The Zen of Python

From the PEP 20 – The Zen of Python:

> Long time Pythoneer Tim Peters succinctly channels the BDFL's guiding principles for Python's design into 20 aphorisms, only 19 of which have been written down.

```
>>> import this
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!
```

Python Basics

Math Operators

From **Highest** to **Lowest** precedence:

Operators	Operation	Example
**	Exponent	`2 ** 3 = 8`
%	Modulus/Remainder	`22 % 8 = 6`
//	Integer division	`22 // 8 = 2`
/	Division	`22 / 8 = 2.75`
*	Multiplication	`3 * 3 = 9`
-	Subtraction	`5 - 2 = 3`
+	Addition	`2 + 2 = 4`

Examples of expressions in the interactive shell:

```
>>> 2 + 3 * 6
20
```

```
>>> (2 + 3) * 6
30
```

```
>>> 2 ** 8
256
```

```
>>> 23 // 7
3
```

```
>>> 23 % 7
2
```

```
>>> (5 - 1) * ((7 + 1) / (3 - 1))
16.0
```

Data Types

Data Type	Examples
Integers	`-2, -1, 0, 1, 2, 3, 4, 5`
Floating-point numbers	`-1.25, -1.0, --0.5, 0.0, 0.5, 1.0, 1.25`
Strings	`'a', 'aa', 'aaa', 'Hello!', '11 cats'`

String Concatenation and Replication

String concatenation:

```
>>> 'Alice' 'Bob'
'AliceBob'
```

Note: Avoid `+` operator for string concatenation. Prefer string formatting.

String Replication:

```
>>> 'Alice' * 5
'AliceAliceAliceAliceAlice'
```

Variables

You can name a variable anything as long as it obeys the following rules:

1. It can be only one word.
2. It can use only letters, numbers, and the underscore (`_`) character.
3. It can't begin with a number.
4. Variable name starting with an underscore (`_`) are considered as "unuseful`.

Example:

```
>>> spam = 'Hello'
>>> spam
'Hello'
```

```
>>> _spam = 'Hello'
```

`_spam` should not be used again in the code.

Comments

Inline comment:

```
# This is a comment
```

Multiline comment:

```
# This is a
# multiline comment
```

Code with a comment:

```
a = 1  # initialization
```

Please note the two spaces in front of the comment.

Function docstring:

```
def foo():
    """
    This is a function docstring
    You can also use:
    ''' Function Docstring '''
    """
```

The print() Function

```
>>> print('Hello world!')
Hello world!
```

```
>>> a = 1
>>> print('Hello world!', a)
Hello world! 1
```

The input() Function

Example Code:

```
>>> print('What is your name?')   # ask for their name
>>> myName = input()
>>> print('It is good to meet you, {}'.format(myName))
What is your name?
Al
It is good to meet you, Al
```

The len() Function

Evaluates to the integer value of the number of characters in a string:

```
>>> len('hello')
5
```

Note: test of emptiness of strings, lists, dictionary, etc, should **not** use len, but prefer direct boolean evaluation.

```
>>> a = [1, 2, 3]
>>> if a:
>>>     print("the list is not empty!")
```

The str(), int(), and float() Functions

Integer to String or Float:

```
>>> str(29)
'29'
```

```
>>> print('I am {} years old.'.format(str(29)))
I am 29 years old.
```

```
>>> str(-3.14)
'-3.14'
```

Float to Integer:

```
>>> int(7.7)
7
```

```
>>> int(7.7) + 1
8
```

Flow Control

Comparison Operators

Operator	Meaning
`==`	Equal to
`!=`	Not equal to
`<`	Less than
`>`	Greater Than
`<=`	Less than or Equal to
`>=`	Greater than or Equal to

These operators evaluate to True or False depending on the values you give them.

Examples:

```
>>> 42 == 42
True
```

```
>>> 40 == 42
False
```

```
>>> 'hello' == 'hello'
True
```

```
>>> 'hello' == 'Hello'
False
```

```
>>> 'dog' != 'cat'
True
```

```
>>> 42 == 42.0
True
```

```
>>> 42 == '42'
False
```

Boolean evaluation

Never use `==` or `!=` operator to evaluate boolean operation. Use the `is` or `is not` operators, or use implicit boolean evaluation.

NO (even if they are valid Python):

```
>>> True == True
True
```

```
>>> True != False
True
```

YES (even if they are valid Python):

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

```
>>> True is not False
True
```

These statements are equivalent:

```
>>> if a is True:
>>>    pass
>>> if a is not False:
>>>    pass
>>> if a:
>>>    pass
```

And these as well:

```
>>> if a is False:
>>>    pass
>>> if a is not True:
>>>    pass
>>> if not a:
>>>    pass
```

Boolean Operators

There are three Boolean operators: and, or, and not.

The *and* Operator's *Truth* Table:

Expression	Evaluates to
`True and True`	`True`
`True and False`	`False`
`False and True`	`False`
`False and False`	`False`

The *or* Operator's *Truth* Table:

Expression	Evaluates to
`True or True`	`True`
`True or False`	`True`
`False or True`	`True`
`False or False`	`False`

The *not* Operator's *Truth* Table:

Expression	Evaluates to
`not True`	`False`

Expression	Evaluates to
`not False`	`True`

Mixing Boolean and Comparison Operators

```
>>> (4 < 5) and (5 < 6)
True
```

```
>>> (4 < 5) and (9 < 6)
False
```

```
>>> (1 == 2) or (2 == 2)
True
```

You can also use multiple Boolean operators in an expression, along with the comparison operators:

```
>>> 2 + 2 == 4 and not 2 + 2 == 5 and 2 * 2 == 2 + 2
True
```

if Statements

```
if name == 'Alice':
    print('Hi, Alice.')
```

else Statements

```
name = 'Bob'
if name == 'Alice':
    print('Hi, Alice.')
else:
    print('Hello, stranger.')
```

elif Statements

```
name = 'Bob'
age = 5
if name == 'Alice':
    print('Hi, Alice.')
elif age < 12:
    print('You are not Alice, kiddo.')
```

```
name = 'Bob'
age = 30
if name == 'Alice':
    print('Hi, Alice.')
elif age < 12:
    print('You are not Alice, kiddo.')
else:
    print('You are neither Alice nor a little kid.')
```

while Loop Statements

```
spam = 0
while spam < 5:
    print('Hello, world.')
    spam = spam + 1
```

break Statements

If the execution reaches a break statement, it immediately exits the while loop's clause:

```
while True:
    print('Please type your name.')
    name = input()
    if name == 'your name':
        break
print('Thank you!')
```

continue Statements

When the program execution reaches a continue statement, the program execution immediately jumps back to the start of the loop.

```
while True:
    print('Who are you?')
    name = input()
    if name != 'Joe':
        continue
    print('Hello, Joe. What is the password? (It is a fish.)')
    password = input()
    if password == 'swordfish':
        break
print('Access granted.')
```

for Loops and the range() Function

```
>>> print('My name is')
>>> for i in range(5):
>>>     print('Jimmy Five Times ({})'.format(str(i)))
My name is
Jimmy Five Times (0)
Jimmy Five Times (1)
Jimmy Five Times (2)
Jimmy Five Times (3)
Jimmy Five Times (4)
```

The *range()* function can also be called with three arguments. The first two arguments will be the start and stop values, and the third will be the step argument. The step is the amount that the variable is increased by after each iteration.

```
>>> for i in range(0, 10, 2):
>>>    print(i)
0
2
4
6
8
```

You can even use a negative number for the step argument to make the for loop count down instead of up.

```
>>> for i in range(5, -1, -1):
>>>     print(i)
5
4
3
2
1
0
```

For else statement

This allows to specify a statement to execute in case of the full loop has been executed. Only useful when a `break` condition can occur in the loop:

```
>>> for i in [1, 2, 3, 4, 5]:
>>>    if i == 3:
>>>        break
>>> else:
>>>    print("only executed when no item of the list is equal to 3")
```

Importing Modules

```
import random
for i in range(5):
    print(random.randint(1, 10))
```

```
import random, sys, os, math
```

```
from random import *
```

Ending a Program Early with sys.exit()

```
import sys

while True:
    print('Type exit to exit.')
    response = input()
    if response == 'exit':
        sys.exit()
    print('You typed {}.'.format(response))
```

Functions

```
>>> def hello(name):
>>>     print('Hello {}'.format(name))
>>>
>>> hello('Alice')
>>> hello('Bob')
Hello Alice
Hello Bob
```

Return Values and return Statements

When creating a function using the def statement, you can specify what the return value should be with a return statement. A return statement consists of the following:

- The return keyword.
- The value or expression that the function should return.

```
import random
def getAnswer(answerNumber):
    if answerNumber == 1:
        return 'It is certain'
    elif answerNumber == 2:
        return 'It is decidedly so'
    elif answerNumber == 3:
        return 'Yes'
    elif answerNumber == 4:
        return 'Reply hazy try again'
    elif answerNumber == 5:
        return 'Ask again later'
    elif answerNumber == 6:
        return 'Concentrate and ask again'
    elif answerNumber == 7:
        return 'My reply is no'
    elif answerNumber == 8:
        return 'Outlook not so good'
    elif answerNumber == 9:
        return 'Very doubtful'

r = random.randint(1, 9)
fortune = getAnswer(r)
print(fortune)
```

The None Value

```
>>> spam = print('Hello!')
Hello!
```

```
>>> spam is None
True
```

Note: never compare to `None` with the `==` operator. Always use `is`.

Keyword Arguments and print()

```
>>> print('Hello', end='')
>>> print('World')
HelloWorld
```

```
>>> print('cats', 'dogs', 'mice')
cats dogs mice
```

```
>>> print('cats', 'dogs', 'mice', sep=',')
cats,dogs,mice
```

Local and Global Scope

- Code in the global scope cannot use any local variables.
- However, a local scope can access global variables.
- Code in a function's local scope cannot use variables in any other local scope.
- You can use the same name for different variables if they are in different scopes. That is, there can be a local variable named spam and a global variable also named spam.

The global Statement

If you need to modify a global variable from within a function, use the global statement:

```
>>> def spam():
>>>     global eggs
>>>     eggs = 'spam'
>>>
>>> eggs = 'global'
>>> spam()
>>> print(eggs)
spam
```

There are four rules to tell whether a variable is in a local scope or global scope:

1. If a variable is being used in the global scope (that is, outside of all functions), then it is always a global variable.
2. If there is a global statement for that variable in a function, it is a global variable.
3. Otherwise, if the variable is used in an assignment statement in the function, it is a local variable.
4. But if the variable is not used in an assignment statement, it is a global variable.

Exception Handling

Basic exception handling

```
>>> def spam(divideBy):
>>>     try:
>>>         return 42 / divideBy
>>>     except ZeroDivisionError as e:
>>>         print('Error: Invalid argument: {}'.format(e))
>>>
>>> print(spam(2))
>>> print(spam(12))
>>> print(spam(0))
>>> print(spam(1))
21.0
3.5
Error: Invalid argument: division by zero
None
42.0
```

Final code in exception handling

Code inside the `finally` section is always executed, no matter if an exception has been raised or not, and even if an exception is not caught.

```
>>> def spam(divideBy):
>>>     try:
>>>         return 42 / divideBy
>>>     except ZeroDivisionError as e:
>>>         print('Error: Invalid argument: {}'.format(e))
>>>     finally:
>>>         print("-- division finished --")
>>> print(spam(2))
-- division finished --
21.0
>>> print(spam(12))
-- division finished --
3.5
>>> print(spam(0))
Error: Invalid Argument division by zero
-- division finished --
None
>>> print(spam(1))
-- division finished --
42.0
```

Lists

```
>>> spam = ['cat', 'bat', 'rat', 'elephant']

>>> spam
['cat', 'bat', 'rat', 'elephant']
```

Getting Individual Values in a List with Indexes

```
>>> spam = ['cat', 'bat', 'rat', 'elephant']
>>> spam[0]
'cat'
```

```
>>> spam[1]
'bat'
```

```
>>> spam[2]
'rat'
```

```
>>> spam[3]
'elephant'
```

Negative Indexes

```
>>> spam = ['cat', 'bat', 'rat', 'elephant']
>>> spam[-1]
'elephant'
```

```
>>> spam[-3]
'bat'
```

```
>>> 'The {} is afraid of the {}.'.format(spam[-1], spam[-3])
'The elephant is afraid of the bat.'
```

Getting Sublists with Slices

```
>>> spam = ['cat', 'bat', 'rat', 'elephant']
>>> spam[0:4]
['cat', 'bat', 'rat', 'elephant']
```

```
>>> spam[1:3]
['bat', 'rat']
```

```
>>> spam[0:-1]
['cat', 'bat', 'rat']
```

```
>>> spam = ['cat', 'bat', 'rat', 'elephant']
>>> spam[:2]
['cat', 'bat']
```

```
>>> spam[1:]
['bat', 'rat', 'elephant']
```

Slicing the complete list will perform a copy:

```
>>> spam2 = spam[:]
['cat', 'bat', 'rat', 'elephant']
>>> spam.append('dog')
>>> spam
['cat', 'bat', 'rat', 'elephant', 'dog']
>>> spam2
['cat', 'bat', 'rat', 'elephant']
```

Getting a List's Length with len()

```
>>> spam = ['cat', 'dog', 'moose']
>>> len(spam)
3
```

Changing Values in a List with Indexes

```
>>> spam = ['cat', 'bat', 'rat', 'elephant']
>>> spam[1] = 'aardvark'

>>> spam
['cat', 'aardvark', 'rat', 'elephant']

>>> spam[2] = spam[1]

>>> spam
['cat', 'aardvark', 'aardvark', 'elephant']

>>> spam[-1] = 12345

>>> spam
['cat', 'aardvark', 'aardvark', 12345]
```

List Concatenation and List Replication

```
>>> [1, 2, 3] + ['A', 'B', 'C']
[1, 2, 3, 'A', 'B', 'C']

>>> ['X', 'Y', 'Z'] * 3
['X', 'Y', 'Z', 'X', 'Y', 'Z', 'X', 'Y', 'Z']

>>> spam = [1, 2, 3]

>>> spam = spam + ['A', 'B', 'C']

>>> spam
[1, 2, 3, 'A', 'B', 'C']
```

Removing Values from Lists with del Statements

```
>>> spam = ['cat', 'bat', 'rat', 'elephant']
>>> del spam[2]
>>> spam
['cat', 'bat', 'elephant']
```

```
>>> del spam[2]
>>> spam
['cat', 'bat']
```

Using for Loops with Lists

```
>>> supplies = ['pens', 'staplers', 'flame-throwers', 'binders']
>>> for i, supply in enumerate(supplies):
>>>     print('Index {} in supplies is: {}'.format(str(i), supply))
Index 0 in supplies is: pens
Index 1 in supplies is: staplers
Index 2 in supplies is: flame-throwers
Index 3 in supplies is: binders
```

Looping Through Multiple Lists with zip()

```
>>> name = ['Pete', 'John', 'Elizabeth']
>>> age = [6, 23, 44]
>>> for n, a in zip(name, age):
>>>     print('{} is {} years old'.format(n, a))
Pete is 6 years old
John is 23 years old
Elizabeth is 44 years old
```

The in and not in Operators

```
>>> 'howdy' in ['hello', 'hi', 'howdy', 'heyas']
True
```

```
>>> spam = ['hello', 'hi', 'howdy', 'heyas']
>>> 'cat' in spam
False
```

```
>>> 'howdy' not in spam
False
```

```
>>> 'cat' not in spam
True
```

The Multiple Assignment Trick

The multiple assignment trick is a shortcut that lets you assign multiple variables with the values in a list in one line of code. So instead of doing this:

```
>>> cat = ['fat', 'orange', 'loud']

>>> size = cat[0]

>>> color = cat[1]

>>> disposition = cat[2]
```

You could type this line of code:

```
>>> cat = ['fat', 'orange', 'loud']

>>> size, color, disposition = cat
```

The multiple assignment trick can also be used to swap the values in two variables:

```
>>> a, b = 'Alice', 'Bob'
>>> a, b = b, a
>>> print(a)
'Bob'
```

```
>>> print(b)
'Alice'
```

Augmented Assignment Operators

Operator	Equivalent
`spam += 1`	`spam = spam + 1`
`spam -= 1`	`spam = spam - 1`
`spam *= 1`	`spam = spam * 1`
`spam /= 1`	`spam = spam / 1`
`spam %= 1`	`spam = spam % 1`

Examples:

```
>>> spam = 'Hello'
>>> spam += ' world!'
>>> spam
'Hello world!'

>>> bacon = ['Zophie']
>>> bacon *= 3
>>> bacon
['Zophie', 'Zophie', 'Zophie']
```

Finding a Value in a List with the index() Method

```
>>> spam = ['Zophie', 'Pooka', 'Fat-tail', 'Pooka']

>>> spam.index('Pooka')
1
```

Adding Values to Lists with the append() and insert() Methods

append():

```
>>> spam = ['cat', 'dog', 'bat']

>>> spam.append('moose')

>>> spam
['cat', 'dog', 'bat', 'moose']
```

insert():

```
>>> spam = ['cat', 'dog', 'bat']

>>> spam.insert(1, 'chicken')

>>> spam
['cat', 'chicken', 'dog', 'bat']
```

Removing Values from Lists with remove()

```
>>> spam = ['cat', 'bat', 'rat', 'elephant']

>>> spam.remove('bat')

>>> spam
['cat', 'rat', 'elephant']
```

If the value appears multiple times in the list, only the first instance of the value will be removed.

Sorting the Values in a List with the sort() Method

```
>>> spam = [2, 5, 3.14, 1, -7]
>>> spam.sort()
>>> spam
[-7, 1, 2, 3.14, 5]
```

```
>>> spam = ['ants', 'cats', 'dogs', 'badgers', 'elephants']
>>> spam.sort()
>>> spam
['ants', 'badgers', 'cats', 'dogs', 'elephants']
```

You can also pass True for the reverse keyword argument to have sort() sort the values in reverse order:

```
>>> spam.sort(reverse=True)
>>> spam
['elephants', 'dogs', 'cats', 'badgers', 'ants']
```

If you need to sort the values in regular alphabetical order, pass str. lower for the key keyword argument in the sort() method call:

```
>>> spam = ['a', 'z', 'A', 'Z']
>>> spam.sort(key=str.lower)
>>> spam
['a', 'A', 'z', 'Z']
```

You can use the built-in function `sorted` to return a new list:

```
>>> spam = ['ants', 'cats', 'dogs', 'badgers', 'elephants']
>>> sorted(spam)
['ants', 'badgers', 'cats', 'dogs', 'elephants']
```

Tuple Data Type

```
>>> eggs = ('hello', 42, 0.5)
>>> eggs[0]
'hello'
```

```
>>> eggs[1:3]
(42, 0.5)
```

```
>>> len(eggs)
3
```

The main way that tuples are different from lists is that tuples, like strings, are immutable.

Converting Types with the list() and tuple() Functions

```
>>> tuple(['cat', 'dog', 5])
('cat', 'dog', 5)
```

```
>>> list(('cat', 'dog', 5))
['cat', 'dog', 5]
```

```
>>> list('hello')
['h', 'e', 'l', 'l', 'o']
```

Dictionaries and Structuring Data

Example Dictionary:

```
myCat = {'size': 'fat', 'color': 'gray', 'disposition': 'loud'}
```

The keys(), values(), and items() Methods

values():

```
>>> spam = {'color': 'red', 'age': 42}
>>> for v in spam.values():
>>>     print(v)
red
42
```

keys():

```
>>> for k in spam.keys():
>>>     print(k)
color
age
```

items():

```
>>> for i in spam.items():
>>>     print(i)
('color', 'red')
('age', 42)
```

Using the keys(), values(), and items() methods, a for loop can iterate over the keys, values, or key-value pairs in a dictionary, respectively.

```
>>> spam = {'color': 'red', 'age': 42}
>>>
>>> for k, v in spam.items():
>>>     print('Key: {} Value: {}'.format(k, str(v)))
Key: age Value: 42
Key: color Value: red
```

Checking Whether a Key or Value Exists in a Dictionary

```
>>> spam = {'name': 'Zophie', 'age': 7}
```

```
>>> 'name' in spam.keys()
True
```

```
>>> 'Zophie' in spam.values()
True
```

```
>>> # You can omit the call to keys() when checking for a key
>>> 'color' in spam
False
```

```
>>> 'color' not in spam
True
```

The get() Method

Get has two parameters: key and default value if the key did not exist

```
>>> picnic_items = {'apples': 5, 'cups': 2}

>>> 'I am bringing {} cups.'.format(str(picnic_items.get('cups', 0)))
'I am bringing 2 cups.'
```

```
>>> 'I am bringing {} eggs.'.format(str(picnic_items.get('eggs', 0)))
'I am bringing 0 eggs.'
```

The setdefault() Method

Let's consider this code:

```
spam = {'name': 'Pooka', 'age': 5}

if 'color' not in spam:
    spam['color'] = 'black'
```

Using `setdefault` we could write the same code more succinctly:

```
>>> spam = {'name': 'Pooka', 'age': 5}
>>> spam.setdefault('color', 'black')
'black'
```

```
>>> spam
{'color': 'black', 'age': 5, 'name': 'Pooka'}
```

```
>>> spam.setdefault('color', 'white')
'black'
```

```
>>> spam
{'color': 'black', 'age': 5, 'name': 'Pooka'}
```

Pretty Printing

```
>>> import pprint
>>>
>>> message = 'It was a bright cold day in April, and the clocks were striking
>>> thirteen.'
>>> count = {}
>>>
>>> for character in message:
>>>     count.setdefault(character, 0)
>>>     count[character] = count[character] + 1
>>>
>>> pprint.pprint(count)
{' ': 13,
 ',': 1,
 '.': 1,
 'A': 1,
 'I': 1,
 'a': 4,
 'b': 1,
 'c': 3,
 'd': 3,
 'e': 5,
 'g': 2,
 'h': 3,
 'i': 6,
 'k': 2,
 'l': 3,
 'n': 4,
 'o': 2,
 'p': 1,
 'r': 5,
 's': 3,
 't': 6,
 'w': 2,
 'y': 1}
```

Merge two dictionaries

```
# in Python 3.5+:
>>> x = {'a': 1, 'b': 2}
>>> y = {'b': 3, 'c': 4}
>>> z = {**x, **y}
>>> z
{'c': 4, 'a': 1, 'b': 3}

# in Python 2.7
>>> z = dict(x, **y)
>>> z
{'c': 4, 'a': 1, 'b': 3}
```

sets

From the Python 3 documentation

> A set is an unordered collection with no duplicate elements. Basic uses include membership testing and eliminating duplicate entries. Set objects also support mathematical operations like union, intersection, difference, and symmetric difference.

Initializing a set

There are two ways to create sets: using curly braces `{}` and the bult-in function `set()`

```
>>> s = {1, 2, 3}
>>> s = set([1, 2, 3])
```

When creating an empty set, be sure to not use the curly braces `{}` or you will get an empty dictionary instead.

```
>>> s = {}
>>> type(s)
<class 'dict'>
```

sets: unordered collections of unique elements

A set automatically remove all the duplicate values.

```
>>> s = {1, 2, 3, 2, 3, 4}
>>> s
{1, 2, 3, 4}
```

And as an unordered data type, they can't be indexed.

```
>>> s = {1, 2, 3}
>>> s[0]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'set' object does not support indexing
>>>
```

set add() and update()

Using the `add()` method we can add a single element to the set.

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

And with `update()`, multiple ones .

```
>>> s = {1, 2, 3}
>>> s.update([2, 3, 4, 5, 6])
>>> s
{1, 2, 3, 4, 5, 6}  # remember, sets automatically remove duplicates
```

set remove() and discard()

Both methods will remove an element from the set, but `remove()` will raise a `key error` if the value doesn't exist.

```
>>> s = {1, 2, 3}
>>> s.remove(3)
>>> s
{1, 2}
>>> s.remove(3)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
KeyError: 3
```

`discard()` won't raise any errors.

```
>>> s = {1, 2, 3}
>>> s.discard(3)
>>> s
{1, 2}
>>> s.discard(3)
>>>
```

set union()

`union()` or `|` will create a new set that contains all the elements from the sets provided.

```
>>> s1 = {1, 2, 3}
>>> s2 = {3, 4, 5}
>>> s1.union(s2)  # or 's1 | s2'
{1, 2, 3, 4, 5}
```

set intersection

`intersection` or `&` will return a set containing only the elements that are common to all of them.

```
>>> s1 = {1, 2, 3}
>>> s2 = {2, 3, 4}
>>> s3 = {3, 4, 5}
>>> s1.intersection(s2, s3)  # or 's1 & s2 & s3'
{3}
```

set difference

`difference` or `-` will return only the elements that are unique to the first set (invoked set).

```
>>> s1 = {1, 2, 3}
>>> s2 = {2, 3, 4}
>>> s1.difference(s2)  # or 's1 - s2'
{1}
>>> s2.difference(s1) # or 's2 - s1'
{4}
```

set symetric_difference

`symetric_difference` or `^` will return all the elements that are not common between them.

```
>>> s1 = {1, 2, 3}
>>> s2 = {2, 3, 4}
>>> s1.symmetric_difference(s2)  # or 's1 ^ s2'
{1, 4}
```

itertools Module

The *itertools* module is a colection of tools intented to be fast and use memory efficiently when handling iterators (like lists or dictionaries).

From the official Python 3.x documentation:

> The module standardizes a core set of fast, memory efficient tools that are useful by themselves or in combination. Together, they form an “iterator algebra” making it possible to construct specialized tools succinctly and efficiently in pure Python.

The *itertools* module comes in the standard library and must be imported.

The operator module will also be used. This module is not necessary when using itertools, but needed for some of the examples below.

accumulate()

Makes an iterator that returns the results of a function.

```
itertools.accumulate(iterable[, func])
```

Example:

```
>>> data = [1, 2, 3, 4, 5]
>>> result = itertools.accumulate(data, operator.mul)
>>> for each in result:
>>>    print(each)
1
2
6
24
120
```

The operator.mul takes two numbers and multiplies them:

```
operator.mul(1, 2)
2
operator.mul(2, 3)
6
operator.mul(6, 4)
24
operator.mul(24, 5)
120
```

Passing a function is optional:

```
>>> data = [5, 2, 6, 4, 5, 9, 1]
>>> result = itertools.accumulate(data)
>>> for each in result:
>>>    print(each)
5
7
13
17
22
31
32
```

If no function is designated the items will be summed:

```
5
5 + 2 = 7
7 + 6 = 13
13 + 4 = 17
17 + 5 = 22
22 + 9 = 31
31 + 1 = 32
```

combinations()

Takes an iterable and a integer. This will create all the unique combination that have r members.

```
itertools.combinations(iterable, r)
```

Example:

```
>>> shapes = ['circle', 'triangle', 'square',]
>>> result = itertools.combinations(shapes, 2)
>>> for each in result:
>>>    print(each)
('circle', 'triangle')
('circle', 'square')
('triangle', 'square')
```

combinations_with_replacement()

Just like combinations(), but allows individual elements to be repeated more than once.

```
itertools.combinations_with_replacement(iterable, r)
```

Example:

```
>>> shapes = ['circle', 'triangle', 'square']
>>> result = itertools.combinations_with_replacement(shapes, 2)
>>> for each in result:
>>>    print(each)
('circle', 'circle')
('circle', 'triangle')
('circle', 'square')
('triangle', 'triangle')
('triangle', 'square')
('square', 'square')
```

count()

Makes an iterator that returns evenly spaced values starting with number start.

```
itertools.count(start=0, step=1)
```

Example:

```
>>> for i in itertools.count(10,3):
>>>    print(i)
>>>    if i > 20:
>>>        break
10
13
16
19
22
```

cycle()

This function cycles through an iterator endlessly.

```
itertools.cycle(iterable)
```

Example:

```
>>> colors = ['red', 'orange', 'yellow', 'green', 'blue', 'violet']
>>> for color in itertools.cycle(colors):
>>>    print(color)
red
orange
yellow
green
blue
violet
red
orange
```

When reached the end of the iterable it start over again from the beginning.

chain()

Take a series of iterables and return them as one long iterable.

```
itertools.chain(*iterables)
```

Example:

```
>>> colors = ['red', 'orange', 'yellow', 'green', 'blue']
>>> shapes = ['circle', 'triangle', 'square', 'pentagon']
>>> result = itertools.chain(colors, shapes)
>>> for each in result:
>>>    print(each)
red
orange
yellow
green
blue
circle
triangle
square
pentagon
```

compress()

Filters one iterable with another.

```
itertools.compress(data, selectors)
```

Example:

```
>>> shapes = ['circle', 'triangle', 'square', 'pentagon']
>>> selections = [True, False, True, False]
>>> result = itertools.compress(shapes, selections)
>>> for each in result:
>>>    print(each)
circle
square
```

dropwhile()

Make an iterator that drops elements from the iterable as long as the predicate is true; afterwards, returns every element.

```
itertools.dropwhile(predicate, iterable)
```

Example:

```
>>> data = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 1]
>>> result = itertools.dropwhile(lambda x: x<5, data)
>>> for each in result:
>>>    print(each)
5
6
7
8
9
10
1
```

filterfalse()

Makes an iterator that filters elements from iterable returning only those for which the predicate is False.

```
itertools.filterfalse(predicate, iterable)
```

Example:

```
>>> data = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 1]
>>> result = itertools.filterfalse(lambda x: x<5, data)
>>> for each in result:
>>>    print(each)
5
6
7
8
9
10
```

groupby()

Simply put, this function groups things together.

```
itertools.groupby(iterable, key=None)
```

Example:

```
>>> robots = [{
    'name': 'blaster',
    'faction': 'autobot'
}, {
    'name': 'galvatron',
    'faction': 'decepticon'
}, {
    'name': 'jazz',
    'faction': 'autobot'
}, {
    'name': 'metroplex',
    'faction': 'autobot'
}, {
    'name': 'megatron',
    'faction': 'decepticon'
}, {
    'name': 'starcream',
    'faction': 'decepticon'
}]
>>> for key, group in itertools.groupby(robots, key=lambda x: x['faction']):
>>>    print(key)
>>>    print(list(group))
autobot
[{'name': 'blaster', 'faction': 'autobot'}]
decepticon
[{'name': 'galvatron', 'faction': 'decepticon'}]
autobot
[{'name': 'jazz', 'faction': 'autobot'}, {'name': 'metroplex', 'faction': 'autobot'}]
decepticon
[{'name': 'megatron', 'faction': 'decepticon'}, {'name': 'starcream', 'faction': 'decepticon'}]
```

islice()

This function is very much like slices. This allows you to cut out a piece of an iterable.

```
itertools.islice(iterable, start, stop[, step])
```

Example:

```
>>> colors = ['red', 'orange', 'yellow', 'green', 'blue',]
>>> few_colors = itertools.islice(colors, 2)
>>> for each in few_colors:
>>>    print(each)
red
orange
```

permutations()

```
itertools.permutations(iterable, r=None)
```

Example:

```
>>> alpha_data = ['a', 'b', 'c']
>>> result = itertools.permutations(alpha_data)
>>> for each in result:
>>>    print(each)
('a', 'b', 'c')
('a', 'c', 'b')
('b', 'a', 'c')
('b', 'c', 'a')
('c', 'a', 'b')
('c', 'b', 'a')
```

product()

Creates the cartesian products from a series of iterables.

```
>>> num_data = [1, 2, 3]
>>> alpha_data = ['a', 'b', 'c']
>>> result = itertools.product(num_data, alpha_data)
>>> for each in result:
    print(each)
(1, 'a')
(1, 'b')
(1, 'c')
(2, 'a')
(2, 'b')
(2, 'c')
(3, 'a')
(3, 'b')
(3, 'c')
```

repeat()

This function will repeat an object over and over again. Unless, there is a times argument.

```
itertools.repeat(object[, times])
```

Example:

```
>>> for i in itertools.repeat("spam", 3):
    print(i)
spam
spam
spam
```

starmap()

Makes an iterator that computes the function using arguments obtained from the iterable.

```
itertools.starmap(function, iterable)
```

Example:

```
>>> data = [(2, 6), (8, 4), (7, 3)]
>>> result = itertools.starmap(operator.mul, data)
>>> for each in result:
>>>    print(each)
12
32
21
```

takewhile()

The opposite of dropwhile(). Makes an iterator and returns elements from the iterable as long as the predicate is true.

```
itertools.takewhile(predicate, iterable)
```

Example:

```
>>> data = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 1]
>>> result = itertools.takewhile(lambda x: x<5, data)
>>> for each in result:
>>>    print(each)
1
2
3
4
```

tee()

Return n independent iterators from a single iterable.

```
itertools.tee(iterable, n=2)
```

Example:

```
>>> colors = ['red', 'orange', 'yellow', 'green', 'blue']
>>> alpha_colors, beta_colors = itertools.tee(colors)
>>> for each in alpha_colors:
>>>    print(each)
red
orange
yellow
green
blue
```

```
>>> colors = ['red', 'orange', 'yellow', 'green', 'blue']
>>> alpha_colors, beta_colors = itertools.tee(colors)
>>> for each in beta_colors:
>>>    print(each)
red
orange
yellow
green
blue
```

zip_longest()

Makes an iterator that aggregates elements from each of the iterables. If the iterables are of uneven length, missing values are filled-in with fillvalue. Iteration continues until the longest iterable is exhausted.

```
itertools.zip_longest(*iterables, fillvalue=None)
```

Example:

```
>>> colors = ['red', 'orange', 'yellow', 'green', 'blue',]
>>> data = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10,]
>>> for each in itertools.zip_longest(colors, data, fillvalue=None):
>>>    print(each)
('red', 1)
('orange', 2)
('yellow', 3)
('green', 4)
('blue', 5)
(None, 6)
(None, 7)
(None, 8)
(None, 9)
(None, 10)
```

Comprehensions

List comprehension

```
>>> a = [1, 3, 5, 7, 9, 11]

>>> [i - 1 for i in a]
[0, 2, 4, 6, 8, 10]
```

Set comprehension

```
>>> b = {"abc", "def"}
>>> {s.upper() for s in b}
{"ABC", "DEF"}
```

Dict comprehension

```
>>> c = {'name': 'Pooka', 'age': 5}
>>> {v: k for k, v in c.items()}
{'Pooka': 'name', 5: 'age'}
```

A List comprehension can be generated from a dictionary:

```
>>> c = {'name': 'Pooka', 'first_name': 'Oooka'}
>>> ["{}:{}".format(k.upper(), v.upper()) for k, v in c.items()]
['NAME:POOKA', 'FIRST_NAME:OOOKA']
```

Manipulating Strings

Escape Characters

Escape character	Prints as
`\'`	Single quote
`\"`	Double quote
`\t`	Tab
`\n`	Newline (line break)

Escape character	Prints as
`\\`	Backslash

Example:

```
>>> print("Hello there!\nHow are you?\nI\'m doing fine.")
Hello there!
How are you?
I'm doing fine.
```

Raw Strings

A raw string completely ignores all escape characters and prints any backslash that appears in the string.

```
>>> print(r'That is Carol\'s cat.')
That is Carol\'s cat.
```

Note: mostly used for regular expression definition (see `re` package)

Multiline Strings with Triple Quotes

```
>>> print('''Dear Alice,
>>>
>>> Eve's cat has been arrested for catnapping, cat burglary, and extortion.
>>>
>>> Sincerely,
>>> Bob''')
Dear Alice,

Eve's cat has been arrested for catnapping, cat burglary, and extortion.

Sincerely,
Bob
```

To keep a nicer flow in your code, you can use the `dedent` function from the `textwrap` standard package.

```
>>> from textwrap import dedent
>>>
>>> def my_function():
>>>     print('''
>>>         Dear Alice,
>>>
>>>         Eve's cat has been arrested for catnapping, cat burglary, and extortion.
>>>
>>>         Sincerely,
>>>         Bob
>>>         ''').strip()
```

This generates the same string than before.

Indexing and Slicing Strings

```
H   e   l   l   o       w   o   r   l   d    !
0   1   2   3   4   5   6   7   8   9   10   11
```

```
>>> spam = 'Hello world!'

>>> spam[0]
'H'
```

```
>>> spam[4]
'o'
```

```
>>> spam[-1]
'!'
```

Slicing:

```
>>> spam[0:5]
'Hello'
```

```
>>> spam[:5]
'Hello'
```

```
>>> spam[6:]
'world!'
```

```
>>> spam[6:-1]
'world'
```

```
>>> spam[:-1]
'Hello world'
```

```
>>> spam[::-1]
'!dlrow olleH'
```

```
>>> spam = 'Hello world!'
>>> fizz = spam[0:5]
>>> fizz
'Hello'
```

The in and not in Operators with Strings

```
>>> 'Hello' in 'Hello World'
True
```

```
>>> 'Hello' in 'Hello'
True
```

```
>>> 'HELLO' in 'Hello World'
False
```

```
>>> '' in 'spam'
True
```

```
>>> 'cats' not in 'cats and dogs'
False
```

The in and not in Operators with list

```
>>> a = [1, 2, 3, 4]
>>> 5 in a
False
```

```
>>> 2 in a
True
```

The upper(), lower(), isupper(), and islower() String Methods

`upper()` and `lower()`:

```
>>> spam = 'Hello world!'
>>> spam = spam.upper()
>>> spam
'HELLO WORLD!'
```

```
>>> spam = spam.lower()
>>> spam
'hello world!'
```

isupper() and islower():

```
>>> spam = 'Hello world!'
>>> spam.islower()
False
```

```
>>> spam.isupper()
False
```

```
>>> 'HELLO'.isupper()
True
```

```
>>> 'abc12345'.islower()
True
```

```
>>> '12345'.islower()
False
```

```
>>> '12345'.isupper()
False
```

The isX String Methods

- **isalpha()** returns True if the string consists only of letters and is not blank.
- **isalnum()** returns True if the string consists only of letters and numbers and is not blank.
- **isdecimal()** returns True if the string consists only of numeric characters and is not blank.
- **isspace()** returns True if the string consists only of spaces,tabs, and new-lines and is not blank.
- **istitle()** returns True if the string consists only of words that begin with an uppercase letter followed by only lowercase letters.

The startswith() and endswith() String Methods

```
>>> 'Hello world!'.startswith('Hello')
True
```

```
>>> 'Hello world!'.endswith('world!')
True
```

```
>>> 'abc123'.startswith('abcdef')
False
```

```
>>> 'abc123'.endswith('12')
False
```

```
>>> 'Hello world!'.startswith('Hello world!')
True
```

```
>>> 'Hello world!'.endswith('Hello world!')
True
```

The join() and split() String Methods

join():

```
>>> ', '.join(['cats', 'rats', 'bats'])
'cats, rats, bats'
```

```
>>> ' '.join(['My', 'name', 'is', 'Simon'])
'My name is Simon'
```

```
>>> 'ABC'.join(['My', 'name', 'is', 'Simon'])
'MyABCnameABCisABCSimon'
```

split():

```
>>> 'My name is Simon'.split()
['My', 'name', 'is', 'Simon']
```

```
>>> 'MyABCnameABCisABCSimon'.split('ABC')
['My', 'name', 'is', 'Simon']
```

```
>>> 'My name is Simon'.split('m')
['My na', 'e is Si', 'on']
```

Justifying Text with rjust(), ljust(), and center()

rjust() and ljust():

```
>>> 'Hello'.rjust(10)
'     Hello'
```

```
>>> 'Hello'.rjust(20)
'              Hello'
```

```
>>> 'Hello World'.rjust(20)
'         Hello World'
```

```
>>> 'Hello'.ljust(10)
'Hello     '
```

An optional second argument to rjust() and ljust() will specify a fill character other than a space character. Enter the following into the interactive shell:

```
>>> 'Hello'.rjust(20, '*')
'***************Hello'
```

```
>>> 'Hello'.ljust(20, '-')
'Hello---------------'
```

center():

```
>>> 'Hello'.center(20)
'       Hello       '
```

```
>>> 'Hello'.center(20, '=')
'=======Hello========'
```

Removing Whitespace with strip(), rstrip(), and lstrip()

```
>>> spam = '    Hello World     '
>>> spam.strip()
'Hello World'
```

```
>>> spam.lstrip()
'Hello World '
```

```
>>> spam.rstrip()
'    Hello World'
```

```
>>> spam = 'SpamSpamBaconSpamEggsSpamSpam'
>>> spam.strip('ampS')
'BaconSpamEggs'
```

Copying and Pasting Strings with the pyperclip Module (need pip install)

```
>>> import pyperclip

>>> pyperclip.copy('Hello world!')

>>> pyperclip.paste()
'Hello world!'
```

String Formatting

% operator

```
>>> name = 'Pete'
>>> 'Hello %s' % name
"Hello Pete"
```

We can use the `%x` format specifier to convert an int value to a string:

```
>>> num = 5
>>> 'I have %x apples' % num
"I have 5 apples"
```

Note: For new code, using str.format or f-strings (Python 3.6+) is strongly recommended over the `%` operator.

String Formatting (str.format)

Python 3 introduced a new way to do string formatting that was later back-ported to Python 2.7. This makes the syntax for string formatting more regular.

```
>>> name = 'John'
>>> age = 20'

>>> "Hello I'm {}, my age is {}".format(name, age)
"Hello I'm John, my age is 20"
```

```
>>> "Hello I'm {0}, my age is {1}".format(name, age)
"Hello I'm John, my age is 20"
```

The official Python 3.x documentation recommend `str.format` over the `%` operator:

> The formatting operations described here exhibit a variety of quirks that lead to a number of common errors (such as failing to display tuples and dictionaries correctly). Using the newer formatted string literals or the str.format() interface helps avoid these errors. These alternatives also provide more powerful, flexible and extensible approaches to formatting text.

Lazy string formatting

You would only use `%s` string formatting on functions that can do lazy parameters evaluation, the most common being logging:

Prefer:

```
>>> name = "alice"
>>> logging.debug("User name: %s", name)
```

Over:

```
>>> logging.debug("User name: {}".format(name))
```

Or:

```
>>> logging.debug("User name: " + name)
```

Formatted String Literals or f-strings (Python 3.6+)

```
>>> name = 'Elizabeth'
>>> f'Hello {name}!'
'Hello Elizabeth!
```

It is even possible to do inline arithmetic with it:

```
>>> a = 5
>>> b = 10
>>> f'Five plus ten is {a + b} and not {2 * (a + b)}.'
'Five plus ten is 15 and not 30.'
```

Template Strings

A simpler and less powerful mechanism, but it is recommended when handling format strings generated by users. Due to their reduced complexity template strings are a safer choice.

```
>>> from string import Template
>>> name = 'Elizabeth'
>>> t = Template('Hey $name!')
>>> t.substitute(name=name)
'Hey Elizabeth!'
```

Regular Expressions

1. Import the regex module with `import re`.
2. Create a Regex object with the `re.compile()` function. (Remember to use a raw string.)
3. Pass the string you want to search into the Regex object's `search()` method. This returns a `Match` object.
4. Call the Match object's `group()` method to return a string of the actual matched text.

All the regex functions in Python are in the re module:

```
>>> import re
```

Matching Regex Objects

```
>>> phone_num_regex = re.compile(r'\d\d\d-\d\d\d-\d\d\d\d')

>>> mo = phone_num_regex.search('My number is 415-555-4242.')

>>> print('Phone number found: {}'.format(mo.group()))
Phone number found: 415-555-4242
```

Grouping with Parentheses

```
>>> phone_num_regex = re.compile(r'(\d\d\d)-(\d\d\d-\d\d\d\d)')

>>> mo = phone_num_regex.search('My number is 415-555-4242.')

>>> mo.group(1)
'415'

>>> mo.group(2)
'555-4242'

>>> mo.group(0)
'415-555-4242'

>>> mo.group()
'415-555-4242'
```

To retrieve all the groups at once: use the groups() method—note the plural form for the name.

```
>>> mo.groups()
('415', '555-4242')

>>> area_code, main_number = mo.groups()

>>> print(area_code)
415

>>> print(main_number)
555-4242
```

Matching Multiple Groups with the Pipe

The | character is called a pipe. You can use it anywhere you want to match one of many expressions. For example, the regular expression r'Batman|Tina Fey' will match either 'Batman' or 'Tina Fey'.

```
>>> hero_regex = re.compile (r'Batman|Tina Fey')

>>> mo1 = hero_regex.search('Batman and Tina Fey.')

>>> mo1.group()
'Batman'

>>> mo2 = hero_regex.search('Tina Fey and Batman.')

>>> mo2.group()
'Tina Fey'
```

You can also use the pipe to match one of several patterns as part of your regex:

```
>>> bat_regex = re.compile(r'Bat(man|mobile|copter|bat)')

>>> mo = bat_regex.search('Batmobile lost a wheel')

>>> mo.group()
'Batmobile'

>>> mo.group(1)
'mobile'
```

Optional Matching with the Question Mark

The ? character flags the group that precedes it as an optional part of the pattern.

```
>>> bat_regex = re.compile(r'Bat(wo)?man')
>>> mo1 = bat_regex.search('The Adventures of Batman')
>>> mo1.group()
'Batman'

>>> mo2 = bat_regex.search('The Adventures of Batwoman')
>>> mo2.group()
'Batwoman'
```

Matching Zero or More with the Star

The * (called the star or asterisk) means “match zero or more”—the group that precedes the star can occur any number of times in the text.

```
>>> bat_regex = re.compile(r'Bat(wo)*man')
>>> mo1 = bat_regex.search('The Adventures of Batman')
>>> mo1.group()
'Batman'

>>> mo2 = bat_regex.search('The Adventures of Batwoman')
>>> mo2.group()
'Batwoman'

>>> mo3 = bat_regex.search('The Adventures of Batwowowowoman')
>>> mo3.group()
'Batwowowowoman'
```

Matching One or More with the Plus

While * means “match zero or more,” the + (or plus) means “match one or more”. The group preceding a plus must appear at least once. It is not optional:

```
>>> bat_regex = re.compile(r'Bat(wo)+man')
>>> mo1 = bat_regex.search('The Adventures of Batwoman')
>>> mo1.group()
'Batwoman'
```

```
>>> mo2 = bat_regex.search('The Adventures of Batwowowowoman')
>>> mo2.group()
'Batwowowowoman'
```

```
>>> mo3 = bat_regex.search('The Adventures of Batman')
>>> mo3 is None
True
```

Matching Specific Repetitions with Curly Brackets

If you have a group that you want to repeat a specific number of times, follow the group in your regex with a number in curly brackets. For example, the regex (Ha){3} will match the string 'HaHaHa', but it will not match 'HaHa', since the latter has only two repeats of the (Ha) group.

Instead of one number, you can specify a range by writing a minimum, a comma, and a maximum in between the curly brackets. For example, the regex (Ha){3,5} will match 'HaHaHa', 'HaHaHaHa', and 'HaHaHaHaHa'.

```
>>> ha_regex = re.compile(r'(Ha){3}')
>>> mo1 = ha_regex.search('HaHaHa')
>>> mo1.group()
'HaHaHa'
```

```
>>> mo2 = ha_regex.search('Ha')
>>> mo2 is None
True
```

Greedy and Nongreedy Matching

Python's regular expressions are greedy by default, which means that in ambiguous situations they will match the longest string possible. The non-greedy version of the curly brackets, which matches the shortest string possible, has the closing curly bracket followed by a question mark.

```
>>> greedy_ha_regex = re.compile(r'(Ha){3,5}')
>>> mo1 = greedy_ha_regex.search('HaHaHaHaHa')
>>> mo1.group()
'HaHaHaHaHa'
```

```
>>> nongreedy_ha_regex = re.compile(r'(Ha){3,5}?')
>>> mo2 = nongreedy_ha_regex.search('HaHaHaHaHa')
>>> mo2.group()
'HaHaHa'
```

The findall() Method

In addition to the search() method, Regex objects also have a findall() method. While search() will return a Match object of the first matched text in the searched string, the findall() method will return the strings of every match in the searched string.

```
>>> phone_num_regex = re.compile(r'\d\d\d-\d\d\d-\d\d\d\d') # has no groups

>>> phone_num_regex.findall('Cell: 415-555-9999 Work: 212-555-0000')
['415-555-9999', '212-555-0000']
```

To summarize what the findall() method returns, remember the following:

- When called on a regex with no groups, such as \d-\d\d\d-\d\d\d\d, the method findall() returns a list of ng matches, such as ['415-555-9999', '212-555-0000'].
- When called on a regex that has groups, such as (\d\d\d)-(d\d)-(\d\d\d\d), the method findall() returns a list of es of strings (one string for each group), such as [('415', '555', '9999'), ('212', '555', '0000')].

Making Your Own Character Classes

There are times when you want to match a set of characters but the shorthand character classes (\d, \w, \s, and so on) are too broad. You can define your own character class using square brackets. For example, the character class [aeiouAEIOU] will match any vowel, both lowercase and uppercase.

```
>>> vowel_regex = re.compile(r'[aeiouAEIOU]')

>>> vowel_regex.findall('Robocop eats baby food. BABY FOOD.')
['o', 'o', 'o', 'e', 'a', 'a', 'o', 'o', 'A', 'O', 'O']
```

You can also include ranges of letters or numbers by using a hyphen. For example, the character class [a-zA-Z0-9] will match all lowercase letters, uppercase letters, and numbers.

By placing a caret character (^) just after the character class's opening bracket, you can make a negative character class. A negative character class will match all the characters that are not in the character class. For example, enter the following into the interactive shell:

```
>>> consonant_regex = re.compile(r'[^aeiouAEIOU]')

>>> consonant_regex.findall('Robocop eats baby food. BABY FOOD.')
['R', 'b', 'c', 'p', ' ', 't', 's', ' ', 'b', 'b', 'y', ' ', 'f', 'd', '.', '
', 'B', 'B', 'Y', ' ', 'F', 'D', '.']
```

The Caret and Dollar Sign Characters

- You can also use the caret symbol (^) at the start of a regex to indicate that a match must occur at the beginning of the searched text.
- Likewise, you can put a dollar sign (\$) at the end of the regex to indicate the string must end with this regex pattern.
- And you can use the ^ and \$ together to indicate that the entire string must match the regex—that is, it's not enough for a match to be made on some subset of the string.

The r'^Hello' regular expression string matches strings that begin with 'Hello':

```
>>> begins_with_hello = re.compile(r'^Hello')

>>> begins_with_hello.search('Hello world!')
<_sre.SRE_Match object; span=(0, 5), match='Hello'>

>>> begins_with_hello.search('He said hello.') is None
True
```

The r'\d\$' regular expression string matches strings that end with a numeric character from 0 to 9:

```
>>> whole_string_is_num = re.compile(r'^\d+$')

>>> whole_string_is_num.search('1234567890')
<_sre.SRE_Match object; span=(0, 10), match='1234567890'>

>>> whole_string_is_num.search('12345xyz67890') is None
True

>>> whole_string_is_num.search('12 34567890') is None
True
```

The Wildcard Character

The . (or dot) character in a regular expression is called a wildcard and will match any character except for a newline:

```
>>> at_regex = re.compile(r'.at')

>>> at_regex.findall('The cat in the hat sat on the flat mat.')
['cat', 'hat', 'sat', 'lat', 'mat']
```

Matching Everything with Dot-Star

```
>>> name_regex = re.compile(r'First Name: (.*) Last Name: (.*)')

>>> mo = name_regex.search('First Name: Al Last Name: Sweigart')

>>> mo.group(1)
'Al'
```

```
>>> mo.group(2)
'Sweigart'
```

The dot-star uses greedy mode: It will always try to match as much text as possible. To match any and all text in a nongreedy fashion, use the dot, star, and question mark (.*?). The question mark tells Python to match in a nongreedy way:

```
>>> nongreedy_regex = re.compile(r'<.*?>')
>>> mo = nongreedy_regex.search('<To serve man> for dinner.>')
>>> mo.group()
'<To serve man>'
```

```
>>> greedy_regex = re.compile(r'<.*>')
>>> mo = greedy_regex.search('<To serve man> for dinner.>')
>>> mo.group()
'<To serve man> for dinner.>'
```

Matching Newlines with the Dot Character

The dot-star will match everything except a newline. By passing re.DOTALL as the second argument to re.compile(), you can make the dot character match all characters, including the newline character:

```
>>> no_newline_regex = re.compile('.*')
>>> no_newline_regex.search('Serve the public trust.\nProtect the innocent.\nUphold the law.').group()
'Serve the public trust.'
```

```
>>> newline_regex = re.compile('.*', re.DOTALL)
>>> newline_regex.search('Serve the public trust.\nProtect the innocent.\nUphold the law.').group()
'Serve the public trust.\nProtect the innocent.\nUphold the law.'
```

Review of Regex Symbols

Symbol	Matches
`?`	zero or one of the preceding group.
`*`	zero or more of the preceding group.
`+`	one or more of the preceding group.
`{n}`	exactly n of the preceding group.
`{n,}`	n or more of the preceding group.
`{,m}`	0 to m of the preceding group.
`{n,m}`	at least n and at most m of the preceding p.
`{n,m}?` or `*?` or `+?`	performs a nongreedy match of the preceding p.
`^spam`	means the string must begin with spam.
`spam$`	means the string must end with spam.
`.`	any character, except newline characters.
`\d`, `\w`, and `\s`	a digit, word, or space character, respectively.
`\D`, `\W`, and `\S`	anything except a digit, word, or space acter, respectively.
`[abc]`	any character between the brackets (such as a, b,).
`[^abc]`	any character that isn't between the brackets.

Case-Insensitive Matching

To make your regex case-insensitive, you can pass re.IGNORECASE or re.I as a second argument to re.compile():

```
>>> robocop = re.compile(r'robocop', re.I)

>>> robocop.search('Robocop is part man, part machine, all cop.').group()
'Robocop'
```

```
>>> robocop.search('ROBOCOP protects the innocent.').group()
'ROBOCOP'
```

```
>>> robocop.search('Al, why does your programming book talk about robocop so much?').group()
'robocop'
```

Substituting Strings with the sub() Method

The sub() method for Regex objects is passed two arguments:

1. The first argument is a string to replace any matches.
2. The second is the string for the regular expression.

The sub() method returns a string with the substitutions applied:

```
>>> names_regex = re.compile(r'Agent \w+')

>>> names_regex.sub('CENSORED', 'Agent Alice gave the secret documents to Agent Bob.')
'CENSORED gave the secret documents to CENSORED.'
```

Another example:

```
>>> agent_names_regex = re.compile(r'Agent (\w)\w*')

>>> agent_names_regex.sub(r'\1****', 'Agent Alice told Agent Carol that Agent Eve knew Agent Bob was a double agent.')
A**** told C**** that E**** knew B**** was a double agent.'
```

Managing Complex Regexes

To tell the re.compile() function to ignore whitespace and comments inside the regular expression string, “verbose mode” can be enabled by passing the variable re.VERBOSE as the second argument to re.compile().

Now instead of a hard-to-read regular expression like this:

```
phone_regex = re.compile(r'((\d{3}|\(\d{3}\))?(\s|-|\.)?\d{3}(\s|-|\.)\d{4}(\s*(ext|x|ext.)\s*\d{2,5})?)')
```

you can spread the regular expression over multiple lines with comments like this:

```
phone_regex = re.compile(r'''(
    (\d{3}|\(\d{3}\))?            # area code
    (\s|-|\.)?                    # separator
    \d{3}                         # first 3 digits
    (\s|-|\.)                     # separator
    \d{4}                         # last 4 digits
    (\s*(ext|x|ext.)\s*\d{2,5})?  # extension
    )''', re.VERBOSE)
```

Handling File and Directory Paths

There are two main modules in Python that deals with path manipulation. One is the `os.path` module and the other is the `pathlib` module. The `pathlib` module was added in Python 3.4, offering an object-oriented way to handle file system paths.

Backslash on Windows and Forward Slash on OS X and Linux

On Windows, paths are written using backslashes (`\`) as the separator between folder names. On Unix based operating system such as macOS, Linux, and BSDs, the forward slash (`/`) is used as the path separator. Joining paths can be a headache if your code needs to work on different platforms.

Fortunately, Python provides easy ways to handle this. We will showcase how to deal with this with both `os.path.join` and `pathlib.Path.joinpath`

Using `os.path.join` on Windows:

```
>>> import os

>>> os.path.join('usr', 'bin', 'spam')
'usr\\bin\\spam'
```

And using `pathlib` on *nix:

```
>>> from pathlib import Path

>>> print(Path('usr').joinpath('bin').joinpath('spam'))
usr/bin/spam
```

`pathlib` also provides a shortcut to joinpath using the `/` operator:

```
>>> from pathlib import Path

>>> print(Path('usr') / 'bin' / 'spam')
usr/bin/spam
```

Notice the path separator is different between Windows and Unix based operating system, that's why you want to use one of the above methods instead of adding strings together to join paths together.

Joining paths is helpful if you need to create different file paths under the same directory.

Using `os.path.join` on Windows:

```
>>> my_files = ['accounts.txt', 'details.csv', 'invite.docx']

>>> for filename in my_files:
>>>     print(os.path.join('C:\\Users\\asweigart', filename))
C:\Users\asweigart\accounts.txt
C:\Users\asweigart\details.csv
C:\Users\asweigart\invite.docx
```

Using `pathlib` on *nix:

```
>>> my_files = ['accounts.txt', 'details.csv', 'invite.docx']
>>> home = Path.home()
>>> for filename in my_files:
>>>     print(home / filename)
/home/asweigart/accounts.txt
/home/asweigart/details.csv
/home/asweigart/invite.docx
```

The Current Working Directory

Using `os` on Windows:

```
>>> import os

>>> os.getcwd()
'C:\\Python34'
>>> os.chdir('C:\\Windows\\System32')

>>> os.getcwd()
'C:\\Windows\\System32'
```

Using `pathlib` on *nix:

```
>>> from pathlib import Path
>>> from os import chdir

>>> print(Path.cwd())
/home/asweigart

>>> chdir('/usr/lib/python3.6')
>>> print(Path.cwd())
/usr/lib/python3.6
```

Creating New Folders

Using `os` on Windows:

```
>>> import os
>>> os.makedirs('C:\\delicious\\walnut\\waffles')
```

Using `pathlib` on *nix:

```
>>> from pathlib import Path
>>> cwd = Path.cwd()
>>> (cwd / 'delicious' / 'walnut' / 'waffles').mkdir()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "/usr/lib/python3.6/pathlib.py", line 1226, in mkdir
    self._accessor.mkdir(self, mode)
  File "/usr/lib/python3.6/pathlib.py", line 387, in wrapped
    return strfunc(str(pathobj), *args)
FileNotFoundError: [Errno 2] No such file or directory: '/home/asweigart/delicious/walnut/waffles'
```

Oh no, we got a nasty error! The reason is that the 'delicious' directory does not exist, so we cannot make the 'walnut' and the 'waffles' directories under it. To fix this, do:

```
>>> from pathlib import Path
>>> cwd = Path.cwd()
>>> (cwd / 'delicious' / 'walnut' / 'waffles').mkdir(parents=True)
```

And all is good :)

Absolute vs. Relative Paths

There are two ways to specify a file path.

- An absolute path, which always begins with the root folder
- A relative path, which is relative to the program's current working directory

There are also the dot (.) and dot-dot (..) folders. These are not real folders but special names that can be used in a path. A single period ("dot") for a folder name is shorthand for "this directory." Two periods ("dot-dot") means "the parent folder."

Handling Absolute and Relative Paths

To see if a path is an absolute path:

Using `os.path` on *nix:

```
>>> import os
>>> os.path.isabs('/')
True
>>> os.path.isabs('..')
False
```

Using `pathlib` on *nix:

```
>>> from pathlib import Path
>>> Path('/').is_absolute()
True
>>> Path('..').is_absolute()
False
```

You can extract an absolute path with both `os.path` and `pathlib`

Using `os.path` on *nix:

```
>>> import os
>>> os.getcwd()
'/home/asweigart'
>>> os.path.abspath('..')
'/home'
```

Using `pathlib` on *nix:

```
from pathlib import Path
print(Path.cwd())
/home/asweigart
print(Path('..').resolve())
/home
```

You can get a relative path from a starting path to another path.

Using `os.path` on *nix:

```
>>> import os
>>> os.path.relpath('/etc/passwd', '/')
'etc/passwd'
```

Using `pathlib` on *nix:

```
>>> from pathlib import Path
>>> print(Path('/etc/passwd').relative_to('/'))
etc/passwd
```

Checking Path Validity

Checking if a file/directory exists:

Using `os.path` on *nix:

```
import os
>>> os.path.exists('.')
True
>>> os.path.exists('setup.py')
True
>>> os.path.exists('/etc')
True
>>> os.path.exists('nonexistentfile')
False
```

Using `pathlib` on *nix:

```
from pathlib import Path
>>> Path('.').exists()
True
>>> Path('setup.py').exists()
True
>>> Path('/etc').exists()
True
>>> Path('nonexistentfile').exists()
False
```

Checking if a path is a file:

Using `os.path` on *nix:

```
>>> import os
>>> os.path.isfile('setup.py')
True
>>> os.path.isfile('/home')
False
>>> os.path.isfile('nonexistentfile')
False
```

Using `pathlib` on *nix:

```
>>> from pathlib import Path
>>> Path('setup.py').is_file()
True
>>> Path('/home').is_file()
False
>>> Path('nonexistentfile').is_file()
False
```

Checking if a path is a directory:

Using `os.path` on *nix:

```
>>> import os
>>> os.path.isdir('/')
True
>>> os.path.isdir('setup.py')
False
>>> os.path.isdir('/spam')
False
```

Using `pathlib` on *nix:

```
>>> from pathlib import Path
>>> Path('/').is_dir()
True
>>> Path('setup.py').is_dir()
False
>>> Path('/spam').is_dir()
False
```

Finding File Sizes and Folder Contents

Getting a file's size in bytes:

Using `os.path` on Windows:

```
>>> import os
>>> os.path.getsize('C:\\Windows\\System32\\calc.exe')
776192
```

Using `pathlib` on *nix:

```
>>> from pathlib import Path
>>> stat = Path('/bin/python3.6').stat()
>>> print(stat) # stat contains some other information about the file as well
os.stat_result(st_mode=33261, st_ino=141087, st_dev=2051, st_nlink=2, st_uid=0,
--snip--
st_gid=0, st_size=10024, st_atime=1517725562, st_mtime=1515119809, st_ctime=1517261276)
>>> print(stat.st_size) # size in bytes
10024
```

Listing directory contents using `os.listdir` on Windows:

```
>>> import os
>>> os.listdir('C:\\Windows\\System32')
['0409', '12520437.cpx', '12520850.cpx', '5U877.ax', 'aaclient.dll',
--snip--
'xwtpdui.dll', 'xwtpw32.dll', 'zh-CN', 'zh-HK', 'zh-TW', 'zipfldr.dll']
```

Listing directory contents using `pathlib` on *nix:

```
>>> from pathlib import Path
>>> for f in Path('/usr/bin').iterdir():
>>>     print(f)
...
/usr/bin/tiff2rgba
/usr/bin/iconv
/usr/bin/ldd
/usr/bin/cache_restore
/usr/bin/udiskie
/usr/bin/unix2dos
/usr/bin/t1reencode
/usr/bin/epstopdf
/usr/bin/idle3
...
```

To find the total size of all the files in this directory:

WARNING: Directories themselves also have a size! So you might want to check for whether a path is a file or directory using the methods in the methods discussed in the above section!

Using `os.path.getsize()` and `os.listdir()` together on Windows:

```
>>> import os
>>> total_size = 0

>>> for filename in os.listdir('C:\\Windows\\System32'):
      total_size = total_size + os.path.getsize(os.path.join('C:\\Windows\\System32', filename))

>>> print(total_size)
1117846456
```

Using `pathlib` on *nix:

```
>>> from pathlib import Path
>>> total_size = 0

>>> for sub_path in Path('/usr/bin').iterdir():
...     total_size += sub_path.stat().st_size
>>>
>>> print(total_size)
1903178911
```

Copying Files and Folders

The shutil module provides functions for copying files, as well as entire folders.

```
>>> import shutil, os

>>> os.chdir('C:\\')

>>> shutil.copy('C:\\spam.txt', 'C:\\delicious')
   'C:\\delicious\\spam.txt'

>>> shutil.copy('eggs.txt', 'C:\\delicious\\eggs2.txt')
   'C:\\delicious\\eggs2.txt'
```

While shutil.copy() will copy a single file, shutil.copytree() will copy an entire folder and every folder and file contained in it:

```
>>> import shutil, os

>>> os.chdir('C:\\')

>>> shutil.copytree('C:\\bacon', 'C:\\bacon_backup')
'C:\\bacon_backup'
```

Moving and Renaming Files and Folders

```
>>> import shutil
>>> shutil.move('C:\\bacon.txt', 'C:\\eggs')
'C:\\eggs\\bacon.txt'
```

The destination path can also specify a filename. In the following example, the source file is moved and renamed:

```
>>> shutil.move('C:\\bacon.txt', 'C:\\eggs\\new_bacon.txt')
'C:\\eggs\\new_bacon.txt'
```

If there is no eggs folder, then move() will rename bacon.txt to a file named eggs.

```
>>> shutil.move('C:\\bacon.txt', 'C:\\eggs')
'C:\\eggs'
```

Permanently Deleting Files and Folders

- Calling os.unlink(path) or Path.unlink() will delete the file at path.
- Calling os.rmdir(path) or Path.rmdir() will delete the folder at path. This folder must be empty of any files or folders.
- Calling shutil.rmtree(path) will remove the folder at path, and all files and folders it contains will also be deleted.

Safe Deletes with the send2trash Module

You can install this module by running pip install send2trash from a Terminal window.

```
>>> import send2trash

>>> with open('bacon.txt', 'a') as bacon_file: # creates the file
...     bacon_file.write('Bacon is not a vegetable.')
25

>>> send2trash.send2trash('bacon.txt')
```

Walking a Directory Tree

```
>>> import os
>>>
>>> for folder_name, subfolders, filenames in os.walk('C:\\delicious'):
>>>     print('The current folder is {}'.format(folder_name))
>>>
>>>     for subfolder in subfolders:
>>>         print('SUBFOLDER OF {}: {}'.format(folder_name, subfolder))
>>>     for filename in filenames:
>>>         print('FILE INSIDE {}: {}'.format(folder_name, filename))
>>>
>>>     print('')
The current folder is C:\delicious
SUBFOLDER OF C:\delicious: cats
SUBFOLDER OF C:\delicious: walnut
FILE INSIDE C:\delicious: spam.txt

The current folder is C:\delicious\cats
FILE INSIDE C:\delicious\cats: catnames.txt
FILE INSIDE C:\delicious\cats: zophie.jpg

The current folder is C:\delicious\walnut
SUBFOLDER OF C:\delicious\walnut: waffles

The current folder is C:\delicious\walnut\waffles
FILE INSIDE C:\delicious\walnut\waffles: butter.txt
```

`pathlib` provides a lot more functionality than the ones listed above, like getting file name, getting file extension, reading/writing a file without manually opening it, etc. Check out the official documentation if you want to know more!

Reading and Writing Files

The File Reading/Writing Process

To read/write to a file in Python, you will want to use the `with` statement, which will close the file for you after you are done.

Opening and reading files with the open() function

```
>>> with open('C:\\Users\\your_home_folder\\hello.txt') as hello_file:
...     hello_content = hello_file.read()
>>> hello_content
'Hello World!'

>>> # Alternatively, you can use the *readlines()* method to get a list of string values from the file, one string for each line of

>>> with open('sonnet29.txt') as sonnet_file:
...     sonnet_file.readlines()
[When, in disgrace with fortune and men's eyes,\n', ' I all alone beweep my
outcast state,\n', And trouble deaf heaven with my bootless cries,\n', And
look upon myself and curse my fate,']

>>> # You can also iterate through the file line by line:
>>> with open('sonnet29.txt') as sonnet_file:
...     for line in sonnet_file: # note the new line character will be included in the line
...         print(line, end='')

When, in disgrace with fortune and men's eyes,
I all alone beweep my outcast state,
And trouble deaf heaven with my bootless cries,
And look upon myself and curse my fate,
```

Writing to Files

```
>>> with open('bacon.txt', 'w') as bacon_file:
...     bacon_file.write('Hello world!\n')
13

>>> with open('bacon.txt', 'a') as bacon_file:
...     bacon_file.write('Bacon is not a vegetable.')
25

>>> with open('bacon.txt') as bacon_file:
...     content = bacon_file.read()

>>> print(content)
Hello world!
Bacon is not a vegetable.
```

Saving Variables with the shelve Module

To save variables:

```
>>> import shelve

>>> cats = ['Zophie', 'Pooka', 'Simon']
>>> with shelve.open('mydata') as shelf_file:
...     shelf_file['cats'] = cats
```

To open and read variables:

```
>>> with shelve.open('mydata') as shelf_file:
...     print(type(shelf_file))
...     print(shelf_file['cats'])
<class 'shelve.DbfilenameShelf'>
['Zophie', 'Pooka', 'Simon']
```

Just like dictionaries, shelf values have keys() and values() methods that will return list-like values of the keys and values in the shelf. Since these methods return list-like values instead of true lists, you should pass them to the list() function to get them in list form.

```
>>> with shelve.open('mydata') as shelf_file:
...     print(list(shelf_file.keys()))
...     print(list(shelf_file.values()))
['cats']
[['Zophie', 'Pooka', 'Simon']]
```

Saving Variables with the pprint.pformat() Function

```
>>> import pprint

>>> cats = [{'name': 'Zophie', 'desc': 'chubby'}, {'name': 'Pooka', 'desc': 'fluffy'}]

>>> pprint.pformat(cats)
"[{'desc': 'chubby', 'name': 'Zophie'}, {'desc': 'fluffy', 'name': 'Pooka'}]"

>>> with open('myCats.py', 'w') as file_obj:
...     file_obj.write('cats = {}\n'.format(pprint.pformat(cats)))
83
```

Reading ZIP Files

```
>>> import zipfile, os

>>> os.chdir('C:\\')    # move to the folder with example.zip
>>> with zipfile.ZipFile('example.zip') as example_zip:
...     print(example_zip.namelist())
...     spam_info = example_zip.getinfo('spam.txt')
...     print(spam_info.file_size)
...     print(spam_info.compress_size)
...     print('Compressed file is %sx smaller!' % (round(spam_info.file_size / spam_info.compress_size, 2)))

['spam.txt', 'cats/', 'cats/catnames.txt', 'cats/zophie.jpg']
13908
3828
'Compressed file is 3.63x smaller!'
```

Extracting from ZIP Files

The extractall() method for ZipFile objects extracts all the files and folders from a ZIP file into the current working directory.

```
>>> import zipfile, os

>>> os.chdir('C:\\')    # move to the folder with example.zip

>>> with zipfile.ZipFile('example.zip') as example_zip:
...     example_zip.extractall()
```

The extract() method for ZipFile objects will extract a single file from the ZIP file. Continue the interactive shell example:

```
>>> with zipfile.ZipFile('example.zip') as example_zip:
...     print(example_zip.extract('spam.txt'))
...     print(example_zip.extract('spam.txt', 'C:\\some\\new\\folders'))
'C:\\spam.txt'
'C:\\some\\new\\folders\\spam.txt'
```

Creating and Adding to ZIP Files

```
>>> import zipfile

>>> with zipfile.ZipFile('new.zip', 'w') as new_zip:
...     new_zip.write('spam.txt', compress_type=zipfile.ZIP_DEFLATED)
```

This code will create a new ZIP file named new.zip that has the compressed contents of spam.txt.

JSON, YAML and configuration files

JSON

Open a JSON file with:

```
import json
with open("filename.json", "r") as f:
    content = json.loads(f.read())
```

Write a JSON file with:

```
import json

content = {"name": "Joe", "age": 20}
with open("filename.json", "w") as f:
    f.write(json.dumps(content, indent=2))
```

YAML

Compared to JSON, YAML allows for much better human maintainability and gives you the option to add comments. It is a convinient choice for configuration files where humans will have to edit it.

There are two main librairies allowing to access to YAML files:

- PyYaml
- Ruamel.yaml

Install them using `pip install` in your virtual environment.

The first one it easier to use but the second one, Ruamel, implements much better the YAML specification, and allow for example to modify a YAML content without altering comments.

Open a YAML file with:

```
from ruamel.yaml import YAML

with open("filename.yaml") as f:
    yaml=YAML()
    yaml.load(f)
```

Anyconfig

Anyconfig is a very handy package allowing to abstract completly the underlying configuration file format. It allows to load a Python dictionary from JSON, YAML, TOML, and so on.

Install it with:

```
pip install anyconfig
```

Usage:

```
import anyconfig

conf1 = anyconfig.load("/path/to/foo/conf.d/a.yml")
```

Debugging

Raising Exceptions

Exceptions are raised with a raise statement. In code, a raise statement consists of the following:

- The raise keyword
- A call to the Exception() function
- A string with a helpful error message passed to the Exception() function

```
>>> raise Exception('This is the error message.')
Traceback (most recent call last):
  File "<pyshell#191>", line 1, in <module>
    raise Exception('This is the error message.')
Exception: This is the error message.
```

Often it's the code that calls the function, not the function itself, that knows how to handle an expection. So you will commonly see a raise statement inside a function and the try and except statements in the code calling the function.

```
def box_print(symbol, width, height):
    if len(symbol) != 1:
      raise Exception('Symbol must be a single character string.')
    if width <= 2:
      raise Exception('Width must be greater than 2.')
    if height <= 2:
      raise Exception('Height must be greater than 2.')
    print(symbol * width)
    for i in range(height - 2):
        print(symbol + (' ' * (width - 2)) + symbol)
    print(symbol * width)
for sym, w, h in (('*', 4, 4), ('O', 20, 5), ('x', 1, 3), ('ZZ', 3, 3)):
    try:
        box_print(sym, w, h)
    except Exception as err:
        print('An exception happened: ' + str(err))
```

Getting the Traceback as a String

The traceback is displayed by Python whenever a raised exception goes unhandled. But can also obtain it as a string by calling traceback.format_exc(). This function is useful if you want the information from an exception's traceback but also want an except statement to gracefully handle the exception. You will need to import Python's traceback module before calling this function.

```
>>> import traceback

>>> try:
>>>      raise Exception('This is the error message.')
>>> except:
>>>      with open('errorInfo.txt', 'w') as error_file:
>>>          error_file.write(traceback.format_exc())
>>>      print('The traceback info was written to errorInfo.txt.')
116
The traceback info was written to errorInfo.txt.
```

The 116 is the return value from the write() method, since 116 characters were written to the file. The traceback text was written to errorInfo.txt.

```
Traceback (most recent call last):
  File "<pyshell#28>", line 2, in <module>
Exception: This is the error message.
```

Assertions

An assertion is a sanity check to make sure your code isn't doing something obviously wrong. These sanity checks are performed by assert statements. If the sanity check fails, then an AssertionError exception is raised. In code, an assert statement consists of the following:

- The assert keyword
- A condition (that is, an expression that evaluates to True or False)
- A comma
- A string to display when the condition is False

```
>>> pod_bay_door_status = 'open'

>>> assert pod_bay_door_status == 'open', 'The pod bay doors need to be "open".'

>>> pod_bay_door_status = 'I\'m sorry, Dave. I\'m afraid I can\'t do that.'

>>> assert pod_bay_door_status == 'open', 'The pod bay doors need to be "open".'

Traceback (most recent call last):
  File "<pyshell#10>", line 1, in <module>
    assert pod_bay_door_status == 'open', 'The pod bay doors need to be "open".'
AssertionError: The pod bay doors need to be "open".
```

In plain English, an assert statement says, "I assert that this condition holds true, and if not, there is a bug somewhere in the program." Unlike exceptions, your code should not handle assert statements with try and except; if an assert fails, your program should crash. By failing fast like this, you shorten the time between the original cause of the bug and when you first notice the bug. This will reduce the amount of code you will have to check before finding the code that's causing the bug.

Disabling Assertions

Assertions can be disabled by passing the -O option when running Python.

Logging

To enable the logging module to display log messages on your screen as your program runs, copy the following to the top of your program (but under the #! python shebang line):

```
import logging

logging.basicConfig(level=logging.DEBUG, format=' %(asctime)s - %(levelname)s- %(message)s')
```

Say you wrote a function to calculate the factorial of a number. In mathematics, factorial 4 is 1 × 2 × 3 × 4, or 24. Factorial 7 is 1 × 2 × 3 × 4 × 5 × 6 × 7, or 5,040. Open a new file editor window and enter the following code. It has a bug in it, but you will also enter several log messages to help yourself figure out what is going wrong. Save the program as factorialLog.py.

```
 >>> import logging
>>>
>>> logging.basicConfig(level=logging.DEBUG, format=' %(asctime)s - %(levelname)s- %(message)s')
>>>
>>> logging.debug('Start of program')
>>>
>>> def factorial(n):
>>>
>>>     logging.debug('Start of factorial(%s)' % (n))
>>>     total = 1
>>>
>>>     for i in range(1, n + 1):
>>>         total *= i
>>>         logging.debug('i is ' + str(i) + ', total is ' + str(total))
>>>
>>>     logging.debug('End of factorial(%s)' % (n))
>>>
>>>     return total
>>>
>>> print(factorial(5))
>>> logging.debug('End of program')
2015-05-23 16:20:12,664 - DEBUG - Start of program
2015-05-23 16:20:12,664 - DEBUG - Start of factorial(5)
2015-05-23 16:20:12,665 - DEBUG - i is 0, total is 0
2015-05-23 16:20:12,668 - DEBUG - i is 1, total is 0
2015-05-23 16:20:12,670 - DEBUG - i is 2, total is 0
2015-05-23 16:20:12,673 - DEBUG - i is 3, total is 0
2015-05-23 16:20:12,675 - DEBUG - i is 4, total is 0
2015-05-23 16:20:12,678 - DEBUG - i is 5, total is 0
2015-05-23 16:20:12,680 - DEBUG - End of factorial(5)
0
2015-05-23 16:20:12,684 - DEBUG - End of program
```

Logging Levels

Logging levels provide a way to categorize your log messages by importance. There are five logging levels, described in Table 10-1 from least to most important. Messages can be logged at each level using a different logging function.

Level	Logging Function	Description
`DEBUG`	`logging.debug()`	The lowest level. Used for small details. Usually you care about these messages only when diagnosing problems.
`INFO`	`logging.info()`	Used to record information on general events in your program or confirm that things are working at their point in the program.
`WARNING`	`logging.warning()`	Used to indicate a potential problem that doesn’t prevent the program from working but might do so in the future.
`ERROR`	`logging.error()`	Used to record an error that caused the program to fail to do something.

Level	Logging Function	Description
`CRITICAL`	`logging.critical()`	The highest level. Used to indicate a fatal error that has caused or is about to cause the program to stop running entirely.

Disabling Logging

After you've debugged your program, you probably don't want all these log messages cluttering the screen. The logging.disable() function disables these so that you don't have to go into your program and remove all the logging calls by hand.

```
>>> import logging

>>> logging.basicConfig(level=logging.INFO, format=' %(asctime)s -%(levelname)s - %(message)s')

>>> logging.critical('Critical error! Critical error!')
2015-05-22 11:10:48,054 - CRITICAL - Critical error! Critical error!

>>> logging.disable(logging.CRITICAL)

>>> logging.critical('Critical error! Critical error!')

>>> logging.error('Error! Error!')
```

Logging to a File

Instead of displaying the log messages to the screen, you can write them to a text file. The logging.basicConfig() function takes a filename keyword argument, like so:

```
import logging

logging.basicConfig(filename='myProgramLog.txt', level=logging.DEBUG, format='%(asctime)s - %(levelname)s - %(message)s')
```

Lambda Functions

This function:

```
>>> def add(x, y):
        return x + y

>>> add(5, 3)
8
```

Is equivalent to the *lambda* function:

```
>>> add = lambda x, y: x + y
>>> add(5, 3)
8
```

It's not even need to bind it to a name like add before:

```
>>> (lambda x, y: x + y)(5, 3)
8
```

Like regular nested functions, lambdas also work as lexical closures:

```
>>> def make_adder(n):
        return lambda x: x + n

>>> plus_3 = make_adder(3)
>>> plus_5 = make_adder(5)

>>> plus_3(4)
7
>>> plus_5(4)
9
```

Note: lambda can only evaluate an expression, like a single line of code.

Ternary Conditional Operator

Many programming languages have a ternary operator, which define a conditional expression. The most common usage is to make a terse simple conditional assignment statement. In other words, it offers one-line code to evaluate the first expression if the condition is true, otherwise it evaluates the second expression.

```
<expression1> if <condition> else <expression2>
```

Example:

```
>>> age = 15

>>> print('kid' if age < 18 else 'adult')
kid
```

Ternary operators can be chained:

```
>>> age = 15

>>> print('kid' if age < 13 else 'teenager' if age < 18 else 'adult')
teenager
```

The code above is equivalent to:

```
if age < 18:
    if age < 13:
        print('kid')
    else:
        print('teenager')
else:
    print('adult')
```

args and kwargs

The names `args and kwargs` are arbitrary - the important thing are the `*` and `**` operators. They can mean:

1. In a function declaration, `*` means "pack all remaining positional arguments into a tuple named `<name>`", while `**` is the same for keyword arguments (except it uses a dictionary, not a tuple).
2. In a function call, `*` means "unpack tuple or list named `<name>` to positional arguments at this position", while `**` is the same for keyword arguments.

For example you can make a function that you can use to call any other function, no matter what parameters it has:

```
def forward(f, *args, **kwargs):
    return f(*args, **kwargs)
```

Inside forward, args is a tuple (of all positional arguments except the first one, because we specified it - the f), kwargs is a dict. Then we call f and unpack them so they become normal arguments to f.

You use `*args` when you have an indefinite amount of positional arguments.

```
>>> def fruits(*args):
>>>    for fruit in args:
>>>       print(fruit)

>>> fruits("apples", "bananas", "grapes")

"apples"
"bananas"
"grapes"
```

Similarly, you use `**kwargs` when you have an indefinite number of keyword arguments.

```
>>> def fruit(**kwargs):
>>>    for key, value in kwargs.items():
>>>        print("{0}: {1}".format(key, value))

>>> fruit(name = "apple", color = "red")

name: apple
color: red
```

```
>>> def show(arg1, arg2, *args, kwarg1=None, kwarg2=None, **kwargs):
>>>   print(arg1)
>>>   print(arg2)
>>>   print(args)
>>>   print(kwarg1)
>>>   print(kwarg2)
>>>   print(kwargs)

>>> data1 = [1,2,3]
>>> data2 = [4,5,6]
>>> data3 = {'a':7,'b':8,'c':9}

>>> show(*data1,*data2, kwarg1="python",kwarg2="cheatsheet",**data3)
1
2
(3, 4, 5, 6)
python
cheatsheet
{'a': 7, 'b': 8, 'c': 9}

>>> show(*data1, *data2, **data3)
1
2
(3, 4, 5, 6)
None
None
{'a': 7, 'b': 8, 'c': 9}

# If you do not specify ** for kwargs
>>> show(*data1, *data2, *data3)
1
2
(3, 4, 5, 6, "a", "b", "c")
None
None
{}
```

Things to Remember(args)

1. Functions can accept a variable number of positional arguments by using `*args` in the def statement.
2. You can use the items from a sequence as the positional arguments for a function with the `*` operator.
3. Using the `*` operator with a generator may cause your program to run out of memory and crash.

4. Adding new positional parameters to functions that accept `*args` can introduce hard-to-find bugs.

Things to Remember(kwargs)

1. Function arguments can be specified by position or by keyword.
2. Keywords make it clear what the purpose of each argument is when it would be confusing with only positional arguments.
3. Keyword arguments with default values make it easy to add new behaviors to a function, especially when the function has existing callers.
4. Optional keyword arguments should always be passed by keyword instead of by position.

Context Manager

While Python's context managers are widely used, few understand the purpose behind their use. These statements, commonly used with reading and writing files, assist the application in conserving system memory and improve resource management by ensuring specific resources are only in use for certain processes.

with statement

A context manager is an object that is notified when a context (a block of code) starts and ends. You commonly use one with the with statement. It takes care of the notifying.

For example, file objects are context managers. When a context ends, the file object is closed automatically:

```
>>> with open(filename) as f:
>>>     file_contents = f.read()

# the open_file object has automatically been closed.
```

Anything that ends execution of the block causes the context manager's exit method to be called. This includes exceptions, and can be useful when an error causes you to prematurely exit from an open file or connection. Exiting a script without properly closing files/connections is a bad idea, that may cause data loss or other problems. By using a context manager you can ensure that precautions are always taken to prevent damage or loss in this way.

Writing your own contextmanager using generator syntax

It is also possible to write a context manager using generator syntax thanks to the `contextlib.contextmanager` decorator:

```
>>> import contextlib
>>> @contextlib.contextmanager
... def context_manager(num):
...     print('Enter')
...     yield num + 1
...     print('Exit')
>>> with context_manager(2) as cm:
...     # the following instructions are run when the 'yield' point of the context
...     # manager is reached.
...     # 'cm' will have the value that was yielded
...     print('Right in the middle with cm = {}'.format(cm))
Enter
Right in the middle with cm = 3
Exit

>>>
```

`__main__` Top-level script environment

`__main__` is the name of the scope in which top-level code executes. A module's **name** is set equal to `__main__` when read from standard input, a script, or from an interactive prompt.

A module can discover whether or not it is running in the main scope by checking its own `__name__`, which allows a common idiom for conditionally executing code in a module when it is run as a script or with `python -m` but not when it is imported:

```
>>> if __name__ == "__main__":
...     # execute only if run as a script
...     main()
```

For a package, the same effect can be achieved by including a **main**.py module, the contents of which will be executed when the module is run with -m

For example we are developing script which is designed to be used as module, we should do:

```
>>> # Python program to execute function directly
>>> def add(a, b):
...     return a+b
...
>>> add(10, 20) # we can test it by calling the function save it as calculate.py
30
>>> # Now if we want to use that module by importing we have to comment out our call,
>>> # Instead we can write like this in calculate.py
>>> if __name__ == "__main__":
...     add(3, 5)
...
>>> import calculate
>>> calculate.add(3, 5)
8
```

Advantages

1. Every Python module has it's `__name__` defined and if this is `__main__`, it implies that the module is being run standalone by the user and we can do corresponding appropriate actions.
2. If you import this script as a module in another script, the **name** is set to the name of the script/module.
3. Python files can act as either reusable modules, or as standalone programs.
4. if `__name__ == "main":` is used to execute some code only if the file was run directly, and not imported.

setup.py

The setup script is the centre of all activity in building, distributing, and installing modules using the Distutils. The main purpose of the setup script is to describe your module distribution to the Distutils, so that the various commands that operate on your modules do the right thing.

The `setup.py` file is at the heart of a Python project. It describes all of the metadata about your project. There a quite a few fields you can add to a project to give it a rich set of metadata describing the project. However, there are only three required fields: name, version, and packages. The name field must be unique if you wish to publish your package on the Python Package Index (PyPI). The version field keeps track of different releases of the project. The packages field describes where you've put the Python source code within your project.

This allows you to easily install Python packages. Often it's enough to write:

```
python setup.py install
```

and module will install itself.

Our initial setup.py will also include information about the license and will re-use the README.txt file for the long_description field. This will look like:

```
>>> from distutils.core import setup
>>> setup(
...    name='pythonCheatsheet',
...    version='0.1',
...    packages=['pipenv',],
...    license='MIT',
...    long_description=open('README.txt').read(),
... )
```

Find more information visit http://docs.python.org/install/index.html.

Dataclasses

`Dataclasses` are python classes but are suited for storing data objects. This module provides a decorator and functions for automatically adding generated special methods such as `__init__()` and `__repr__()` to user-defined classes.

Features

1. They store data and represent a certain data type. Ex: A number. For people familiar with ORMs, a model instance is a data object. It represents a specific kind of entity. It holds attributes that define or represent the entity.
2. They can be compared to other objects of the same type. Ex: A number can be greater than, less than, or equal to another number.

Python 3.7 provides a decorator dataclass that is used to convert a class into a dataclass.

python 2.7

```
>>> class Number:
...     def __init__(self, val):
...         self.val = val
...
>>> obj = Number(2)
>>> obj.val
2
```

with dataclass

```
>>> @dataclass
... class Number:
...     val: int
...
>>> obj = Number(2)
>>> obj.val
2
```

Default values

It is easy to add default values to the fields of your data class.

```
>>> @dataclass
... class Product:
...     name: str
...     count: int = 0
...     price: float = 0.0
...
>>> obj = Product("Python")
>>> obj.name
Python
>>> obj.count
0
>>> obj.price
0.0
```

Type hints

It is mandatory to define the data type in dataclass. However, If you don't want specify the datatype then, use `typing.Any`.

```
>>> from dataclasses import dataclass
>>> from typing import Any

>>> @dataclass
... class WithoutExplicitTypes:
...     name: Any
...     value: Any = 42
...
```

Virtual Environment

The use of a Virtual Environment is to test python code in encapsulated environments and to also avoid filling the base Python installation with libraries we might use for only one project.

virtualenv

1. Install virtualenv

   ```
   pip install virtualenv
   ```

2. Install virtualenvwrapper-win (Windows)

   ```
   pip install virtualenvwrapper-win
   ```

Usage:

1. Make a Virtual Environment

   ```
   mkvirtualenv HelloWold
   ```

   Anything we install now will be specific to this project. And available to the projects we connect to this environment.

2. Set Project Directory

   To bind our virtualenv with our current working directory we simply enter:

   ```
   setprojectdir .
   ```

3. Deactivate

   To move onto something else in the command line type ‘deactivate’ to deactivate your environment.

   ```
   deactivate
   ```

   Notice how the parenthesis disappear.

4. Workon

   Open up the command prompt and type ‘workon HelloWold’ to activate the environment and move into your root project folder

   ```
   workon HelloWold
   ```

poetry

> Poetry is a tool for dependency management and packaging in Python. It allows you to declare the libraries your project depends on and it will manage (install/update) them for you.

1. Install Poetry

   ```
   pip install --user poetry
   ```

2. Create a new project

   ```
   poetry new my-project
   ```

   This will create a my-project directory:

   ```
   my-project
   ├── pyproject.toml
   ├── README.rst
   ├── poetry_demo
   │   └── __init__.py
   └── tests
       ├── __init__.py
       └── test_poetry_demo.py
   ```

   The pyproject.toml file will orchestrate your project and its dependencies:

   ```
   [tool.poetry]
   name = "my-project"
   version = "0.1.0"
   description = ""
   authors = ["your name <your@mail.com>"]

   [tool.poetry.dependencies]
   python = "*"

   [tool.poetry.dev-dependencies]
   pytest = "^3.4"
   ```

3. Packages

   To add dependencies to your project, you can specify them in the tool.poetry.dependencies section:

```
[tool.poetry.dependencies]
pendulum = "^1.4"
```

Also, instead of modifying the pyproject.toml file by hand, you can use the add command and it will automatically find a suitable version constraint.

```
$ poetry add pendulum
```

To install the dependencies listed in the pyproject.toml:

```
poetry install
```

To remove dependencies:

```
poetry remove pendulum
```

For more information, check the documentation.

pipenv

> Pipenv is a tool that aims to bring the best of all packaging worlds (bundler, composer, npm, cargo, yarn, etc.) to the Python world. Windows is a first-class citizen, in our world.

1. Install pipenv

   ```
   pip install pipenv
   ```

2. Enter your Project directory and install the Packages for your project

   ```
   cd my_project
   pipenv install <package>
   ```

   Pipenv will install your package and create a Pipfile for you in your project's directory. The Pipfile is used to track which dependencies your project needs in case you need to re-install them.

3. Uninstall Packages

   ```
   pipenv uninstall <package>
   ```

4. Activate the Virtual Environment associated with your Python project

   ```
   pipenv shell
   ```

5. Exit the Virtual Environment

   ```
   exit
   ```

Find more information and a video in docs.pipenv.org.

anaconda

Anaconda is another popular tool to manage python packages.

> Where packages, notebooks, projects and environments are shared. Your place for free public conda package hosting.

Usage:

1. Make a Virtual Environment

   ```
   conda create -n HelloWorld
   ```

2. To use the Virtual Environment, activate it by:

   ```
   conda activate HelloWorld
   ```

   Anything installed now will be specific to the project HelloWorld

3. Exit the Virtual Environment

   ```
   conda deactivate
   ```