns "2"
animals                  # Returns "{'fish', 'dog'}"
```

- You can start with an empty set and build it up:

```
animals = set()
animals.add('fish', 'dog') # Returns "{'fish', 'dog'}"
```

- 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}
```

- Make a one layer deep copy using the `copy` method:

```
s = {1, 2, 3}
s1 = s.copy() # s is {1, 2, 3}
s1 is s       # Returns False
```

Set Operations

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

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

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

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

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

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

- As usual, everything you want to know about sets can be found in the Python documentation.

Iterating Over a Set

- Iterating over a set has the same syntax as iterating over a list; however since sets are unordered, you cannot make assumptions about the order in which you visit the elements of the set:

```
animals = {'cat', 'dog', 'fish'}
for idx, animal in enumerate(animals):
    print('#%d: %s' % (idx + 1, animal))
# Prints "#1: fish", "#2: dog", "#3: cat"
```

Set Comprehensions

- Like lists and dictionaries, we can easily construct sets using set comprehensions.
- As an example, consider a `for` loop that creates a set of all the first letters in a sequence of words:

```
first_letters = set()
for w in words:
    first_letters.add(w[0])
```

- That same code written as a set comprehension:

```
first_letters = {w[0] for w in words}
```

- As another example:

```
from math import sqrt
nums = {int(sqrt(x)) for x in range(30)}
print(nums) # Prints "{0, 1, 2, 3, 4, 5}"
```

- Yet another example:

```
{x for x in 'abcddeef' if x not in 'abc'} # Returns {'d', 'e', 'f'}
```

Tuples

- A tuple is an immutable ordered list of values.

```
t = (1, 2, 3)
t[0]      # Returns "1"
t[0] = 3  # Raises a "TypeError: 'tuple' object does not support item assignment"
```

- Note that syntactically, 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))   # Returns <class 'int'>
type((1,))  # Returns <class 'tuple'>
type(())    # Returns <class 'tuple'>
```

> A tuple is in many ways similar to a list; one of the most important differences is that tuples **can be used as keys in dictionaries** and as **elements of sets**, while lists cannot:

```
d = {(x, x + 1): x for x in range(10)} # Create a dictionary with tuple keys
t = (1, 2)                             # Create a tuple
type(t)                                # Returns "<class 'tuple'>"
d[t]                                   # Returns "1"
d[(1, 2)]                              # Returns "1"

l = [1, 2]
d[l]                                   # Raises a "TypeError: unhashable type: 'list'"
d[[1, 2]]                              # Raises a "TypeError: unhashable type: 'list'"
```

- You can do most of the list operations on tuples too:

```
len(tup)         # Returns 3
tup + (4, 5, 6)  # Returns (1, 2, 3, 4, 5, 6)
```

```
tup[:2]          # Returns (1, 2)
2 in tup         # Returns True
```

- Just like lists, you can unpack tuples 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
```

- You can find all you need to know about tuples in the Python documentation.

Functions

- Python functions are defined using the `def` keyword:

```
def sign(x):
    if x > 0:
        return 'positive'
    elif x < 0:
        return 'negative'
    else:
        return 'zero'

for x in [-1, 0, 1]:
    print(sign(x)) # Prints "negative", "zero", "positive"
```

- We will often define functions to take (optional) keyword arguments, like this:

```
def hello(name, loud=False):
    if loud:
        print('HELLO, %s!' % name.upper())
    else:
        print('Hello, %s' % name)

hello('Bob') # Prints "Hello, Bob"
hello('Fred', loud=True) # Prints "HELLO, FRED!"
```

- Keyword arguments can arrive in any order:

```
def add(x, y):
    print("x is {} and y is {}".format(x, y))
    return x + y

add(y=6, x=5) # Returns 11
```

- You can define functions that take a variable number of positional arguments. In a function definition, `*` packs all arguments in a tuple (this process is called tuple-packing).

```
def varargs(*args):
    return args

varargs(1, 2, 3)  # Returns (1, 2, 3)
```

- You can define functions that take a variable number of keyword arguments, as well. In a function definition, `**` packs all arguments in a dictionary (this process is called dictionary-packing).

```
def keyword_args(**kwargs):
    return kwargs

keyword_args(big="foot", loch="ness") # Returns {"big": "foot", "loch": "ness"}
```

- You can do both tuple- and dictionary-packing, if you like:

```
def all_the_args(*args, **kwargs):
    print(args) # Prints (1, 2)
    print(kwargs) # Prints {"a": 3, "b": 4}
```

- There is a lot more information about Python functions in the Python documentation.
- In a function call, `*` and `**` play the opposite role as in a function definition. `*` unpacks all arguments in a tuple (this process is called tuple-unpacking). `**`

unpacks all arguments in a dictionary (this process is called dictionary-unpacking).

```
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)
```

- With Python, you can return multiple values from functions as intuitively as returning a single value:

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

- Python’s functions are first-class objects. This implies that you can assign them to variables, store them in data structures, pass them as arguments to other functions, and even return them as values from other functions:

```
def create_adder(x):
    def adder(y):
        return x + y
    return adder

add_10 = create_adder(10)
add_10(3) # Returns 13
```

- Note that while the following short-circuit AND code-pattern is seen in function return statements, it is rarely used but it is still valuable to know its usage:

```
# Short-circuit AND: The expression x and y first evaluates x; if x is false, its value is
returned; otherwise, y is evaluated and the resulting value is returned.
# Per https://docs.python.org/3.6/reference/expressions.html#boolean-operations
def short_circuit_and(a, b):
    return a and b

print(short_circuit_and(a=None, b=None)) # prints None
```

```
print(short_circuit_and(a=None, b=1))    # prints None
print(short_circuit_and(a=1, b=None))    # prints None
print(short_circuit_and(a=1, b=2))       # prints 2
```

- Note that `return a and b` is equivalent to:

```
if a:
    return b
else:
    return None
```

Nested Functions

- A function defined inside another function is called a nested function. Nested functions can access variables of the enclosing scope.
- In Python, these “non-local” variables are read-only by default and we must declare them explicitly as non-local (using `nonlocal` keyword) in order to modify them.
- Following is an example of a nested function accessing a non-local variable. In the code snippet below, we can see that the nested `printer()` function was able to access the non-local `msg` variable of the enclosing function.

```
def print_msg(msg):
    # This is the outer enclosing function

    def printer():
        # This is the nested function
        print(msg)

    printer()

# We execute the function
# Output: Hello
print_msg("Hello")
```

- which outputs:

Defining a Nonlocal Variable in a Nested Function Using `nonlocal`

- A `nonlocal` declaration is analogous to a `global` declaration. Both are needed only when a function assigns to a variable. Normally, such a variable would be made

local to the function. The `nonlocal` and `global` declarations cause it to refer to the variable that exists outside of the function. In other words, `nonlocal` lets you assign values to a variable in an outer (but non-global) scope similar to how `global` lets you assign values to a variable in a global scope. See PEP 3104 for more details on `nonlocal`.

- Note that if a function does not assign to a variable, then the declarations are not needed, and it automatically looks for it in a higher scope.
- As an example, consider the following code snippet that does not use `nonlocal`:

```
x = 0
def outer():
    x = 1
    def inner():
        x = 2
        print("inner:", x)

    inner()
    print("outer:", x)

outer()
print("global:", x)

# inner: 2
# outer: 1
# global: 0
```

- The following code snippet is a variation of the above which uses `nonlocal`, where `inner()`'s `x` is now also `outer()`'s `x`:

```
x = 0
def outer():
    x = 1
    def inner():
        nonlocal x
        x = 2
        print("inner:", x)

    inner()
    print("outer:", x)

outer()
print("global:", x)

# inner: 2
```

```
# outer: 2
# global: 0
```

- If we were to use `global`, it would bind `x` to the “global” `x`:

```
x = 0
def outer():
    x = 1
    def inner():
        global x
        x = 2
        print("inner:", x)

    inner()
    print("outer:", x)

outer()
print("global:", x)

# inner: 2
# outer: 1
# global: 2
```

Closure

Defining a Global Variable Using `global`

Type Hinting

- Introduced in Python 3.5, the `typing` module offers type hint functionality, which documents what type the contents of the containers needed to be.
- In the function `greeting` below, the argument `name` is expected to be of type `str` (annotated as `name: str`) and the return type `str`. Subtypes are accepted as arguments.

```
def greeting(name: str) -> str:
    return 'Hello ' + name
```

Type Aliases

- A type alias is defined by assigning the type to the alias. In this example, `Vector` and `list[float]` will be treated as interchangeable synonyms:

```
Vector = list[float]

def scale(scalar: float, vector: Vector) -> Vector:
    return [scalar * num for num in vector]

# typechecks; a list of floats qualifies as a Vector.
new_vector = scale(2.0, [1.0, -4.2, 5.4])
```

- Type aliases are useful for simplifying complex type signatures. For example:

```
from collections.abc import Sequence

ConnectionOptions = dict[str, str]
Address = tuple[str, int]
Server = tuple[Address, ConnectionOptions]

def broadcast_message(message: str, servers: Sequence[Server]) -> None:
    ...

# The static type checker will treat the previous type signature as
# being exactly equivalent to this one.
def broadcast_message(
        message: str,
        servers: Sequence[tuple[tuple[str, int], dict[str, str]]]) -> None:
    ...
```

- Note that `None` as a type hint is a special case and is replaced by `type(None)`.

Any

- The `Any` type is special in that it indicates an unconstrained datatype. A static type checker will treat every type as being compatible with `Any` and `Any` as being compatible with every type.
- This means that it is possible to perform any operation or method call on a value of type Any and assign it to any variable:

```
from typing import Any

a: Any = None
```

```
a = []          # OK
a = 2           # OK

s: str = ''
s = a           # OK

def foo(item: Any) -> int:
    # Typechecks; 'item' could be any type,
    # and that type might have a 'bar' method
    item.bar()
    ...
```

- Notice that no typechecking is performed when assigning a value of type `Any` to a more precise type. For example, the static type checker did not report an error when assigning `a` to `s` even though s was declared to be of type `str` and receives an `int` value at runtime!
- Furthermore, all functions without a return type or parameter types will implicitly default to using `Any`:

```
def legacy_parser(text):
    ...
    return data

# A static type checker will treat the above
# as having the same signature as:
def legacy_parser(text: Any) -> Any:
    ...
    return data
```

- This behavior allows `Any` to be used as an escape hatch when you need to mix dynamically and statically typed code.

Tuple

- Tuple type; `Tuple[X, Y]` is the type of a tuple of two items with the first item of type `X` and the second of type `Y`. The type of the empty tuple can be written as `Tuple[()]`.
- Example: `Tuple[T1, T2]` is a tuple of two elements corresponding to type variables `T1` and `T2`. `Tuple[int, float, str]` is a tuple of an int, a float and a string.

- To specify a variable-length tuple of homogeneous type, use literal ellipsis, e.g. `Tuple[int, ...]`. A plain Tuple is equivalent to `Tuple[Any, ...]`, and in turn to `tuple` (starting Python 3.9).
- In the example below, we're expecting the type of the `points` variable to be a tuple that contains two floats within.

```
from typing import Tuple

def example(points: Tuple[float, float]):
    return map(do_stuff, points)
```

List

- Practically similar to tuple type, just that list type is, as the name suggests, for lists.
- In the example below, we're expecting the function to return a list of dicts that map strings as keys to strings as values.

```
from typing import List

def example() -> List[Dict[str, str]]:
    return [{"1":"2"}]
```

Union

- Used to signify support for two or more dataypes; `Union[X, Y]` is equivalent to `X | Y` and means either `X or Y`.
- To define a union, use e.g. `Union[int, str]` or the shorthand `int | str`. Using the shorthand version is recommended. Details:
- The arguments must be types and there must be at least one.
- Unions of unions are flattened, e.g.:

```
Union[Union[int, str], float] == Union[int, str, float]
```

- Unions of a single argument vanish, e.g.:

```
Union[int] == int  # The constructor actually returns int
```

- Redundant arguments are skipped, e.g.:

```
Union[int, str, int] == Union[int, str] == int | str
```

- When comparing unions, the argument order is ignored, e.g.:

```
Union[int, str] == Union[str, int]
```

- You cannot subclass or instantiate a Union.
- You cannot write `Union[X][Y]`.

Optional

- `Optional[X]` is equivalent to `X | None` (or `Union[X, None]`). In other words, `Optional[...]` is a shorthand notation for `Union[..., None]`, telling the type checker that either an object of the specific type is required, or `None` is required. Note that `...` stands for any valid type hint, including complex compound types or a `Union[]` of more types.
- Note that this is not the same concept as an optional argument, which is one that has a default. An optional argument with a default does not require the `Optional` qualifier on its type annotation just because it is optional. For example:
- On the other hand, if an explicit value of `None` is allowed, the use of `Optional` is appropriate, whether the argument is optional or not. For example:

```
def foo(arg: Optional[int] = None) -> None:
    ...
```

- Thus, whenever you have a keyword argument with default value `None`, you should use `Optional`. (Note: If you are targeting Python 3.10 or newer, PEP 604 introduced a better syntax, see below).

- As an two example, if you have `dict` and `list` container types, but the default value for the a keyword argument shows that `None` is permitted too, use `Optional[...]`:

```
from typing import Optional

def test(a: Optional[dict] = None) -> None:
    #print(a) ==> {'a': 1234}
    #or
    #print(a) ==> None

def test(a: Optional[list] = None) -> None:
    #print(a) ==> [1, 2, 3, 4, 'a', 'b']
    #or
    #print(a) ==> None
```

- There is technically no difference between using `Optional[]` on a `Union[]`, or just adding `None` to the `Union[]`. So `Optional[Union[str, int]]` and `Union[str, int, None]` are exactly the same thing.
- As a recommendation, stick to using `Optional[]` when setting the type for a keyword argument that uses `= None` to set a default value, this documents the reason why `None` is allowed better. Moreover, it makes it easier to move the `Union[...]` part into a separate type alias, or to later remove the `Optional[...]` part if an argument becomes mandatory.
- For example:

```
from typing import Optional, Union

def api_function(optional_argument: Optional[Union[str, int]] = None) -> None:
    """API Function that does blah.

    If optional_argument is given, it must be an id of type string or int.
    """
```

- then documentation is improved by pulling out the `Union[str, int]` into a type alias:

```
from typing import Optional, Union

# ID types can be strings or integers -- support both.
IdTypes = Union[str, int]
```

```
def api_function(optional_argument: Optional[IdTypes] = None) -> None:
    """API Function that does blah.

    If optional_argument is given, it must be an id of type string or int.
    """
```

- The refactor to move the `Union[]` into an alias was made all the much easier because `Optional[...]` was used instead of `Union[str, int, None]`. The `None` value is not a ID type after all, it's not part of the value, `None` is meant to flag the absence of a value.
- When only reading from a container type, you may just as well accept any immutable abstract container type; lists and tuples are `Sequence` objects, while `dict` is a `Mapping` type:

```
from typing import Mapping, Optional, Sequence, Union

def test(a: Optional[Mapping[str, int]] = None) -> None:
    """accepts an optional map with string keys and integer values"""
    # print(a) ==> {'a': 1234}
    # or
    # print(a) ==> None

def test(a: Optional[Sequence[Union[int, str]]] = None) -> None:
    """accepts an optional sequence of integers and strings
    # print(a) ==> [1, 2, 3, 4, 'a', 'b']
    # or
    # print(a) ==> None
```

- In Python 3.9 and up, the standard container types have all been updated to support using them in type hints, see PEP 585. But, while you now can use `dict[str, int]` or `list[Union[int, str]]`, you still may want to use the more expressive `Mapping` and `Sequence` annotations to indicate that a function won't be mutating the contents (they are treated as 'read only'), and that the functions would work with any object that works as a mapping or sequence, respectively.
- Python 3.10 introduces the `|` union operator into type hinting, see PEP 604. Instead of `Union[str, int]` you can write `str | int`. In line with other type-hinted languages, the preferred (and more concise) way to denote an optional argument in Python 3.10 and up, is now `Type | None`, e.g. `str | None` or `list | None`.

Control Flow

`if` Statement

```
# Let's just make a variable
some_var = 5

# Here is an if statement.
# 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.")
```

Conditional Expression

- `if` can also be used as an expression to form a conditional expression, as an equivalent of C's `?:` ternary operator:
- Conditional expressions can be used in all kinds of situations where you want to choose between two expression values based on some condition:

```
value = 123
print(value, 'is', 'even' if value % 2 == 0 else 'odd')
```

`for` Loop

- `for` loops iterate over iterables such as lists, tuples, dictionaries, and sets.

```
for animal in ["dog", "cat", "mouse"]:
    print("{} is a mammal".format(animal))
```

- which outputs:

```
dog is a mammal
cat is a mammal
mouse is a mammal
```

- `range()` and `for` loops are a powerful combination. `range(number)` returns an iterable of numbers from zero to the given number. More on `range()` in its dedicated section.

```
for i in range(4):
    print(i)
```

- which outputs:
- `range(lower, upper)` returns an iterable of numbers from the lower number to the upper number.

```
for i in range(4, 8):
    print(i)
```

- which outputs:
- `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`.
- which outputs:
- To loop over a list, and retrieve both the index and the value of each item in the list, use `enumerate(iterable)`:

```
animals = ["dog", "cat", "mouse"]
for i, value in enumerate(animals):
    print(i, value)
```

- which outputs:
- For an in-depth treatment on how `for` loops work in Python, refer to our section on The Iterator Protocol.

`else` Clause

- `for` loops also have an `else` clause which most of us are unfamiliar with. The `else` clause executes after the loop completes normally. This means that the loop did not

encounter a `break` statement. They are really useful once you understand where to use them.

- The common construct is to run a loop and search for an item. If the item is found, we break out of the loop using the `break` statement. There are two scenarios in which the loop may end. The first one is when the item is found and `break` is encountered. The second scenario is that the loop ends without encountering a `break` statement. Now we may want to know which one of these is the reason for a loop's completion. One method is to set a flag and then check it once the loop ends. Another is to use the `else` clause.
- This is the basic structure of a `for/else` loop:

```
found_obj = None
for obj in objects:
    if obj.key == search_key:
        # Found it!
        found_obj = obj
        break
else:
    # Didn't find anything
    print('No object found.')
```

- Using `for` `else` or `while` `else` blocks in production code is not recommended owing to their obscurity. Thus, anytime you see this construct, a better alternative is to either encapsulate the search in a function:

```
def find_obj(search_key):
    for obj in objects:
        if obj.key == search_key:
            return obj
```

- Or simply use a list comprehension:

```
matching_objs = [o for o in objects if o.key == search_key]
if matching_objs:
    print('Found {}'.format(matching_objs[0]))
else:
    print('No object found.')
```

- Note that while the list comprehension version is not semantically equivalent to the other two versions, but it works good enough for non-performance critical code where it doesn't matter whether you iterate the whole list or not.
- Consider a simple example, which finds factors for numbers between 2 to 10:

```
for n in range(2, 10):
    for x in range(2, n):
        if n % x == 0:
            print(n, 'equals', x, '*', n/x)
            break
```

- By adding an additional `else` block which catches the numbers which have no factors and are therefore prime numbers:

```
for n in range(2, 10):
    for x in range(2, n):
        if n % x == 0:
            print( n, 'equals', x, '*', n/x)
            break
    else:
        # Loop fell through without finding a factor
        print(n, 'is a prime number')
```

`while` Loop

- While loops go on until a condition is no longer met:

```
x = 0
while x < 4:
    print(x)
    x += 1 # Shorthand for x = x + 1
```

- which outputs:

Lambda Functions

- Lambda expressions are a special syntax in Python for creating anonymous functions. The `lambda` syntax itself is generally referred to as a lambda expression, while the function you get back from this is called a lambda function.

- Python's lambda expressions allow a function to be created and passed around (often into another function) all in one line of code.
- Lambda expressions allow us to take this code:

```
colors = ["Goldenrod", "Purple", "Salmon", "Turquoise", "Cyan"]

def normalize_case(string):
    return string.casefold()

normalized_colors = map(normalize_case, colors)
```

- And turn it into this code:

```
colors = ["Goldenrod", "Purple", "Salmon", "Turquoise", "Cyan"]

normalized_colors = map(lambda s: s.casefold(), colors)
```

- Lambda expressions are just a special syntax for making functions. They can only have one statement in them and they return the result of that statement automatically.
- The inherent limitations of `lambda` expressions are actually part of their appeal. When an experienced Python programmer sees a lambda expression they know that they're working with a function that is **only used in one place** and does **just one thing**.
- Other examples of lambda functions:

```
(lambda x: x > 2)(3)                # Returns "True"
(lambda x, y: x ** 2 + y ** 2)(2, 1) # Returns "5"
```

Lambda Function Use-cases

- You'll typically see `lambda` expressions used when calling functions (or classes) that accept a function as an argument.
- Python's built-in `sorted` function accepts a function as its `key` argument. This key function is used to compute a comparison key when determining the sorting order of

items.

- So `sorted` is a great example of a place that lambda expressions are often used:

```
colors = ["Goldenrod", "purple", "Salmon", "turquoise", "cyan"]
sorted(colors, key=lambda s: s.casefold()) # Returns ['cyan', 'Goldenrod', 'purple', 'Salm
on', 'turquoise']
```

- The above code returns the given colors sorted in a case-insensitive way.
- The sorted function isn't the only use of lambda expressions, but it's a common one.

The Pros and Cons of Lambda Functions

- Both `lambda` expressions and `def` offer tools to define functions, but each of them have different limitations and use a different syntax.
- The main ways lambda expressions are different from def:
  - They can be immediately passed around (no variable needed)
  - They can only have a single line of code within them
  - They return automatically
  - They can't have a docstring and they don't have a name
  - They use a different and unfamiliar syntax
- The fact that `lambda` expressions can be passed around is their biggest benefit. Returning automatically is neat but generally not a big benefit. I find the "single line of code" limitation is neither good nor bad overall. The fact that lambda functions can't have docstrings and don't have a name is unfortunate and their unfamiliar syntax can be troublesome for newer Pythonistas.

Misuse and Overuse Scenarios

- In some cases, lambda expressions are used in ways that are unideal. Other times lambda expressions are simply being overused, i.e., they're acceptable but code written a different way would probably serve better.
- Let's take a look at the various ways lambda expressions are misused and overused.

Misuse: Naming Lambda Expressions

- PEP8, the official Python style guide, advises never to write code like this:

```
normalize_case = lambda s: s.casefold()
```

- The above statement makes an anonymous function and then assigns it to a variable. The above code ignores the reason lambda functions are useful: **lambda functions can be passed around without needing to be assigned to a variable first**.
- If you want to create a one-liner function and store it in a variable, you should use `def` instead:

```
def normalize_case(s): return s.casefold()
```

- PEP8 recommends this because named functions are a common and easily understood thing. This also has the benefit of giving our function a proper name, which could make debugging easier. Unlike functions defined with `def`, lambda functions never have a name (it's always `<lambda>`):

```
normalize_case = lambda s: s.casefold()
normalize_case # Returns "<function <lambda> at 0x7f264d5b91e0>"
def normalize_case(s): return s.casefold()
normalize_case # Returns "<function normalize_case at 0x7f247f68fea0>"
```

- **If you want to create a function and store it in a variable, define your function using `def`**. That's exactly what it's for. It doesn't matter if your function is a single line of code or if you're defining a function inside of another function, `def` works just fine for those use cases.

Misuse: Needless Function Calls

- I frequently see lambda expressions used to wrap around a function that was already appropriate for the problem at hand.
- For example take this code:

```
sorted_numbers = sorted(numbers, key=lambda n: abs(n))
```

- The person who wrote this code likely learned that lambda expressions are used for making a function that can be passed around. But they missed out on a slightly bigger picture idea: **all functions in Python (not just lambda functions) can be passed around**.
- Since abs (which returns the absolute value of a number) is a function and all functions can be passed around, we could actually have written the above code like this:

```
sorted_numbers = sorted(numbers, key=abs)
```

- Now this example might feel contrived, but it's not terribly uncommon to overuse lambda expressions in this way. Here's another example I've seen:

```
pairs = [(4, 11), (8, 8), (5, 7), (11, 3)]
sorted_by_smallest = sorted(pairs, key=lambda items: min(items))
```

- Because we're accepting exactly the same arguments as we're passing into `min`, we don't need that extra function call. We can just pass the `min` function to `key` instead:

```
pairs = [(4, 11), (8, 8), (5, 7), (11, 3)]
sorted_by_smallest = sorted(pairs, key=min)
```

- You don't need a lambda function if you already have another function that does what you want.

Overuse: Simple, but Non-trivial Functions

- It's common to see lambda expressions used to make a function that returns a couple of values in a tuple:

```
colors = ["Goldenrod", "Purple", "Salmon", "Turquoise", "Cyan"])
colors_by_length = sorted(colors, key=lambda c: (len(c), c.casefold()))
```

- That `key` function here is helping us sort these colors by their length followed by their case-normalized name.
- The code below carries out the same functionality as the above code, but is much more readable:

```
def length_and_alphabetical(string):
    """Return sort key: length first, then case-normalized string."""
    return (len(string), string.casefold())

colors = ["Goldenrod", "Purple", "Salmon", "Turquoise", "Cyan"])
colors_by_length = sorted(colors, key=length_and_alphabetical)
```

- This code is quite a bit more verbose, but I find the name of that key function makes it clearer what we're sorting by. We're not just sorting by the length and we're not just sorting by the color: we're sorting by both.
- **If a function is important, it deserves a name**. You could argue that most functions that are used in a lambda expression are so trivial that they don't deserve a name, but there's often little downside to naming functions and I find it usually makes my code more readable overall.
- Naming functions often makes code more readable, the same way using tuple unpacking to name variables instead of using arbitrary index-lookups often makes code more readable.

Overuse: When Multiple Lines Would Help

- Sometimes the "just one line" aspect of lambda expressions cause us to write code in convoluted ways. For example take this code:

```
points = [((1, 2), 'red'), ((3, 4), 'green')]
points_by_color = sorted(points, key=lambda p: p[1])
```

- We’re hard-coding an index lookup here to sort points by their color. If we used a named function we could have used tuple unpacking to make this code more readable:

```
def color_of_point(point):
    """Return the color of the given point."""
    (x, y), color = point
    return color

points = [((1, 2), 'red'), ((3, 4), 'green')]
points_by_color = sorted(points, key=color_of_point)
```

- Tuple unpacking can improve readability over using hard-coded index lookups. **Using lambda expressions often means sacrificing some Python language features**, specifically those that require multiple lines of code (like an extra assignment statement).

Overuse: Lambda with Map and Filter

- Python’s map and filter functions are almost always paired with lambda expressions. It’s common to see StackOverflow questions asking “what is lambda” answered with code examples like this:

```
numbers = [2, 1, 3, 4, 7, 11, 18]
squared_numbers = map(lambda n: n**2, numbers)
odd_numbers = filter(lambda n: n % 2 == 1, numbers)
```

- Python’s `map` and `filter` functions are used for looping over an iterable and making a new iterable that either slightly changes each element or filters the iterable down to only elements that match a certain condition. We can accomplish both of those tasks just as well with list comprehensions or generator expressions:

```
numbers = [2, 1, 3, 4, 7, 11, 18]
squared_numbers = (n**2 for n in numbers)
odd_numbers = (n for n in numbers if n % 2 == 1)
```

- Personally, I’d prefer to see the above generator expressions written over multiple lines of code (see my article on comprehensions) but I find even these one-line

generator expressions more readable than those `map` and `filter` calls.

- The general operations of mapping and filtering are useful, but we really don't need the `map` and `filter` functions themselves. Generator expressions are a special syntax that exists just for the tasks of mapping and filtering. So my advice is to **use generator expressions instead of the map and `filter` functions**.

Misuse: Sometimes You Don't Even Need to Pass a Function

- What about cases where you need to pass around a function that performs a single operation?
- Newer Pythonistas who are keen on functional programming sometimes write code like this:

```
from functools import reduce

numbers = [2, 1, 3, 4, 7, 11, 18]
total = reduce(lambda x, y: x + y, numbers)
```

- This code adds all the numbers in the numbers list. There's an even better way to do this:
- Python's built-in `sum` function was made just for this task.
- The `sum` function, along with a number of other specialized Python tools, are easy to overlook. But I'd encourage you to seek out the more specialized tools when you need them because they often make for more readable code.
- Instead of passing functions into other functions, **look into whether there is a more specialized way to solve your problem instead**.

Overuse: Using Lambda for Very Simple Operations

- Let's say instead of adding numbers up, we're multiply numbers together:

```
from functools import reduce

numbers = [2, 1, 3, 4, 7, 11, 18]
product = reduce(lambda x, y: x * y, numbers, 1)
```

- The above lambda expression is necessary because we're not allowed to pass the `*` operator around as if it were a function. If there was a function that was equivalent to `*`, we could pass it into the `reduce` function instead.
- Python's standard library actually has a whole module meant to address this problem:

```
from functools import reduce
from operator import mul

numbers = [2, 1, 3, 4, 7, 11, 18]
product = reduce(mul, numbers, 1)
```

- Python's operator module exists to make various Python operators easy to use as functions. If you're practicing functional(ish) programming, **Python's `operator` module is your friend**.
- In addition to providing functions corresponding to Python's many operators, the operator module provides a couple common higher level functions for accessing items and attributes and calling methods.
- There's `itemgetter` for accessing indexes of a list/sequence or keys of a dictionary/mapping:

```
# Without operator: accessing a key/index
rows_sorted_by_city = sorted(rows, key=lambda row: row['city'])

# With operator: accessing a key/index
from operator import itemgetter
rows_sorted_by_city = sorted(rows, key=itemgetter('city'))
```

- There's also `attrgetter` for accessing attributes on an object:

```
# Without operator: accessing an attribute
products_by_quantity = sorted(products, key=lambda p: p.quantity)

# With operator: accessing an attribute
from operator import attrgetter
products_by_quantity = sorted(products, key=attrgetter('quantity'))
```

- And `methodcaller` for calling methods on an object:

```
# Without operator: calling a method
sorted_colors = sorted(colors, key=lambda s: s.casefold())

# With operator: calling a method
from operator import methodcaller
sorted_colors = sorted(colors, key=methodcaller('casefold'))
```

- Functions in the `operator` module typically make code more readable than using the equivalent lambda expressions.

Overuse: When Higher Order Functions Add Confusion

- A function that accepts a function as an argument is called a higher order function. Higher order functions are the kinds of functions that we tend to pass lambda functions to.
- The use of higher order functions is common when practicing functional programming. Functional programming isn't the only way to use Python though: Python is a multi-paradigm language so we can mix and match coding disciplines to make our code more readable.
- Compare this:

```
from functools import reduce

numbers = [2, 1, 3, 4, 7, 11, 18]
product = reduce(lambda x, y: x * y, numbers, 1)
```

- To this:

```
def multiply_all(numbers):
    """Return the product of the given numbers."""
    product = 1
    for n in numbers:
        product *= n
    return product

numbers = [2, 1, 3, 4, 7, 11, 18]
product = multiply_all(numbers)
```

- The second code is longer, but folks without a functional programming background will often find it easier to understand.
- Anyone who has gone through one of my Python training courses can probably understand what that `multiply_all` function does, whereas that `reduce` / `lambda` combination is likely a bit more cryptic for many Python programmers.
- In general, **passing one function into another function, tends to make code more complex, which can hurt readability**.

The Iterator Protocol: How `for` Loops Work in Python

- In this section, we'll discover what makes `for` loops tick.

Looping with Indexes

- Let's try using a traditional looping idiom from the world of C: looping with indexes!

```
colors = ["red", "green", "blue", "purple"]
i = 0
while i < len(colors):
    print(colors[i])
    i += 1
```

- Note that while this works on lists, it fails on sets:

```
colors = {"red", "green", "blue", "purple"}
i = 0
while i < len(colors):
...     print(colors[i])
...     i += 1
...
```

- which outputs:

```
Traceback (most recent call last):
  File "<stdin>", line 2, in <module>
TypeError: 'set' object does not support indexing
```

- This approach only works on **sequences**, which are data types that have indexes from `0` to one less than their length. Sequences have three important properties:
  - They are indexable,
  - They have known lengths and,
  - They are are finite.
- **Lists**, **strings**, and **tuples** are **sequences**. **Dictionaries**, **sets**, and many other iterables are **not sequences**.
- The key takeaway here is that this looping construct, which essentially indexes the iterable does **not** work on all iterables, but only on **sequences**.

Iterables

- Python offers a fundamental abstraction called the iterable. Formally, an iterable is any Python object capable of returning its members one at a time, permitting it to be iterated over in a for-loop.
- Like we discussed in the prior section, iterables can either be sequences or not.
- Here's an infinite iterable which provides every multiple of 5 as you loop over it:

```
from itertools import count
multiples_of_five = count(step=5)
```

- When we were using `for` loops, we could have looped over the beginning of this iterable like this:

```
for n in multiples_of_five:
    if n > 100:
        break
    print(n)
```

- If we removed the `break` condition from the aforementioned `for` loop, it would simply go on printing forever.
- So iterables can be infinitely long: which means that we can't always convert an iterable to a list (or any other sequence) before we loop over it. We need to

somehow ask our iterable for each item of our iterable individually, the same way our `for` loop works.

Iterators

- While an iterable is anything you're able to **loop over**, an iterator is the object that does the **actual iterating**.
- Iterators have exactly one job: return the "next" item in our iterable. They're sort of like tally counters, but they don't have a reset button and instead of giving the next number they give the next item in our iterable.
- All iterables can be passed to the built-in `iter` function to get an iterator from them:

```
iter(['some', 'list']) # Returns "<list_iterator object at 0x7f227ad51128>"
iter({'some', 'set'})  # Returns "<set_iterator object at 0x7f227ad32b40>"
iter('some string')    # Returns "<str_iterator object at 0x7f227ad51240>"
```

- And iterators can be passed to `next` to get their next item:

```
iterator = iter('hi')
next(iterator) # Returns "h"
next(iterator) # Returns "i"
next(iterator)
```

- which outputs:

```
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
StopIteration
```

- So iterators can be passed to the built-in `next` function to get the next item from them and if there is no next item (because we reached the end), a `StopIteration` exception will be raised.

Iterators are Also Iterables

- So calling `iter` on an iterable gives us an iterator. And calling `next` on an iterator gives us the next item or raises a `StopIteration` exception if there aren't any more

items.

- There's actually a bit more to it than that though. You can pass iterators to the built-in `iter` function to get themselves back. That means that iterators are also iterables.

```
iterator = iter('hi')
iterator2 = iter(iterator)
iterator is iterator2 # Returns "True"
```

The Iterator Protocol

- The iterator protocol is a fancy phrase which basically implies "how iterables actually work in Python".
- To summarize the key takeaway points about iterables and iterators:
  - Iterables:
    - Can be passed to the `iter` function to get an iterator for them.
  - Iterators:
    - Can be passed to the next function which gives their next item or raises `StopIteration`.
    - Return themselves when passed to the iter function.
- The inverse of these statements should also hold true. Which means:
  - Anything that can be passed to iter without an error is an iterable.
  - Anything that can be passed to next without an error (except for StopIteration) is an iterator.
  - Anything that returns itself when passed to iter is an iterator.

Looping with Iterators

- With what we've learned about iterables and iterators, we should now be able to recreate a `for` loop without actually using a `for` loop!
- This `while` loop manually loops over some iterable, printing out each item as it goes:

```
def print_each(iterable):
    iterator = iter(iterable)
    while True:
        try:
            item = next(iterator)
        except StopIteration:
            break  # Iterator exhausted: stop the loop
        else:
            print(item)
```

- We can call this function with any iterable and it will loop over it:
- which outputs:
- The above function is essentially the same as this one which uses a `for` loop:

```
def print_each(iterable):
    for item in iterable:
        print(item)
```

- This `for` loop is automatically doing what we were doing manually: calling `iter` to get an iterator and then calling next over and over until a `StopIteration` exception is raised.
- The iterator protocol is used by `for` loops, tuple unpacking, and all built-in functions that work on generic iterables. Using the iterator protocol (either manually or automatically) is the only universal way to loop over any iterable in Python.
- **Key takeaways**
  - Everything you can loop over is an iterable.
  - Looping over iterables works via getting an iterator from an iterable and then repeatedly asking the iterator for the next item.
  - The way iterators and iterables work is called the iterator protocol. List comprehensions, tuple unpacking, `for` loops, and all other forms of iteration rely on the iterator protocol.

Putting It All Together

- Packing everything that we covered so far in code:

```
filled_dict = {"one": 1, "two": 2, "three": 3}
our_iterable = filled_dict.keys()
print(our_iterable) # Returns dict_keys(['one', 'two', 'three']). This is an object that i
mplements Python's iterator protocol.

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

# However we cannot address elements by index, since a dict is not a sequence (but is an i
terable).
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)  # Returns "one"

# It maintains state as we iterate.
next(our_iterator)  # Returns "two"
next(our_iterator)  # Returns "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
```

Creating a Custom Iterator

Why Make an Iterator?

- Iterators allow you to make an iterable that computes its items as it goes. Which means that you can make iterables that are **lazy**, in that they don’t determine what their next item is until you ask them for it.
- Using an iterator instead of a list, set, or another iterable data structure can sometimes allow us to save memory. For example, we can use `itertools.repeat()` to create an iterable that provides 100 million 4’s to us:

```
from itertools import repeat
lots_of_fours = repeat(4, times=100_000_000)
```

- This iterator takes up 56 bytes of memory (this number can vary depending on the architectural specification of your machine):

```
import sys
sys.getsizeof(lots_of_fours) # Returns "56"
```

- An equivalent list of 100 million 4's takes up many megabytes of memory:

```
lots_of_fours = [4] * 100_000_000
import sys
sys.getsizeof(lots_of_fours) # Returns "800000064"
```

- While iterators can save memory, they can also save time. For example if you wanted to print out just the first line of a 10 gigabyte log file, you could do this:

```
print(next(open('giant_log_file.txt'))) # Returns "This is the first line in a giant file"
```

- File objects in Python are implemented as iterators. As you loop over a file, data is read into memory one line at a time. If we instead used the `readlines` method to store all lines in memory, we might run out of system memory.
- So iterators can **save us memory**, and can **sometimes save us time** also.
- Additionally, iterators have **abilities that other iterables don't**. For example, the laziness of iterables can be used to make iterables that have an unknown length. In fact, you can even make infinitely long iterators.
- For example, the `itertools.count()` utility will give us an iterator that will provide every number from 0 upward as we loop over it:

```
from itertools import count
for n in count():
...     print(n)
...
```

- which outputs:
- That `itertools.count()` object is essentially an infinitely long iterable. And it's implemented as an iterator.

Making an Iterator: the Object-oriented Way

- So we've seen that iterators can save us memory, save us time, and unlock new abilities for us.
- Let's make our own iterators. We'll start be re-inventing the `itertools.count()` iterator object.
- Here's an iterator implemented using a class:

```
class Count:
    """Iterator that counts upward forever."""

    def __init__(self, start=0):
        self.num = start

    def __iter__(self):
        return self

    def __next__(self):
        num = self.num
        self.num += 1
        return num
```

- This class has an initializer that initializes our current number to `0` (or whatever is passed in as the start). The things that make this class usable as an iterator are the `__iter__` and `__next__` methods.
- When an object is passed to the `str` built-in function, its `__str__` method is called. When an object is passed to the `len` built-in function, its `__len__` method is called.

```
numbers = [1, 2, 3]
str(numbers), numbers.__str__() # Returns "('[1, 2, 3]', '[1, 2, 3]')"
len(numbers), numbers.__len__() # Returns "(3, 3)""
```

- Calling the built-in `iter` function on an object will attempt to call its `__iter__` method. Calling the built-in next function on an object will attempt to call its `__next__`

method.

- The `iter` function is supposed to return an iterator. So our `__iter__` function must return an iterator. But our object is an iterator, so should return ourself. Therefore our `Count` object returns self from its `__iter__` method because it is its own iterator.
- The `next` function is supposed to return the next item in our iterator or raise a `StopIteration` exception when there are no more items. We're returning the current number and incrementing the number so it'll be larger during the next `__next__` call.
- We can manually loop over our `Count` iterator class like this:
- We could also loop over our `Count` object using a `for` loop, as with any other iterable:
- which outputs:
- This object-oriented approach to making an iterator is cool, but it's not the usual way that Python programmers make iterators. Usually when we want an iterator, we create a generator, which brings us to the topic of our next section.

Generators

- Generators are an easy way to make iterators.
- What separates generators from typical iterators is that fact that they offer lazy (on demand) generation of values, which translates to lower memory usage.
- Furthermore, we do not need to wait until all the elements have been generated before we start to use them, which yields a performance improvement.
- Note that a generator will provide performance benefits only if we **do not** intend to use the set of generated values more than once.
- Let's motivate generators by considering a simple example of building a list of the first numbers (i.e., a count function) and returning it:

```
# Build and return a list
def firstn(n):
    num, nums = 0, []
    while num < n:
        nums.append(num)
        num += 1
    return nums
```

```
sum_of_first_n = sum(firstn(1000000))
```

- The code is quite simple and straightforward, but it builds the full list in memory. This is clearly not acceptable in our case, because we cannot afford to keep all “10 megabyte” integers in memory.
- Let’s switch over to generators to figure out how they help solve the aforementioned problem. Generators are memory-efficient because they only load the data needed to process the next value in the iterable. This allows them to perform operations on otherwise prohibitively large value ranges. The following implements generator as an iterable object:

```
class firstn(object):
    def __init__(self, n):
        self.n = n
        self.num = 0

    def __iter__(self):
        return self

    # Python 3 compatibility
    def __next__(self):
        return self.next()

    def next(self):
        if self.num < self.n:
            cur, self.num = self.num, self.num+1
            return cur
        else:
            raise StopIteration()

sum_of_first_n = sum(firstn(1000000))
```

- This will perform as we expect, but we have the following issues:
  - There is a lot of boilerplate code.
  - The logic is expressed isn’t expressed in a straightforward way, and is somewhat convoluted.
- Furthermore, this is a pattern that we will use over and over for many similar constructs. Imagine writing all that just to get an iterator!

- This leads us to the two ways to easily create generators in Python: generator functions and generator expressions.

Generator Functions

- Generator functions differ from plain old functions based on the fact that they have one or more `yield` statements.
- Python provides **generator functions** as a convenient shortcut to building iterators. Let's rewrite the iterator in the previous section as a generator function:

```
# A generator that yields items instead of returning a list
def firstn(n):
    num = 0
    while num < n:
        yield num
        num += 1

result = firstn(1000000)
result       # Returns "<generator object firstn at some address>"
type(result) # Returns "<class 'generator'>"
next(result) # Returns 0
next(result) # Returns 1
next(result) # Returns 2
```

- Alternatively, note that calling `iter` on the generator function **also yields a generator object**:

```
result = iter(firstn(1000000))
result # Returns "<generator object firstn at some address>"
```

- Like we discussed earlier, note that the mere presence of a `yield` statement turns a function into a generator function.
- Performing an operation such as `sum()` that requires all elements of the iterator to be available leads to static generation of the list, similar to a regular iterator:

```
sum_of_first_n = sum(firstn(1000000))
```

- You can use a `for` loop over this generator which automatically calls on `next()` to loop through elements:
- Note that this function is considerably shorter (with much less boilerplate code) than the `firstn` class we created in the previous section.
- As another example, given this list of numbers:
- We can make a generator that will lazily provide us with all the squares of these numbers like this:

```
def square_all(numbers):
    for n in numbers:
        yield n**2

squares = square_all(favorite_numbers)
```

Generator Expressions

- Similar to list comprehensions, you can create generator comprehensions as well, which are more commonly known as generator expressions. In other words, generator expressions offer list comprehension-like syntax that allows us to create generators.
- In fact, we can turn a list comprehension into a generator expression by replacing the square brackets `[ ]` with parentheses `( )`:

```
values = (x for x in [1, 2, 3, 4, 5]) # or (x for x in range(1, 6))

for x in values:
    print(x)
```

- Alternately, we can think of list comprehensions as generator expressions wrapped in a list constructor. In other words, you can obtain a list comprehension by casting a generator comprehension to a list:

```
# list comprehension
doubles = [2 * n for n in range(50)]

# same as the list comprehension above
doubles = list(2 * n for n in range(50))
```

- As another example, here's a generator expression that filters empty lines from a file and strips newlines from the end:

```
lines = (line.rstrip('\n') for line in poem_file if line != '\n')
```

- Yet another example, given this list of numbers:
- We can make a generator that will lazily provide us with all the squares of these numbers like this:

```
squares = (n**2 for n in favorite_numbers)
```

- Thus, generator expressions use a shorter inline syntax compared to generator functions. Since they are essentially abstracting away details, they can be a little less powerful compared to generator functions in certain scenarios.
    - If you're doing a simple **mapping or filtering** operation, a **generator expression** is a great solution.
    - If you're doing something a bit more **sophisticated**, you'll likely need a **generator function**.

Generator Expressions vs. Generator Functions

- You can think of generator expressions as the list comprehensions of the generator world.
- First, let's talk about terminology. The word "generator" is used in quite a few ways in Python:
    - A generator, also called a generator object, is an iterator whose type is generator.
    - A generator function is a special syntax that allows us to make a function which returns a generator object when we call it.
    - A generator expression is a comprehension-like syntax that allows you to create a generator object inline.

- Second, you can also copy-paste your way from a generator function to a function that returns a generator expression:
  - If you can write your generator function in this form:

```
def get_a_generator(some_iterable):
    for item in some_iterable:
        if some_condition(item):
            yield item
```

  - Then you can replace it with a generator expression:

```
def get_a_generator(some_iterable):
    return (item for item in some_iterable if some_condition(item))
```

  - If you can't write your generator function in that form, then you can't create a generator expression to replace it.
- Generator expressions are to generator functions as list comprehensions are to a simple `for` loop with an `append()` and an `if` condition.
- Generator expressions are so similar to comprehensions, that you might even be tempted to say generator comprehensions instead of generator expressions. Technically, that's incorrect, but if you say it everyone will still know what you're talking about :)

Decorators

- Decorators are functions that wrap around other functions.
- A decorators dynamically alters the functionality of a function or method, or class without having to directly use subclasses or change the source code of the function being decorated.
- In the below example, `beg` wraps `say`. If `say_please` is `True` then it will change the returned message.

```
def beg(target_function):
    def wrapper(*args, **kwargs):
        msg, say_please = target_function(*args, **kwargs)
```

```
        if say_please:
            return "{} {}".format(msg, "Please! I am poor :(")
        return msg

    return wrapper

@beg
def say(say_please=False):
    msg = "Can you buy me a beer?"
    return msg, say_please

say()               # Returns "Can you buy me a beer?"
say(say_please=True) # Returns "Can you buy me a beer? Please! I am poor :("
```

- Another example:

```
def my_simple_logging_decorator(func):
    def you_will_never_see_this_name(*args, **kwargs):
        print('Calling {}'.format(func.__name__))
        return func(*args, **kwargs)
    return you_will_never_see_this_name

@my_simple_logging_decorator
def double(x):
    'Doubles a number.'
    return 2 * x

double(155) # Returns "Calling double"
            # Returns "310"
```

- You can also chain decorators:

```
def makebold(fn):
    def wrapped(*args, **kwargs):
        return "<b>" + fn(*args, **kwargs) + "</b>"
    return wrapped

def makeitalic(fn):
    def wrapped(*args, **kwargs):
        return "<i>" + fn(*args, **kwargs) + "</i>"
    return wrapped

@makebold
@makeitalic
def hello():
    return "hello world"

@makebold
```

```
@makeitalic
def log(s):
    return s

hello()      # Returns "<b><i>hello world</i></b>"
log('hello') # Returns "<b><i>hello</i></b>"
```

File I/O

- In this section, you'll learn about Python file operations. More specifically, opening a file, reading from it, writing into it, closing it, and various file methods that you should be aware of.

Files

- Files are named locations on disk to store related information. They are used to permanently store data in a non-volatile memory (e.g. hard disk).
- Since Random Access Memory (RAM) is volatile (which loses its data when the computer is turned off), we use files for future use of the data by permanently storing them.
- When we want to read from or write to a file, we need to open it first. When we are done, it needs to be closed so that the resources that are tied with the file are freed.
- Hence, in Python, a file operation takes place in the following order:
    1. Open a file
    2. Read or write (perform operation)
    3. Close the file

Opening Files in Python

- Python has a built-in open() function to open a file. This function returns a file object, also called a handle, as it is used to read or modify the file accordingly.

```
>>> f = open("test.txt")    # open file in current directory
>>> f = open("C:/Python38/README.txt")  # specifying full path
```

- We can specify the mode while opening a file. In mode, we specify whether we want to read r, write w or append a to the file. We can also specify if we want to open the file in text mode or binary mode.
- The default is reading in text mode. In this mode, we get strings when reading from the file.
- On the other hand, binary mode returns bytes and this is the mode to be used when dealing with non-text files like images or executable files.

Mode	Description
`r`	Opens a file for reading. (default)
`w`	Opens a file for writing. Creates a new file if it does not exist or truncates the file if it exists.
`x`	Opens a file for exclusive creation. If the file already exists, the operation fails.
`a`	Opens a file for appending at the end of the file without truncating it. Creates a new file if it does not exist.
`t`	Opens in text mode. (default)
`b`	Opens in binary mode.
`+`	Opens a file for updating (reading and writing)

- As an example:

```
f = open("test.txt")      # equivalent to 'r' or 'rt'
f = open("test.txt", 'w')  # write in text mode
f = open("img.bmp", 'r+b') # read and write in binary mode
```

- Unlike other languages, the character `a` does not imply the number 97 until it is encoded using `ASCII` (or other equivalent encodings).
- Moreover, the default encoding is platform dependent. In windows, it is `cp1252` but `utf-8` in Linux.
- So, we must not also rely on the default encoding or else our code will behave differently in different platforms.
- Hence, when working with files in text mode, it is highly recommended to specify the encoding type.

```
f = open("test.txt", mode='r', encoding='utf-8')
```

Closing Files in Python

- When we are done with performing operations on the file, we need to properly close the file.
- Closing a file will free up the resources that were tied with the file. It is done using the `close()` method available in Python.
- Python has a garbage collector to clean up unreferenced objects but we must not rely on it to close the file.

```
f = open("test.txt", encoding = 'utf-8')
# perform file operations
f.close()
```

- This method is not entirely safe. If an exception occurs when we are performing some operation with the file, the code exits without closing the file.
- A safer way is to use a `try... finally` block.

```
try:
   f = open("test.txt", encoding = 'utf-8')
   # perform file operations
finally:
   f.close()
```

- This way, we are guaranteeing that the file is properly closed even if an exception is raised that causes program flow to stop.
- The best way to close a file is by using the `with` statement. This ensures that the file is closed when the block inside the `with` statement is exited.
- We don’t need to explicitly call the `close()` method. It is done internally.

```
with open("test.txt", encoding = 'utf-8') as f:
   # Perform file operations
```

Writing to Files in Python

- In order to write into a file in Python, we need to open it in write `w`, append `a` or exclusive creation `x` mode.
- We need to be careful with the w mode, as it will overwrite into the file if it already exists. Due to this, all the previous data are erased.
- Writing a string or sequence of bytes (for binary files) is done using the `write()` method. This method returns the number of characters written to the file.

```
with open("test.txt",'w', encoding = 'utf-8') as f:
   f.write("my first file\n")
   f.write("This file\n\n")
   f.write("contains three lines\n")
```

- This program will create a new file named `test.txt` in the current directory if it does not exist. If it does exist, it is overwritten.
- We must include the newline characters ourselves to distinguish the different lines.

Reading Files in Python

- To read a file in Python, we must open the file in reading `r` mode. As such, there are various methods available.
- Note that all these reading methods return empty values when the end of file (EOF) is reached.

`read()`

- We can use the `read(size)` method to read in the size number of data. If the `size` parameter is not specified, it reads and returns up to the end of the file.
- We can read the `text.txt` file we wrote in the above section in the following way:

```
f = open("test.txt",'r', encoding = 'utf-8')

# Read the first 4 data
f.read(4) # Returns 'This'

# Read the next 4 data,
f.read(4) # Returns ' is '
```

```
# Read in the rest till end of file
f.read() # Returns 'my first file\nThis file\ncontains three lines\n'

# Further reading returns empty sting
f.read() # Returns ''
```

- We can see that the `read()` method returns a newline as `'\n'`. Once the end of the file is reached, we get an empty string on further reading.
- We can change our current file cursor (position) using the `seek()` method. Similarly, the `tell()` method returns our current position (in number of bytes).

```
# Get the current file position
f.tell()  # Returns 56

# Vring file cursor to initial position
f.seek(0) # Returns 0

# Read the entire file
print(f.read())

# Outputs:
# This is my first file
# This file
# contains three lines
```

`for` Loop

- We can read a file line-by-line using a `for` loop. This is both efficient and fast. Note that with this setup, the lines in the file itself include a newline character `\n`. So, we use the `end` parameter of the `print()` function to avoid two newlines when printing.

```
for line in f:
     print(line, end = '')

# Outputs:
# This is my first file
# This file
# contains three lines
```

- Note that the above is a **common method for "lazy" reading** of big files in Python, especially when reading large files on a system with limited memory.

```
for line in open('really_big_file.dat'):
    process_data(line)
```

- Alternatively, you can just use `yield`:

```
def read_in_chunks(file_object, chunk_size=1024):
    """Lazy function (generator) to read a file piece by piece.
    Default chunk size: 1k."""
    while True:
        data = file_object.read(chunk_size)
        if not data:
            break
        yield data

with open('really_big_file.dat') as f:
    for piece in read_in_chunks(f):
        process_data(piece)
```

- Another option would be to use `iter` and a helper function:

```
f = open('really_big_file.dat')
def read1k():
    return f.read(1024)

for piece in iter(read1k, ''):
    process_data(piece)
```

`readline()`

- We can use the `readline()` method to read individual lines of a file. This method reads a file till the newline, including the newline character.

```
f.readline() # Returns 'This is my first file\n'

f.readline() # Returns 'This file\n'

f.readline() # Returns 'contains three lines\n'

f.readline() # Returns ''
```

`readlines()`

- `readlines()` returns a list of remaining lines of the entire file.

```
f.readlines() # Returns ['This is my first file\n', 'This file\n', 'contains three lines
\n']
```

Python File Methods

- There are various methods available with the file object. Some of them have been used in the above examples.
- Here is the complete list of methods in text mode with a brief description:

Method	Description
close()	Closes an opened file. It has no effect if the file is already closed.
detach()	Separates the underlying binary buffer from the `TextIOBase` and returns it.
fileno()	Returns an integer number (file descriptor) of the file.
flush()	Flushes the write buffer of the file stream.
isatty()	Returns `True` if the file stream is interactive.
read(n)	Reads at most n characters from the file. Reads till end of file if it is negative or `None`.
readable()	Returns `True` if the file stream can be read from.
readline(n=-1)	Reads and returns one line from the file. Reads in at most n bytes if specified.
readlines(n=-1)	Reads and returns a list of lines from the file. Reads in at most n bytes/characters if specified.
seek(offset,from=`SEEK_SET`)	Changes the file position to offset bytes, in reference to from (start, current, end).
seekable()	Returns `True` if the file stream supports random access.
tell()	Returns the current file location.
truncate(size=`None`)	Resizes the file stream to size bytes. If size is not specified, resizes to current location.
writable()	Returns `True` if the file stream can be written to.
write(s)	Writes the string s to the file and returns the number of characters written.

writelines(lines)	Writes a list of lines to the file.

Magic Methods

- Magic methods are special methods that you can define to add “magic” to your classes. They’re always surrounded by double leading and trailing underscores (e.g. `__init__()` or `__lt__()`).

Common Magic Methods

Construction and Initialization

- The most common magic method is `__init__()`. It’s the way that we can define the initialization behavior of an object. However, when we call `x = SomeClass()`, `__init__()` is not the first thing to get called.
- Actually, it’s a method called `__new__()`, which actually creates the instance, then passes any arguments at creation on to the initializer. At the other end of the object’s lifespan, there’s `__del__()`. Let’s take a closer look at these 3 magic methods:
    - `__new__()` is the first method to get called in an object’s instantiation. It takes the class, then any other arguments that it will pass along to `__init__()`. `__new__()` is used fairly rarely, but it does have its purposes, particularly when subclassing an immutable type like a tuple or a string. I don’t want to go in to too much detail on `__new__()` because it’s not too useful, but it is covered in great detail in the Python docs.
    - `__init__()` is the initializer for the class. It gets passed whatever the primary constructor was called with (so, for example, if we called `x = SomeClass(10, 'foo')`), `__init__()` would get passed `10` and `'foo'` as arguments. `__init__()` is almost universally used in Python class definitions.
    - `__del__()`: If `__new__()` and `__init__()` formed the constructor of the object, `__del__()` is the destructor. It doesn’t implement behavior for the statement `del x` (so that code would not translate to `x.__del__()`). Rather, it defines behavior for when an object is garbage collected. It can be quite useful for objects that might require extra cleanup upon deletion, like sockets or file objects. Be careful, however, as there is no guarantee that `__del__()` will be executed if the

object is still alive when the interpreter exits, so `__del__()` can't serve as a replacement for good coding practices (like always closing a connection when you're done with it. In fact, `__del__()` should almost never be used because of the precarious circumstances under which it is called; use it with caution!

Comparison Magic Methods

- `__cmp__()` is the most basic of the comparison magic methods. It actually implements behavior for all of the comparison operators (<, ==, !=, etc.), but it might not do it the way you want (for example, if whether one instance was equal to another were determined by one criterion and and whether an instance is greater than another were determined by something else).
- `__cmp__()` should return a negative integer if self < other, zero if self == other, and positive if self > other. It's usually best to define each comparison you need rather than define them all at once, but `__cmp__()` can be a good way to save repetition and improve clarity when you need all comparisons implemented with similar criteria.
- `__eq__()`: Defines behavior for the equality operator, `==`.
- `__ne__()`: Defines behavior for the inequality operator, `!=`.
- `__lt__()`: Defines behavior for the less-than operator, `<`.
- `__gt__()`: Defines behavior for the greater-than operator, `>`.
- `__le__()`: Defines behavior for the less-than-or-equal-to operator, `<=`.
- `__ge__()`: Defines behavior for the greater-than-or-equal-to operator, `>=`.

Numeric Magic Methods

- Just like you can create ways for instances of your class to be compared with comparison operators, you can define behavior for numeric operators.
- For organization's sake, we've split the numeric magic methods into 5 categories: unary operators, normal arithmetic operators, reflected arithmetic operators (more on this later), augmented assignment, and type conversions.

Unary Operators and Functions

- Unary operators and functions only have one operand, e.g. negation, absolute value, etc.
    - `__pos__()`: Implements behavior for unary positive (e.g. `+some_object`)
    - `__neg__()`: Implements behavior for negation (e.g. `some_object`)
    - `__abs__()`: Implements behavior for the built in `abs()` function.

Normal Arithmetic Operators

- Now, we cover the typical binary operators (and a function or two): `+`, `-`, `*` and the like. These are, for the most part, pretty self-explanatory.
    - `__add__()`: Implements addition.
    - `__sub__()`: Implements subtraction.
    - `__mul__()`: Implements multiplication.
    - `__floordiv__()`: Implements integer division using the `//` operator.
    - `__div__()`: Implements division using the `/` operator.

Creating Custom Sequences

- There's a number of ways to get your Python classes to act like built-in sequences (dict, tuple, list, str, etc.). These are some of the most powerful magic methods in Python because of the absurd degree of control they give you and the way that they magically make a whole array of global functions work beautifully on instances of your class.
- But before we get down to the good stuff, a quick word on requirements.

Requirements

- Now that we're talking about creating your own sequences in Python, it's time to talk about protocols. Protocols are somewhat similar to interfaces in other languages in that they give you a set of methods you must define. However, in Python protocols are totally informal and require no explicit declarations to implement. Rather, they're more like guidelines.
- Why are we talking about protocols now? Because implementing custom container types in Python involves using some of these protocols. First, there's the protocol

for defining immutable containers: to make an immutable container, you need only define `__len__()` and `__getitem__()` (more on these later). The mutable container protocol requires everything that immutable containers require plus `__setitem__()` and `__delitem__()`. Lastly, if you want your object to be iterable, you'll have to define `__iter__()`, which returns an iterator. That iterator must conform to an iterator protocol, which requires iterators to have methods called `__iter__()` (returning itself) and next.

The Magic Behind Containers

- Here are the magic methods that containers use:
  - `__len__()`: Returns the length of the container. Part of the protocol for both immutable and mutable containers.
  - `__getitem__()`: Defines behavior for when an item is accessed, using the notation `self[key]`. This is also part of both the mutable and immutable container protocols. It should also raise appropriate exceptions: `TypeError` if the type of the key is wrong and `KeyError` if there is no corresponding value for the key. More on exceptions in its section below.
  - `__setitem__()`: Defines behavior for when an item is assigned to, using the notation `self[nkey] = value`. This is part of the mutable container protocol. Again, you should raise `KeyError` and `TypeError` where appropriate.
  - `__delitem__()`: Defines behavior for when an item is deleted (e.g. `del self[key]`). This is only part of the mutable container protocol. You must raise the appropriate exceptions when an invalid key is used.
  - `__iter__()`: Should return an iterator for the container. Iterators are returned in a number of contexts, most notably by the `iter()` built in function and when a container is looped over using the form `for x in container:`. Iterators are their own objects, and they also must define an `__iter__()` method that returns self.

Example

- For our example, let's look at a list that implements some functional constructs:

```
class FunctionalList:
    '''A class wrapping a list with some extra functional magic, like head,
    tail, init, last, drop, and take.'''
```

```
    def __init__(self, values=None):
        if values is None:
            self.values = []
        else:
            self.values = values

    def __len__(self):
        return len(self.values)

    def __getitem__(self, key):
        # if key is of invalid type or value, the list values will raise the error
        return self.values[key]

    def __setitem__(self, key, value):
        self.values[key] = value

    def __delitem__(self, key):
        del self.values[key]

    def __iter__(self):
        return iter(self.values)
```

Making Operators Work on Custom Classes

- One of the biggest advantages of using Python's magic methods is that they provide a simple way to make objects behave like built-in types. That means you can avoid ugly, counter-intuitive, and nonstandard ways of performing basic operators. In some languages, it's common to do something like this:

```
if instance.equals(other_instance):
    # do something
```

- You could certainly do this in Python, too, but this adds confusion and is unnecessarily verbose. Different libraries might use different names for the same operations, making the client do way more work than necessary. With the power of magic methods, however, we can define one method (`__eq__()`, in this case), and say what we mean instead:

```
if instance == other_instance:
    # do something
```

- That’s part of the power of magic methods. The vast majority of them allow us to define meaning for operators so that we can use them on our own classes just like they were built in types.

Common Magic Methods Scenarios

- Some of the magic methods in Python directly map to built-in functions, in which case, how to invoke them is fairly obvious. However, in other cases, the invocation is far less obvious. The table below is devoted to exposing non-obvious syntax that leads to magic methods getting called.

Magic Method	When it gets invoked (example)	Explanation
`__new__(cls [,...])`	`obj = MyClass(arg1, arg2)`	Instance creation calls `__new__`
`__init__(self [,...])`	`obj = MyClass(arg1, arg2)`	Instance init calls `__init__`
`__cmp__(self, other)`	`self == other`, `self > other`, etc.	Called for comparisons between objects
`__pos__(self)`	`+self`	Unary plus sign
`__neg__(self)`	`-self`	Unary minus sign
`__invert__(self)`	`~self`	Bitwise inversion
`__index__(self)`	`x[self]`	Called when object is used as index
`__nonzero__(self)`	`bool(self)`	Boolean value of the object
`__getattr__(self, name)`	`self.name # name doesn't exist`	Accessing non-existent attribute
`__setattr__(self, name, val)`	`self.name = val`	Assigning to an attribute
`__delattr__(self, name)`	`del self.name`	Deleting an attribute
`__getattribute__(self, name)`	`self.name`	Accessing any attribute
`__getitem__(self, key)`	`self[key]`	Accessing an item using an index
`__setitem__(self, key, val)`	`self[key] = val`	Assigning an item using an index
`__delitem__(self, key)`	`del self[key]`	Deleting an item using an index
`__iter__(self)`	`for x in self`	Iteration
`__contains__(self, value)`	`value in self`, `value not in self`	Membership tests using `in`
`__call__(self [,...])`	`self(args)`	"Calling" an instance
`__enter__(self)`	`with self as x:`	

		`with` statement context manager
`__exit__(self, exc, val, trace)`	`with self as x:`	`with` statement context manager
`__getstate__(self)`	`pickle.dump(file, self)`	Pickling
`__setstate__(self)`	`data = pickle.load(file)`	Pickling

Exceptions

- An exception is an illegal operation that occurs during the execution of a program. Exceptions are known to non-programmers as instances that do not conform to a general rule.
- The name “exception” in computer science has this meaning as well – it implies that the problem (the exception) doesn’t occur frequently, i.e., the exception is the “exception to the rule”.

Exception Handling

- Exception handling is the process of responding to the occurrence of exceptions –- anomalous or exceptional conditions requiring special processing – during the execution of a program. Since exception handling ensures that the flow of the program doesn’t break when an exception occurs, it fosters robust code.
- Error handling is generally resolved by saving the state of execution at the moment the error occurred and interrupting the normal flow of the program to execute a special function or piece of code, which is known as the “exception handler”.
- Depending on the kind of error (“division by zero”, “file open error”, etc.) which has occurred, the error handler can “fix” the problem and the program can be continued afterwards with the previously saved data.
- Terminology:
  - The code, which harbors the risk of an exception, is embedded within a `try` block.
  - Exceptions are caught by an `except` clause.
  - `raise` statements generate exceptions.
- Let’s look at a simple example. Assume that we want to ask the user to enter an integer. If we use a `input()`, the input will be a string, which will need to be cast into

an integer. If the input isn't a valid integer, we will generate (raise) a `ValueError`.

```
n = int(input("Please enter a number: "))
```

- which outputs:

```
Please enter a number: 23.5
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-9-02fbe8840e1a> in <module>
----> 1 n = int(input("Please enter a number: "))

ValueError: invalid literal for int() with base 10: '23.5'
```

- With the aid of exception handling, we can write robust code for reading an integer from input:

```
while True:
    try:
        n = input("Please enter an integer: ")
        n = int(n)
        break
    except ValueError:
        print("No valid integer! Please try again ...")
print("Great, you successfully entered an integer!")
```

- which outputs:

```
Please enter an integer: abc
No valid integer! Please try again ...
Please enter an integer: 42.0
No valid integer! Please try again ...
Please enter an integer: 42
Great, you successfully entered an integer!
```

- It's a loop, which breaks only if a valid integer has been given. The while loop is entered. The code within the try clause will be executed statement by statement. If no exception occurs during the execution, the execution will reach the break statement and the while loop will be left.

- If an exception occurs, i.e., in the casting of `n`, the rest of the try block will be skipped and the except clause will be executed. The raised error, in this particular case a `ValueError`, has to match one of the names after except. After having printed the text of the `print` statement, the execution does another loop. It starts with a new `input()`.

Multiple `except` Clauses

- A `try` statement may have more than one except clause for different exceptions. But at most one except clause will be executed.
- Our next example shows a try clause, in which we open a file for reading, read a line from this file and convert this line into an integer. There are at least two possible exceptions:
- Just in case we have an additional unnamed except clause for an unexpected error:

```
import sys

try:
    f = open('integers.txt')
    s = f.readline()
    i = int(s.strip())
except IOError as e:
    errno, strerror = e.args
    print("I/O error({0}): {1}".format(errno,strerror))
    # e can be printed directly without using .args:
    # print(e)
except ValueError:
    print("No valid integer in line.")
except:
    print("Unexpected error:", sys.exc_info()[0])
    raise
```

- which outputs:

```
I/O error(2): No such file or directory
```

- The handling of the `IOError` in the previous example is of special interest. The except clause for the `IOError` specifies a variable `e` after the exception name (`IOError`).

- The variable `e` is bound to an exception instance with the arguments stored in `instance.args`.
- If we call the above script with a non-existing file, we get the message:

```
I/O error(2): No such file or directory
```

- And if the file `integers.txt` is not readable, say if we don't have the permission to read it, we get the following message:

```
I/O error(13): Permission denied
```

- An `except` clause may name more than one exception in a tuple of error names, as we see in the following example:

```
try:
    f = open('integers.txt')
    s = f.readline()
    i = int(s.strip())
except (IOError, ValueError):
    print("An I/O error or a ValueError occurred")
except:
    print("An unexpected error occurred")
    raise
```

- which outputs:

```
An I/O error or a ValueError occurred
```

- Here's what happens if we call a function within a `try` block and if an exception occurs inside the function call:

```
def f():
    x = int("four")

try:
    f()
except ValueError as e:
    print("got it :-) ", e)
```

```
print("Let's get on")
```

- which outputs:

```
got it :-)  invalid literal for int() with base 10: 'four'
Let's get on
```

- So, our function has caught the exception.
- We modify our example so that the function catches the exception directly:

```
def f():
    try:
        x = int("four")
    except ValueError as e:
        print("got it in the function :-) ", e)

try:
    f()
except ValueError as e:
    print("got it :-) ", e)

print("Let's get on")
```

- which outputs:

```
got it in the function :-)  invalid literal for int() with base 10: 'four'
Let's get on
```

- As expected, the exception is caught inside the function and not in the callers exception.
- We now add a `raise`, which generates the `ValueError` again, so that the exception will be propagated to the caller:

```
def f():
    try:
        x = int("four")
    except ValueError as e:
        print("got it in the function :-) ", e)
```

```
        raise

try:
    f()
except ValueError as e:
    print("got it :-) ", e)

print("Let's get on")
```

- which outputs:

```
got it in the function :-)  invalid literal for intT() with base 10: 'four'
got it :-)  invalid literal for int() with base 10: 'four'
Let's get on
```

Custom Exceptions

- It’s possible to create custom exceptions using the `raise` statement which forces a specified exception to occur:

```
raise SyntaxError("Sorry, my fault!")
Traceback (most recent call last):
```

- which outputs:

```
  File "interactiveshell.py", line 3326, in run_code
    exec(code_obj, self.user_global_ns, self.user_ns)

  File "<ipython-input-15-a5649918d59e>", line 1, in <module>
    raise SyntaxError("Sorry, my fault!")

  File "<string>", line unknown
SyntaxError: Sorry, my fault!
```

- The Pythonic way to do this is to define an exception class which inherits from the Exception class:
- which outputs:

```
raise MyException("An exception doesn't always prove the rule!")
------------------------------------------------------------------------
```

```
MyException                               Traceback (most recent call last)
<ipython-input-3-d75bff75fe3a> in <module>
      2     pass
      3
----> 4 raise MyException("An exception doesn't always prove the rule!")

MyException: An exception doesn't always prove the rule!
```

Clean-up Actions (`try` `finally`)

- So far the `try` statement had always been paired with except clauses. But there is another way to use it as well. The try statement can be followed by a `finally` clause.
- `finally` clauses are called clean-up or termination clauses, because they must be executed under all circumstances, i.e., a `finally` clause is always executed regardless if an exception occurred in a try block or not. A simple example to demonstrate the `finally` clause:

```
try:
    x = float(input("Your number: "))
    inverse = 1.0 / x
finally:
    print("There may or may not have been an exception.")
print("The inverse: ", inverse)
```

- which outputs:

```
Your number: 34
There may or may not have been an exception.
The inverse:  0.029411764705882353
```

Combining `try`, `except` and `finally`

- `finally` and `except` can be used together for the same `try` block:

```
try:
    x = float(input("Your number: "))
    inverse = 1.0 / x
except ValueError:
    print("You should have given either an int or a float")
except ZeroDivisionError:
```

```
    print("Infinity")
finally:
    print("There may or may not have been an exception.")
```

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 b
locks
    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")
```

- which outputs:

```
Your number: 23
There may or may not have been an exception.
```

`else` Clause

- The `try ... except` statement has an optional `else` clause. An `else` block has to be positioned after all the except clauses. An `else` clause will be executed if the `try` clause doesn't raise an exception.
- The following example opens a file and reads in all the lines into a list called "text":

```
import sys
file_name = sys.argv[1]
text = []
try:
    fh = open(file_name, 'r')
    text = fh.readlines()
    fh.close()
except IOError:
    print('cannot open', file_name)
```

```
if text:
    print(text[100])
```

- which outputs:
- This example receives the file name via a command line argument. So make sure that you call it properly: Let's assume that you saved this program as "exception_test.py". In this case, you have to call it with:

```
python exception_test.py integers.txt
```

- If you don't want this behavior, just change the line `file_name = sys.argv[1]` to `file_name = 'integers.txt'`.
- The previous example is nearly the same as:

```
import sys
file_name = sys.argv[1]
text = []
try:
    fh = open(file_name, 'r')
except IOError:
    print('cannot open', file_name)
else:
    text = fh.readlines()
    fh.close()

if text:
    print(text[100])
```

- which outputs:
- The main difference is that in the first case, all statements of the `try` block can lead to the same error message "cannot open …", which is wrong, if `fh.close()` or `fh.readlines()` raise an error.

`with` Statement

- Instead of `try` / `finally` to cleanup resources, you can use simply use a `with` context-manager:

```
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) # Prints {"aa": 12, "bb": 21}

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

Handling Nested Exceptions

- In this section, we cover re-raising exceptions in nested try/except blocks.
- As of Python 3 the traceback is stored in the exception, so a simple `raise e` will do the (mostly) right thing:

```
try:
    something()
except SomeError as e:
    try:
        plan_B()
    except AlsoFailsError:
        raise e  # or raise e from None - see below
```

- The traceback produced will include an additional notice that `SomeError` occurred while handling `AlsoFailsError` (because of `raise e` being inside except `AlsoFailsError`). This is misleading because what actually happened is the other way around - we encountered `AlsoFailsError`, and handled it, while trying to recover from `SomeError`. To obtain a traceback that doesn't include `AlsoFailsError`, replace `raise e` with `raise e from None`.

- In Python 2 you'd store the exception type, value, and traceback in local variables and use the three-argument form of raise:

```
try:
    something()
except SomeError:
    t, v, tb = sys.exc_info()
    try:
        plan_B()
    except AlsoFailsError:
        raise t, v, tb
```

Built-in Exceptions

- Some of the common built-in exceptions in Python programming along with the error that cause them are listed below. An exhaustive list of built-in exceptions in Python can be found in the Python documentation.

Exception	Cause of Error
`AssertionError`	Raised when an `assert` statement fails.
`AttributeError`	Raised when attribute assignment or reference fails.
`EOFError`	Raised when `input()` hits end-of-file condition.
`FloatingPointError`	Raised when a floating point operation fails.
`GeneratorExit`	Raise when a generator's `close()` method is called.
`ImportError`	Raised when the imported module is not found.
`IndexError`	Raised when the index of a sequence is out of range.
`KeyError`	Raised when a key is not found in a dictionary.
`KeyboardInterrupt`	Raised when the user hits the interrupt key (`Ctrl+C` or `Delete`).
`MemoryError`	Raised when an operation runs out of memory.
`NameError`	Raised when a variable is not found in local or global scope.
`NotImplementedError`	Raised by abstract methods.
`OSError`	Raised when system operation causes system related error.
`OverflowError`	Raised when the result of an arithmetic operation is too large to be represented.
`ReferenceError`	Raised when a weak reference proxy is used to access a garbage collected

	referent.
`RuntimeError`	Raised when an error does not fall under any other category.
`StopIteration`	Raised by `next()` to indicate that there is no further item to be returned by the iterator.
`SyntaxError`	Raised by parser when syntax error is encountered.
`IndentationError`	Raised when there is incorrect indentation.
`TabError`	Raised when indentation consists of inconsistent tabs and spaces.
`SystemError`	Raised when interpreter detects an internal error.
`SystemExit`	Raised by `sys.exit()` function.
`TypeError`	Raised when a function or operation is applied to an object of incorrect type.
`UnboundLocalError`	Raised when a reference is made to a local variable in a function or method, but no value has been bound to that variable.
`UnicodeDecodeError`	Raised when a Unicode-related error occurs during decoding.
`ValueError`	Raised when a function gets an argument of correct type but improper value.
`ZeroDivisionError`	Raised when the second operand of division or modulo operation is zero.

- We can view all the built-in exceptions using the built-in `local()` function as follows:

```
print(dir(locals()['__builtins__']))
```

- `locals()['__builtins__']` returns a module of built-in exceptions, functions, and attributes. `dir` allows us to list these attributes as strings.

Modules

- You can import modules:

```
import math
print(math.sqrt(16)) # Returns 4.0
```

- You can get specific functions from a module:

```
from math import ceil, floor
print(ceil(3.7))  # Returns 4.0
```

```
print(floor(3.7)) # Returns 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) # Returns 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 also find out which functions and attributes are defined in a module using `dir`, which we covered in detail in its section.

```
import math
dir(math)
```

- A gotcha with Python's module imports is that 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.

Modules vs. Packages

- A module is a single file that can be imported and used. In other words, any Python file is a module, its name being the file's base name without the `.py` extension. A module can be imported as:
- A package is a collection of modules in directories, containing an additional `__init__.py` file, to distinguish a package from a directory that just happens to contain a bunch of Python scripts. Packages can be nested to any depth, provided that the corresponding directories contain their own `__init__.py` file.

- Owing to multiple modules within a package (that give a package hierarchy), a package can be hierarchically imported as:

```
from my_package.timing.danger.internets import function_of_love
```

- Thus, regular modules in Python are just "**files**", while packages are "**directories**".
- The distinction between module and package seems to hold just at the file system level. When you import a module or a package, the corresponding object created by Python is always of type `module`. Note, however, when you import a package, only variables/functions/classes in the `__init__.py` file of that package are directly visible, not sub-packages or modules. As an example, consider the xml package in the Python standard library: its `xml` directory contains an `__init__.py` file and four sub-directories; the sub-directory `etree` contains an `__init__.py` file and, among others, an `ElementTree.py` file. See what happens when you try to interactively import package/modules:

```
>>> import xml
>>> type(xml)
<type 'module'>
>>> xml.etree.ElementTree
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'module' object has no attribute 'etree'
>>> import xml.etree
>>> type(xml.etree)
<type 'module'>
>>> xml.etree.ElementTree
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'module' object has no attribute 'ElementTree'
>>> import xml.etree.ElementTree
>>> type(xml.etree.ElementTree)
<type 'module'>
>>> xml.etree.ElementTree.parse
<function parse at 0x00B135B0>
```

- Regular modules can be "imported" and can be "executed" (as shown in the examples above), package modules also can be "imported" and can be "executed", however, you may rightly complain: "but we can't directly write code in directories! Code is written in files only!", and that's indeed a very good complaint, as it leads us

to the second special thing about package modules. The code for a package module is written in files inside its directory, and the names of these files are also reserved by Python. If you want to “import” a package module, you’ll have to put its code in an `__init__.py` file in its directory, and if you want to “execute” a package module, you’ll have to put the execution code of it in a `__main__.py` file in its directory.

- As an example:

```
# hierarchy of files and folders:
.
├── bar_pack/
│   ├── __init__.py
│   ├── __main__.py
│   foo.py
```

```
# bar_pack/__init__.py

def talk():
    print("bar")
```

```
# bar_pack/__main__.py

import __init__

__init__.talk()
```

```
# foo.py

import bar_pack # <-- importing package module "bar_pack"

bar_pack.talk() # <-- prints "bar"
```

```
# Run this command in the terminal:
python3 bar_pack # <-- executing the package module "bar_pack", prints "bar"
```

Classes

- The syntax for defining classes in Python is straightforward:

```
# We use the "class" statement to create a class
class Human:
    # A class attribute (shared by all instances of this class)
    species = "Homo sapiens"

    # Constructor
    # Note that all methods of a class take "self" as the first argument
    def __init__(self, name):
        # Assign the argument to the instance's name attribute
        self.name = name
        # Initialize property
        self._age = 0

    # Instance method
    def say(self, msg):
        print("{name}: {message}".format(name=self.name, message=msg))

    # Another instance method
    def greet(self, loud=False):
        if loud:
            print('HELLO, %s!'.format(self.name.upper()))
        else:
            print('Hello, %s'.format(self.name))

    # A class method is shared among all instances
    # They are called with the calling class as the first argument
    @classmethod
    def get_species(cls):
        return cls.species

    # A static method is called without a class or instance reference
    @staticmethod
    def grunt():
        return "*grunt*"

    # A property is just like a getter.
    # It turns the method "age()" into an read-only attribute of the same name.
    @property
    def age(self):
        return self._age

    # This allows the property to be set
    @age.setter
    def age(self, age):
        self._age = age

    # This allows the property to be deleted
    @age.deleter
    def age(self):
        del self._age
```

```
# When a Python interpreter reads a source file it executes all its code.
# This __name__ check makes sure this code block is only executed when this
# module is the main program.
if __name__ == '__main__':
    # Instantiate the class
    i = Human(name="Ian")
    i.say("hi")                     # Call an instance method; prints "Ian: hi"
    i.greet()                       # Call an instance method; prints "Hello, Ian"

    # Instantiate the class again
    j = Human("Joel")
    j.say("hello")                  # Call an instance method; prints "Joel: hello"
    j.greet(loud=True)              # Call an instance method; prints "HELLO, JOEL!"
    # i and j are instances of type Human, or in other words: they are "Human" objects

    # Call our class method
    i.say(i.get_species())          # Prints "Ian: H. sapiens"

    # Change the shared attribute
    Human.species = "Homo neanderthalensis"
    i.say(i.get_species())          # Prints "Ian: H. neanderthalensis"
    j.say(j.get_species())          # Prints "Joel: H. neanderthalensis"

    # Call the static method
    print(Human.grunt())            # Prints "*grunt*"

    # Cannot call static method with instance of object
    # because i.grunt() will automatically put "self" (the object i) as an argument
    print(i.grunt())                # Prints "TypeError: grunt() takes 0 positional argume
nts but 1 was given"

    # Update the property for this instance
    i.age = 42
    # Get the property
    i.say(i.age)                    # Prints "Ian: 42"
    j.say(j.age)                    # Prints "Joel: 0"
    # Delete the property
    del i.age

    # i.age                         # This would raise an AttributeError
```

Inheritance

- Inheritance allows new child classes to be defined that inherit methods and variables from their parent class.
- Using the Human class defined above as the base or parent class, we can define a child class, Superhero, which inherits the class variables like `species`, `name`, and

`age`, as well as methods, like `greet()` and `grunt()` from the Human class, but can also have its own unique properties.

- To take advantage of modularization by file you could place the classes above in their own files, say, `human.py`.
- To import functions from other files use the following format `from "filename-without-extension" import "function-or-class"`.
    - For our particular example: `from human import Human`.

```
from human import Human

# Specify the parent class(es) as parameters to the class definition
class Superhero(Human):

    # If the child class should inherit all of the parent's definitions without
    # any modifications, you can just use the "pass" keyword (and nothing else)
    # but in this case it is commented out to allow for a unique child class:
    # pass

    # Child classes can override their parents' attributes
    species = 'Superhuman'

    # Children automatically inherit their parent class's constructor including
    # its arguments, but can also define additional arguments or definitions
    # and override its methods such as the class constructor.
    # This constructor inherits the "name" argument from the "Human" class and
    # adds the "superpower" and "movie" arguments:
    def __init__(self, name, movie=False,
                 superpowers=["super strength", "bulletproofing"]):
        # add additional class attributes:
        self.fictional = True
        self.movie = movie
        # be aware of mutable default values, since defaults are shared
        self.superpowers = superpowers

        # The "super" function lets you access the parent class's methods
        # that are overridden by the child, in this case, the __init__ method.
        # This calls the parent class constructor:
        super().__init__(name)

    # override the greet method
    def greet(self, loud=False):
        if loud:
            print('HEY, %s!'.format(self.name.upper()))
        else:
            print('Hey, %s'.format(self.name))

    # add an additional instance method
```

```
    def boast(self):
        for power in self.superpowers:
            print("I wield the power of {pow}!".format(pow=power))

if __name__ == '__main__':
    sup = Superhero(name="Flash")

    # Instance type checks
    if isinstance(sup, Human):
        print('I am human')
    if type(sup) is Superhero:
        print('I am a superhero')

    # Get the Method Resolution search Order used by both getattr() and super()
    # This attribute is dynamic and can be updated
    print(Superhero.__mro__)    # Prints "(<class '__main__.Superhero'>,
                                # <class 'human.Human'>, <class 'object'>)"

    # Calls parent method but uses its own class attribute
    print(sup.get_species())    # Prints "Superhuman"

    # Calls overridden method
    print(sup.greet())          # Prints "HEY, Flash"

    # Calls method from Human
    sup.say('Spoon')            # Prints "Flash: Spoon"

    # Call method that exists only in Superhero
    sup.boast()                 # Prints "I wield the power of super strength!"
                                # Prints "I wield the power of bulletproofing!"

    # Inherited class attribute
    sup.age = 31
    print(sup.age)              # Returns 31

    # Attribute that only exists within Superhero
    print('Am I Oscar eligible? ' + str(sup.movie))
```

Multiple Inheritance

- Multiple inheritance can be best explained with an example.
- Assume we have a class `Bat` defined in `bat.py`:

```
# bat.py
class Bat:
    species = 'Baty'

    def __init__(self, can_fly=True):
        self.fly = can_fly
```

```
    # This class also has a say method
    def say(self, msg):
        msg = '... ... ...'
        return msg

    # And its own method as well
    def sonar(self):
        return ')))  ... ((('

if __name__ == '__main__':
    b = Bat()
    print(b.say('hello'))
    print(b.fly)
```

- And another class `Batman` that inherits from both `Superhero` and `Bat`:

```
# superhero.py
from superhero import Superhero
from bat import Bat

# Define Batman as a child that inherits from both Superhero and Bat
class Batman(Superhero, Bat):
    def __init__(self, *args, **kwargs):
        # Typically to inherit attributes you have to call super:
        # super(Batman, self).__init__(*args, **kwargs)
        # However we are dealing with multiple inheritance here, and super()
        # only works with the next base class in the MRO list.
        # So instead we explicitly call __init__ for all ancestors.
        # The use of *args and **kwargs allows for a clean way to pass arguments,
        # with each parent "peeling a layer of the onion".
        Superhero.__init__(self, 'anonymous', movie=True,
                           superpowers=['Wealthy'], *args, **kwargs)
        Bat.__init__(self, *args, can_fly=False, **kwargs)
        # override the value for the name attribute
        self.name = 'Sad Affleck'

    def greet(self, loud=False):
        if loud:
            print("I'M BATMAN!")
        else:
            print("I'm batman!")

if __name__ == '__main__':
    sup = Batman()

    # Get the Method Resolution search Order used by both getattr() and super().
    # This attribute is dynamic and can be updated
    print(Batman.__mro__)       # Prints "(<class '__main__.Batman'>,
                                # <class 'superhero.Superhero'>,
                                # <class 'human.Human'>,
```

```
                                # <class 'bat.Bat'>, <class 'object'>)"

    # Calls parent method but uses its own class attribute
    print(sup.get_species())    # Prints "Superhuman"

    # Calls overridden method
    print(sup.greet())          # Prints "I'M BATMAN!"

    # Calls method from Human, because inheritance order matters
    sup.say('I agree')          # Returns Sad Affleck: I agree

    # Call method that exists only in 2nd ancestor
    print(sup.sonar())          # Prints ")))  ... ((("

    # Inherited class attribute
    sup.age = 100
    print(sup.age)              # Prints "100"

    # Inherited attribute from 2nd ancestor whose default value was overridden.
    print('Can I fly? ' + str(sup.fly)) # Returns Can I fly? False
```

Selected Built-ins

- In this section, we present a selected set of commonly used built-in functions.
- For an exhaustive list of Python's built-in functions, refer the Python documentation.

`any` / `all`

- Python's `any` and `all` functions can be interpreted as a series of logical `or` and `and` operators, respectively.
- `any` returns `True` if any element of the iterable is true. If the iterable is empty, return `False`.
- `all` returns `True` if all elements of the iterable are true. If the iterable is empty, still return `True`.

```
any([0, 0.0, False, (), '0']), all([1, 0.0001, True, (False,)]) # Returns (True, True)
```

- If the iterables are empty, `any` returns `False`, and `all` returns `True`:

```
any([]), all([]) # Returns (False, True)
```

- Using the concepts we've seen so far,
- In summary,

	any	all
All True values	True	True
All False values	False	False
One True value (all others are False)	True	False
One False value (all others are True)	True	False
Empty Iterable	False	True

`dir`

- `dir()` is a powerful inbuilt function, which returns list of the attributes and methods of any object (say functions , modules, strings, lists, dictionaries etc.)
- When a class object is passed in as a parameter, `dir()` returns a list of names of the attributes contained in that object:

```
a = list() # same as initializing with []

# dir() will return all the attributes of the "arr" list object
dir(a) # Returns ['__add__', '__class__', '__contains__', '__delattr__', '__delitem__',
 ...]

d = dict() # same as initializing with {}
# dir() will return all the attributes of the "d" dictionary object
dir(d) # Returns ['__class__', '__contains__', '__delattr__', '__delitem__', '__dir__', '_
_doc__', ...]
```

- When a module is passed in as a parameter, it returns a list of the attributes contained in that module:

```
# import the random module
import random

# Prints a list of all the attributes in the random module
dir(random) # Returns ['BPF', 'LOG4', 'NV_MAGICCONST', 'RECIP_BPF', ...]
```

- When a user defined class object with an overridden `__dir__()` method is passed in as a parameter, `dir()` returns a list of the attributes contained in that object:

```
# Creating of a simple class with an overridden __dir()__ function
class Supermarket:
    # Function __dir()___ which list all the attributes to be used.
    def __dir__(self):
        return['customer_name', 'product',
               'quantity', 'price', 'date']

my_cart = Supermarket()

dir(my_cart) # Returns ['customer_name', 'date', 'price', 'product', 'quantity']
```

- If no parameters are passed, it returns a list of names in the current local scope:

```
# Note that we have not imported any modules
dir() # Returns ['__builtins__', '__cached__', '__doc__', '__file__', ...]

# Now let's import two modules
import random
import math

# dir() now returns the imported modules added to the local namespace including all the ex
isting ones as before
dir() # Returns ['__builtins__', '__cached__', '__doc__', '__file__', ..., 'math', 'rando
m']
```

- The most common use-case of `dir()` is for debugging since it helps list out all the attributes of the input parameter. This is especially useful when especially dealing a lot of classes and functions in a large project.
- `dir()` also offers information about the operations we can perform with the parameter that was passed in (class object, module, etc.), which can come in handy when you have little to no information about the parameter.

`eval`

- `eval()` lets a Python program run Python code within itself:

`filter`

- `filter()` constructs an iterator from those elements of the input iterable for which the input function returns `True`.

```
# Function that filters vowels
def filterVowels(variable):
    letters = ['a', 'e', 'i', 'o', 'u']
    if (variable in letters):
        return True
    else:
        return False

sequence = ['g', 'e', 'e', 'j', 'k', 's', 'p', 'r']

list(filter(filterVowels, sequence)) # Returns ['e', 'e']
```

- As a common use-case, `filter()` is normally used with lambda functions to operate on list, tuple, or sets.

```
# Define a list contains both even and odd numbers.
seq = [0, 1, 2, 3, 5, 8, 13]

# Result contains odd numbers of the list
result = filter(lambda x: x % 2 != 0, seq)
list(result) # Returns [1, 3, 5, 13]

# Result contains even numbers of the list
result = filter(lambda x: x % 2 == 0, seq)
list(result) # Returns [0, 2, 8]
```

`isinstance`

- `isinstance()` is used to check if the object argument is an instance of a certain class or any of its subclasses.

```
isinstance(3, int)          # Returns True
isinstance([1, 2, 3], list) # Returns True
isinstance("aa", str)       # Returns True
isinstance(u"as", unicode)  # Returns True
```

- You can even pass a tuple of classes to be checked for the object:

```
# Check if "Hello" is one of the types described in the tuple parameter
isinstance("Hello", (float, int, str, list, dict, tuple)) # Returns True
```

- `isinstance()` also works for objects of custom-defined classes:

```
# Parent class
class Vehicles:
    def __init__(self):
        return

# Child class
class Car(Vehicles):
    # Constructor
    def __init__(self):
        Vehicles.__init__('Car')

# initializing objects
v = Vehicles()
c = Car()

isinstance(v, Vehicles) # Returns True
isinstance(c, Car)      # Returns True
isinstance(v, Car)      # Returns False
isinstance(c, Vehicles) # Returns True
```

- A gotcha with `isinstance()` is that the `bool` datatype is a subclass of the `int` datatype:

```
isinstance(True, int)   # Returns True
isinstance(True, float) # Returns False
```

`issubclass`

- `issubclass()` is used to check if a class is a subclass of another class.

```
# Parent class
class Vehicles:
    def __init__(self):
        return

# Child class
class Car(Vehicles):
    # Constructor
    def __init__(self):
        Vehicles.__init__('Car')

# Driver's code
issubclass(Car, Vehicles) # Returns True
issubclass(Car, list)     # Returns False
```

- Similar to `isinstance()`, you can pass a tuple of classes to be checked for the class:

```
# Check if Car is a subclass of the types described in the type parameter
issubclass(Car, (list, Vehicles)) # Returns True
```

- Note that `issubclass()` considers a class to be a subclass of itself:

```
issubclass(list, list) # Returns True
```

- Again, similar to `isinstance()`, a gotcha with `issubclass()` is that the `bool` datatype is a subclass of the `int` datatype:

```
issubclass(bool, int) # Returns True
```

- Getting confused between `isinstance()` and `issubclass()` is a common pitfall. While both functions appear to be quite similar at a glance, keep in mind that their names themselves coveys their use-cases and thus, their differences.

`iter`

- `iter()` returns an iterator for the given object.
- Working of Python `iter()`:

```
# list of vowels
vowels = ['a', 'e', 'i', 'o', 'u']
vowels_iter = iter(vowels)

next(vowels_iter) # Returns 'a'
next(vowels_iter) # Returns 'e'
next(vowels_iter) # Returns 'i'
next(vowels_iter) # Returns 'o'
next(vowels_iter) # Returns 'u'
```

- `iter()` for custom objects:

```
class PrintNumber:
    def __init__(self, max):
        self.max = max
    def __iter__(self):
        self.num = 0
        return self
    def __next__(self):
        if(self.num >= self.max):
            raise StopIteration
        self.num += 1
        return self.num

print_num = PrintNumber(3)

print_num_iter = iter(print_num)
next(print_num_iter) # Returns '1'
next(print_num_iter) # Returns '2'
next(print_num_iter) # Returns '3'

# raises StopIteration
next(print_num_iter)
```

- `iter()` with sentinel parameter:

```
with open('mydata.txt') as fp:
    for line in iter(fp.readline, ''):
        processLine(line)
```

- When you run the program, it will open the mydata.txt file in reading mode.
- Then, the `iter(fp.readline, '')` in the `for` loop calls `readline` (which reads each line in the text file) until the sentinel character, `''` (empty string), is reached.

`len`

- `len()` returns the number of items in an object.
- Internally, `len()` calls the object's `__len__()` function. You can think of `len()` as:
- `len()` works with tuples, lists and range:

```
testList = []
len(testList) # Returns 0

testList = [1, 2, 3]
len(testList) # Returns 3
```

```
testTuple = (1, 2, 3)
len(testList) # Returns 3

testRange = range(1, 10)
len(testList) # Returns 9
```

- `len()` works with strings and bytes:

```
testString = ''
len(testString) # Returns 0

testString = 'Python'
len(testString) # Returns 6

# Byte object
testByte = b'Python'
len(testByte)   # Returns 6

# Converting to bytes object
testList = [1, 2, 3]
testByte = bytes(testList)
len(testByte)   # Returns 3
```

- `len()` works with dictionaries and sets:

```
testSet = {1, 2, 3}
len(testSet)       # Returns 3

# Empty Set
testSet = set()
len(testSet)       # Returns 0

testDict = {1: 'one', 2: 'two'}
len(testDict)      # Returns 2

testDict = {}
len(testDict)      # Returns 0

# frozenSet
testSet = {1, 2}
frozenTestSet = frozenset(testSet)
len(frozenTestSet) # Returns 2
```

- `len()` works for custom objects:

```
class Session:
    def __init__(self, number=0):
      self.number = number

    def __len__(self):
      return self.number

s1 = Session()
len(s1) # Returns 0 since the default length is 0

# giving custom length
s2 = Session(6)
len(s2) # Returns 6
```

`range`

- `range()` returns a sequence of numbers, within a given range. It accepts `start`, `stop` and `step` parameters. Only `stop` is required, while `start` defaults to 0 and `step` defaults to 1 when unspecified.
- `range()` is commonly used with `for` loops.

```
# Initializing list using range, 1 parameter only stop parameter
list(range(6))     # Returns [0, 1, 2, 3, 4, 5]

# Initializing list using range, 2 parameters only step and stop parameters
list(range(3, 6))  # Returns [3, 4, 5]

# Initializing list using range, 2 parameter only step and stop parameters
list(range(-6, 2)) # Returns [-6, -5, -4, -3, -2, -1, 0, 1]
```

- Demonstrating `range()` using step:

```
list(range(3, 10, 2))   # Returns [3, 5, 7, 9]

list(range(10, -5, -3)) # Returns [10, 7, 4, 1, -2]

list(range(10, -5, 3))  # Returns []

# list(range(3, 7, 0))  # Returns ValueError: range() arg 3 must not be zero
```

`reversed`

- `reversed()` reverses the order of a list's contents in place.

```
# A list of numbers
L1 = [1, 2, 3, 4, 1, 2, 6]
L1.reverse()
print(L1) # Prints [6, 2, 1, 4, 3, 2, 1]

# A list of characters
L2 = ['a', 'b', 'c', 'd', 'a', 'a']
L2.reverse()
print(L2) # Prints ['a', 'a', 'd', 'c', 'b', 'a']
```

- Note that `reversed()` only supports lists. Datatypes other than a list return an `AttributeError`:

```
string = "abgedge"
string.reverse()
print(string) # Returns AttributeError: 'str' object has no attribute 'reverse'
```

`sort`

Basics

- `sort()` returns a new sorted list from the items in iterable.
- To perform a simple ascending sort, just call the `sorted()` function without any input arguments. It returns a new sorted list:

```
sorted([5, 2, 3, 1, 4]) # Returns [1, 2, 3, 4, 5]
```

- You can also use the `<list>.sort()` function of a list. It modifies the list **in-place** (and returns `None` to avoid confusion). Usually it's less convenient than `sorted()`, but if you don't need the original list, it's slightly more efficient.

```
a = [5, 2, 3, 1, 4]
a.sort()
a # Returns [1, 2, 3, 4, 5]
```

- Another difference is that `list.sort()` is only defined for lists. In contrast, the `sorted()` function accepts any iterable.

```
sorted({1: 'D', 2: 'B', 3: 'B', 4: 'E', 5: 'A'}) # Returns [1, 2, 3, 4, 5]
```

Sort Key

- Both `list.sort()` and `sorted()` added a `key` parameter to specify a function to be called on each list element prior to making comparisons.
- For example, here's a case-insensitive string comparison:

```
sorted("This is a test string".split(), key=str.lower) # Returns ['a', 'Andrew', 'from',
 'is', 'string', 'test', 'This']
```

- The value of the key parameter should be a function that takes a single argument and returns a key to use for sorting purposes. This technique is fast because the key function is called exactly once for each input record.
- A common pattern is to sort complex objects using some of the object's indices as a key. You can use a `lambda` function to access the relevant index of the object's attributes. Since these access patterns are very common, Python provides convenience functions to make accessor functions easier and faster. The operator module offers `itemgetter`, which serves this role. For example:

```
from operator import itemgetter

class Student:
    def __init__(self, name, grade, age):
        self.name = name
        self.grade = grade
        self.age = age
    def __repr__(self):
        return repr((self.name, self.grade, self.age))
    def weighted_grade(self):
        return 'CBA'.index(self.grade) / float(self.age)

student_tuples = [('john', 'A', 15), ('jane', 'B', 12), ('dave', 'B', 10)]

# Sort by age
sorted(student_tuples, key=lambda student: student[2]) # Returns [('dave', 'B', 10), ('jan
e', 'B', 12), ('john', 'A', 15)]

# Sort by age
sorted(student_tuples, key=itemgetter(2))              # Returns [('dave', 'B', 10), ('jan
e', 'B', 12), ('john', 'A', 15)]
```

- The same technique works for objects with named attributes. Similar to the above case, you can use a `lambda` function to access the named attributes within the object. Alternatively, you can use `attrgetter`, which is offered by the operator module. For example:

```
from operator import attrgetter

class Student:
    def __init__(self, name, grade, age):
        self.name = name
        self.grade = grade
        self.age = age
    def __repr__(self):
        return repr((self.name, self.grade, self.age))
    def weighted_grade(self):
        return 'CBA'.index(self.grade) / float(self.age)

student_tuples = [('john', 'A', 15), ('jane', 'B', 12), ('dave', 'B', 10)]
student_objects = [Student('john', 'A', 15), Student('jane', 'B', 12), Student('dave',
 'B', 10)]

# Sort by age
sorted(student_objects, key=lambda student: student.age) # Returns [('dave', 'B', 10), ('j
ane', 'B', 12), ('john', 'A', 15)]

# Same effect as above
sorted(student_objects, key=attrgetter('age'))           # Returns [('dave', 'B', 10), ('j
ane', 'B', 12), ('john', 'A', 15)]
```

- Note that compared to `lambda` functions, `itemgetter` offers succinct syntax, for example, if you need to get a number of elements at once. For instance,
- is the same as:
- Both `list.sort()` and `sorted()` accept a `reverse` parameter with a boolean value, which can serve as a flag for descending sorts. For example, to get the student data in reverse age order:

```
from operator import itemgetter, attrgetter

class Student:
    def __init__(self, name, grade, age):
        self.name = name
        self.grade = grade
```

```
        self.age = age
    def __repr__(self):
        return repr((self.name, self.grade, self.age))
    def weighted_grade(self):
        return 'CBA'.index(self.grade) / float(self.age)

student_tuples = [('john', 'A', 15), ('jane', 'B', 12), ('dave', 'B', 10)]
student_objects = [Student('john', 'A', 15), Student('jane', 'B', 12), Student('dave',
 'B', 10)]

# Reverse sort by age
sorted(student_tuples, key=itemgetter(2), reverse=True)                # Returns [('john',
'A', 15), ('jane', 'B', 12), ('dave', 'B', 10)]

# Same effect as above
sorted(student_objects, key=lambda student: student.age, reverse=True) # Returns [('john',
'A', 15), ('jane', 'B', 12), ('dave', 'B', 10)]

# Same effect as above
sorted(student_objects, key=attrgetter('age'), reverse=True)           # Returns [('john',
'A', 15), ('jane', 'B', 12), ('dave', 'B', 10)]
```

- For sorting examples and a brief sorting tutorial, see Python's Sorting HowTo wiki.

`zip`

- `zip()` aggregates elements from each of its inputs. It returns a zip object, which is an iterator of tuples where the tuple contains the element from each of the input iterables.
- With a single iterable argument, it returns an iterator of single-length tuples. With no arguments, it returns an empty iterator.
- As a simple example,

```
a = [1, 2]
b = [4, 5]

print(list(zip(a, b))) # Prints [(1, 4), (2, 5)]
```

- `zip()` in conjunction with the `*` operator can be used to unzip a list:

```
x = [1, 2, 3]
y = [4, 5, 6]
zipped = zip(x, y)
print(list(zipped))             # Prints [(1, 4), (2, 5), (3, 6)]
```

```
x2, y2 = zip(*zip(x, y))
x == list(x2) and y == list(y2) # Returns True
```

- `zip()` can work with iterables of different lengths, in which case the iterator with the least items decides the length of its output. In other words, `zip()` stops iterating when the shortest-length input iterable is exhausted.
    - `zip()` should only be used with unequal length inputs when you don't care about trailing, unmatched values from the longer iterables. If those values are important, use `itertools.zip_longest()` instead.
- To read more about the `zip()` function, refer Python's built-in functions documentation.