launch binder

About

Basic cheatsheet for Python mostly based on the book written by Al Sweigart, Automate the Boring Stuff with Python under the Creative Commons license and many other sources.

Contribute

All contributions are welcome:

- Read the issues, Fork the project and do a Pull Request.
- Request a new topic creating a `New issue` with the `enhancement` tag.
- Find any kind of errors in the cheat sheet and create a `New issue` with the details or fork the project and do a Pull Request.
- Suggest a better or more pythonic way for existing examples.

Read It

- Website
- Github
- PDF
- Jupyter Notebook

Python Cheatsheet

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!
```

Return to the Top

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
```

Return to the Top

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'`

Return to the Top

String Concatenation and Replication

String concatenation:

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

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

String Replication:

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

Return to the Top

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.

Return to the Top

Comments

Inline comment:

```
# This is a comment
```

Multiline comment:

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

Multiline comment with multiline string:

```
"""
This is a
multiline comment
"""
```

Code with a comment:

```
a = 1  # initialization
```

Please note the two spaces in front of the comment.

Return to the Top

The print() Function

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

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

Return to the Top

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
```

Return to the Top

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!")
```

Return to the Top

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
```

Return to the Top

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
```

Return to the Top

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`
`not False`	`True`

Return to the Top

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
```

Return to the Top

if Statements

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

Return to the Top

else Statements

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

Return to the Top

while Loop Statements

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

Return to the Top

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!')
```

[Return to the Top](#)

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.')
```

[Return to the Top](#)

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
```

[Return to the Top](#)

Importing Modules

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

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

[Return to the Top](#)

Functions

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

[Return to the Top](#)

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)
```

Return to the Top

The None Value

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

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

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

Return to the Top

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
```

Return to the Top

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.

Return to the Top

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.

Return to the Top

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
```

[Return to the Top](#)

Lists

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

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

[Return to the Top](#)

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'
```

[Return to the Top](#)

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.'
```

[Return to the Top]

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']
```

[Return to the Top]

Getting a List’s Length with len()

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

Return to the Top

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]
```

Return to the Top

Using for Loops with Lists

Direct

```
>>> supplies = ['pens', 'staplers', 'flame-throwers', 'binders']
>>> for supply in supplies:
>>>     print('Supplies contains: {}'.format(supply))
Supplies contains: pens
Supplies contains: staplers
Supplies contains: flame-throwers
Supplies contains: binders
```

With automatic enumeration (assigning indices from 0 to len(list)-1)

```
>>> 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
```

Return to the Top

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
```

Return to the Top

The Multiple Assignment Trick (List unpacking)

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'
```

[Return to the Top](#)

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']
```

[Return to the Top](#)

Adding Values to Lists with the append()

append():

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

>>> spam.append('moose')

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

[Return to the Top](#)

Dictionaries and Structuring Data

Example Dictionary:

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

Return to the Top

The items() Method

items():

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

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

```
>>> 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
```

Return to the Top

Checking Whether a Key Exists in a Dictionary

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

```
>>> 'color' in spam
False
```

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

Manipulating Strings

Escape Characters

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

`\\`	Backslash
`\b`	Backspace
`\ooo`	Octal value
`\r`	Carriage Return

Example:

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

Return to the Top

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
```

Return to the Top

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'
```

Return to the Top

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
```

Return to the Top

The upper(), lower() String Methods

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

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

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

Return to the Top

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']
```

Return to the Top

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.

Return to the Top

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.*

Return to the Top

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.'
```

Return to the Top

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.

Return to the Top

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
```

Return to the Top

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.”

Return to the Top

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
```

`pathlib` provides a lot more functionality. Check out the official documentation if you want to know more!

Return to the Top

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.

Return to the Top

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 text:

>>> 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,
```

Return to the Top

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.
```

Return to the Top

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))
```

Return to the Top

YAML

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

There are two main libraries 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)
```

Return to the Top

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 exception. 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))
```

Return to the Top

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.

Return to the Top

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.

Return to the Top

`__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.

Return to the Top

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.

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
   ```

Return to the Top