## **High Order Functions map, reduce**

## **Map**

A common pattern in any programming language is to transform a collection by applying a function to each element. In a procedural-looking way:

**output = []**

**for element in iterable:**

**val = function(element)**

**output.append(val)**

## We can accomplish the same task in a declarative-looking way:

**output = [**

**function(element)**

**for element in iterable**

**]**

## For example, if we have

## **languages = ["python", "perl", "java", "c++"], then [len(s) for s in languages] evaluates to [6, 4, 4, 3].**

But really, what are we even doing? We're just applying a function over a collection - it doesn't quite matter exactly how the collection is built up. In this case, Python provides us with a new function - the map function:

## **map(fn, iter)**

The map function transforms a stream of data from an iterable and produces a stream of data by applying the function to each element. Interestingly, the first argument to map is a function object! This is the first time we've seen a function object be passed as an argument to another function.

Practically, this means that Python lets us rewrite the above example to:

## **map(len, languages)**

## The map function doesn't actual produce a list, though. It returns an object which we can consume to get the transformed values – for example, we could see that:

## **tuple(map(len, languages)) # => (6, 4, 4, 3)**

## 

## 

## **Removing Elements With filter**

## **Filter**

## Another common pattern in any programming language is to transform a collection by applying a function to each element. In a procedural-looking way:

**output = []**

**for element in iterable:**

**if predicate(element):**

**output.append(element)**

## We can accomplish the same task in a declarative-looking way:

**output = [**

**element**

**for element in iterable**

**if predicate(element)**

**]**

## If we wanted to find only the members from our rolodex, we might run:

## **[client for client in rolodex if is\_member(client)]**

## But really, what are we even doing? We're just filtering the elements of a collection with a predicate function - it doesn't quite matter exactly how the collection is built up. In this case, Python provides us with another new function - the filter function:

## **filter(predicate\_function, iterable)**

## The filter function filters only the elements from a stream of data that pass through a predicate function. As before, the first argument to filter is a function object!

## Practically, this means that Python lets us rewrite the above example to:

## **filter(is\_member, rolodex)**

## As before, the filter function doesn't actual produce a list, though. It returns an object which we can consume to get the filtered values.

## **Map and Filter In Action**

### **Aesthetics of map and filter**

## Benefits

## Compute data-on-demand, don't buffer it.

## Faster than list comprehensions in some cases.

## Beauty?

## Up to you. An elegant functional reframing of the problem, or an unnecessary tool that's more pain than its worth?

### **New Terms**

|  |  |
| --- | --- |
| **Term** | **Definition** |
| **filter** | **A built-in function that filters an iterable by keeping only elements that successfully pass a predicate function.** |
| **map** | **A built-in function that applies a function to every element of an iterable.** |

### **Further Reading**

## [filter](https://docs.python.org/3/library/functions.html#filter)**: The built-in filter function.**

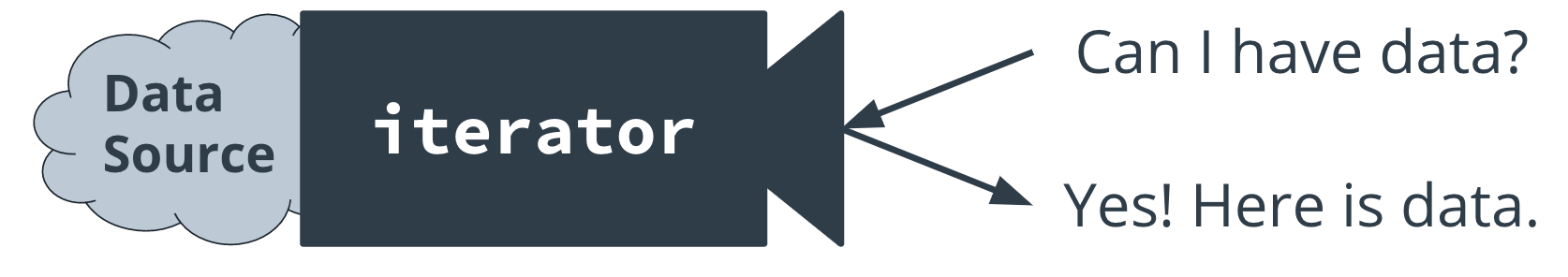
## [map](https://docs.python.org/3/library/functions.html#map)**: The built-in map function.**

## [Map, Filter, and Reduce (Learn Python)](https://www.learnpython.org/en/Map,_Filter,_Reduce)**: Learn Python's overview of map, filter, and reduce.**

## **Iterators**

An **iterator** represents a (finite or infinite) stream of data.

The next(iter) call asks an iterator to yield a successive value. If there are no more values, it raises StopIteration. The iter(data) function produces an iterator from an iterable data source.



### **QUESTION 1 OF 2**

Which of the following are best described as streaming data - that is, which of the following might be represented as iterators in Python?

* The names of reunion attendees who walk through the front door.
* All of the first names of teachers at your high school.
* The colors of fish that pass by a particular spot in a river.
* The words in the English language.
* Keyboard events as a user types their name into an input form.

## **Using Iterators**

You can build an iterator from data:

**# Build an iterator over [1,2,3]**

**it = iter([1,2,3])**

**next(it) # => 1**

**next(it) # => 2**

**next(it) # => 3**

**next(it) # raises StopIteration error**

Iterators are fundamental to the language. The humble for loop is really using an iterator!

**for element in data:**

**process(data)**

**# actually behaves like**

**for element in iter(data):**

**process(element)**

Built-in functions can consume iterables:

* max(iterable)
* min(iterable)
* any(iterable)
* all(iterable)
* element in iterable

Some built-in functions even produce iterables:

* range(stop)
* enumerate(iterable)
* zip(\*iterables)
* map(fn, iterable)
* filter(pred, iterable)

Iterators maintain some semblance of state:

**it = iter(range(100))**

**66 in it**

**next(it) # => 67**

## **Thinking about Iterators**

Consider the code snippet:

nine\_is\_a\_square\_with\_map = 9 **in** map(**lambda** x: x \*\* 2, range(1000000))

nine\_is\_a\_square\_with\_listcomp = 9 **in** [x \*\* 2 **for** x **in** range(1000000)]

Using what you now know about iterators, describe - in a few sentences - what happens in each of the above lines. What are the tradeoffs of using one over the other?

## **Generator Expressions**

Generator expressions are a useful feature for describing a stream of data inline. They behave a bit like a "lazy list comprehension."

**gen = (costly\_fn(data) for data in iterable)**

**print(gen) # <generator object <genexpr> at 0x109055cf0>**

**next(gen) # => The first transformed element.**

**next(gen) # => The second transformed element.**

Generator expressions are appropriate when you want data-on-demand, but you might not want to compute all of it at once in memory. Consider:

needle **in**

(expensive\_fn(item) **for** item **in** haystack)

needle **in**

[expensive\_fn(item) **for** item **in** haystack]

The top expression only calculates expensive\_fn as it needs to, and will stop when it finds a match. The bottom expression first evaluates the entire list comprehension, and the begins the search. So, the top expression can be more efficient in some cases.

## **Generator Functions**

A **generator function** looks like a normal function, except it contains the keyword yield.

When called, a generator function returns a generator iterator that can produce subsequent values on demand by running the function until it encounters a yield statement, and then pausing. In this way, generators are an advanced way to describe a stream of data.

To build a generator function, define a function containing the yield keyword. To use it, call the generator function to get a generator iterator, and iterator over it to your heart's content.

**def generate\_ints(n):**

**for i in range(n):**

**yield i**

**g = generate\_ints(3) # Doesn't start the function! Just sets up the iterator**

**type(g) # => <class 'generator'>**

**next(g) # => 0. Run until the next yield statement.**

**next(g) # => 1. Run until the next yield statement.**

**next(g) # => 2. Run until the next yield statement.**

**next(g) # raises StopIteration. Finished the function before finding another yield statement.**

This can be used to describe fancier streams of data, that aren't easily accomplished with built-ins or generator expressions:

**def generate\_fibs():**

**a, b = 0, 1**

**while True:**

**a, b = b, a + b**

**yield a**

**g = generate\_fibs()**

**next(g) # => 1**

**next(g) # => 1**

**next(g) # => 2**

**next(g) # => 3**

**next(g) # => 5**

**max(g) # Don't run this line of code. What happens?**

## **Infinite Data Streams**

Representations of infinite streams of data, like the generate\_fibs generator function, can be nicely paired with other consumers:

**def fibs\_under(n):**

**for fib in generate\_fibs(): # Loops over 1, 1, 2, ...**

**if fib > n:**

**break**

**print(fib)**

### **New Terms**

|  |  |
| --- | --- |
| **Term** | **Definition** |
| Generator Expression | A syntactical shortcut to create on-the-fly iterators with simple implementations. |
| Generator Function | A special type of function containing the keyword yield that can produce a stream of values when called. |
| Iterable | An object that can be iterated over. |
| Iterator | An object representing a stream of data that can yield successive values. |

### **Further Reading**

* [Built-in Functions](https://docs.python.org/3/library/functions.html): Built-in functions, many of which consume or produce iterables.
* [Generator Expression Syntax](https://docs.python.org/3/reference/expressions.html#generator-expressions): Language reference for generator expression syntax.
* [Generators in Python (RealPython)](https://realpython.com/introduction-to-python-generators/): How to use generators and yield in Python.
* [HOWTO: Generator Expressions](https://docs.python.org/3/howto/functional.html#generator-expressions-and-list-comprehensions): An introduction to the use of generator expressions.
* [HOWTO: Generators](https://docs.python.org/3/howto/functional.html#generators): An introduction to the use of generators.
* [HOWTO: Iterators](https://docs.python.org/3/howto/functional.html#iterators): An introduction to the concept of iterators.
* [Iterables vs. Iterators (GeeksForGeeks)](https://www.geeksforgeeks.org/python-difference-iterable-iterator/): An overview of the (subtle) differences between iterables and iterators.
* [PEP 289](https://www.python.org/dev/peps/pep-0289/): The PEP introducing generator expressions into the language.
* [Tribonacci Numbers (Wiki)](https://en.wikipedia.org/wiki/Generalizations_of_Fibonacci_numbers#Tribonacci_numbers): Wikipedia's overview of Tribonacci numbers.

## **Decorators**

Decorators are, by far, among the most satisfying features of Python. To understand them, we'll need to reframe some building blocks, because decorators are wild.

### **Functions as Arguments**

We've already seen that functions can be arguments – the map and filter function expected the first argument to be a function object. We can go one step further, and write a function that takes in everything we need to run another function – the function object, as well as variadic collections of arguments:

**def** **perform\_twice**(fn, \*args, \*\*kwargs):

fn(\*args, \*\*kwargs)

fn(\*args, \*\*kwargs)

perform\_twice(**print**, 5, 10, sep='&', end='...')

*# 5&10...5&10...*

The perform\_twice function takes a function and its arguments, and calls it twice. Notice that the parameter list includes a variadic positional parameter and a variadic keyword parameter. In the implementation, it unpacks these captured arguments into the fn. This is a common pattern in which a function, such as perform\_twice can capture any collection of arguments and forward them along to some captured function object, such as fn.

### **Functions as Return Values**

Can we also create functions and return them from other functions. Why, of course! We can write a function make\_divisibility\_test(n) that produces a function to check whether a number m is divisible by n:

**def** **make\_divisibility\_test**(n):

**def** **divisible\_by\_n**(m):

**return** m % n == 0

**return** divisible\_by\_n

div\_by\_3 = make\_divisibility\_test(3)

tuple(filter(div\_by\_3, range(10)) *# => (0, 3, 6, 9)*

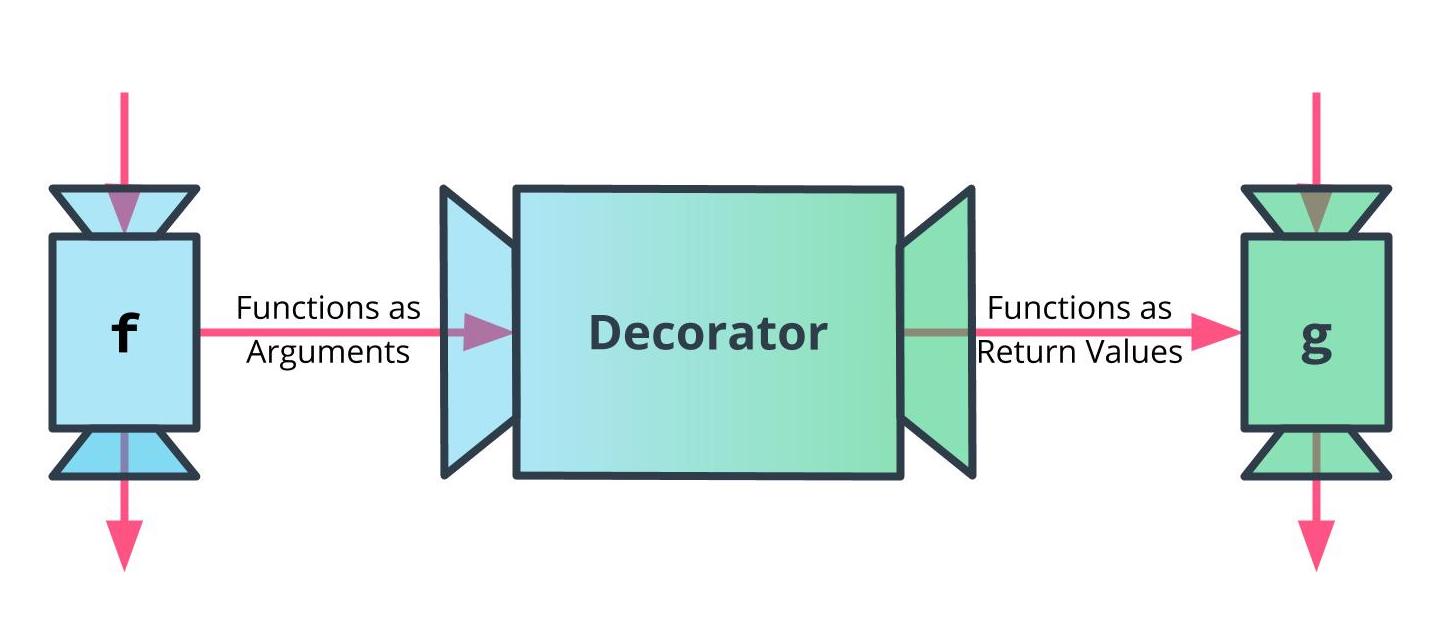
make\_divisibility\_test(5)(10) *# => True*

This might feel like a surprising tactic, but we can indeed write functions that define inner functions and return them to the outside world.

### **The Big Idea**

If we can send functions as arguments, and we can produce functions and return values, can we do both?

Yes – that's precisely what a decorator does.



Why Not Both?

## **Writing A Decorator**

We'll write a decorator that can modify any function so that it prints out its arguments when invoked so that we have an easier time debugging.

**def** **print\_args**(function):

**def** **wrapper**(\*args, \*\*kwargs):

print(args, kwargs)

**return** function(\*args, \*\*kwargs)

**return** wrapper

This decorator captures a function, and creates and returns a new function named wrapper. The wrapper function captures any collection of arguments, prints them out, and then forwards them to the supplied function. Let's take a look at this decorator in action

**def** **compute**(x, y, z=1):

**return** (x + y) \* z

compute(3, 5, z=2)

*# => 16*

compute\_log = print\_args(compute)

compute\_log(3, 5, z=2)

*# (3, 5) {'z': 2}*

*# => 16*

Wow! We modified a function to build a new one that can print out its arguments. It'd be a bit of a pain to refactor the rest of the codebase to use compute\_log instead of compute, so let's just rename compute itself.

compute = print\_args(compute)

compute(3, 5, z=2)

**# (3, 5) {'z': 2}**

**# => 16**

Now, anywhere else in the codebase that uses compute will use the version that prints its arguments instead. A nice feature for quick debugging!

### **Using the @ Sign**

### **Decorator Syntax**

It's a bit awkward to have to write def compute and then compute = print\_args(compute). If only there was a better way! In Python, there is.

@print\_args

**def** **compute**(x, y, z=1):

**return** (x + y) \* z

This new syntax (@decorator) applies a decorator to whatever's defined immediately below it. In this case, the compute function will be defined as normal, and then immediately passed through the print\_args decorator. In this way, we only need to toggle one line of code (@print\_args) to decorate our function with this new functionality. What's more, we can use this print\_args decorator on any function we'd like! We wrote one decorator, and we get to use it everywhere.

One more thing before we move on – our new wrapper function (which we've returned and renamed compute) doesn't necessarily have all of the accessories that accompanied the original function. For example, the original function's name and the original function's documentation string will be lost as is. Luckily, Python has a tool to help us – the functools module has a decorator of its own called functools.wraps that attaches the interesting accessories from our old function onto our new function. We'll just make a slight tweak to our decorator from before:

**import** functools

**def** **print\_args**(function):

@functools.wraps(function)

**def** **wrapper**(\*args, \*\*kwargs):

print(args, kwargs)

**return** function(\*args, \*\*kwargs)

**return** wrapper

The @functools.wraps(function) decorator update the wrapper function so that it clones important properties from function.

And that's it! We have a new decorator that can be used to help us debug any function we'd like, and moreover, we have the template for any other decorator we'd want to create.

### **Recap: Decorators**

Fundamentally, a decorator is a transformation that can be applied to a function.

Decorators are inconceivably useful, and appear in surprising places. They can be used to:

* Handle shared behavior to append to functions (as with print\_args)
* Cache return values to increase performance
* Set a timeout on blocking functions
* Mark class properties as "read-only"
* Mark methods as static methods or class methods
* Define event-driven handlers (for GUIs, or web-based clients)

Decorators are quite a complex topic - at the top of the functional food chain, they command respect. But if you can wrap your mind around decorators, and all of the details that make them possible, then you're in good shape to approach future problems from a functional design standpoint.

### **New Terms**

|  |  |
| --- | --- |
| **Term** | **Definition** |
| @decorator | Special syntax to apply a decorator to an immediately-subsequent function definition. |
| Decorator | A high-level function that traditionally transforms an input function into a slightly-modified output function with desired decorated behavior. |

### **Further Reading**

* [Decorators Primer (RealPython)](https://realpython.com/primer-on-python-decorators/): An involved - but really high-quality - treatment of decorators in Python.
* [PEP 318](https://www.python.org/dev/peps/pep-0318/): The PEP that introduced the decorator syntax into the language.
* [The decorator term](https://docs.python.org/3/glossary.html#term-decorator): Python's glossary definition of the term decorator.

**Overrides**

<https://www.youtube.com/watch?v=hJXKji285KY&t=253s>

As you practice inheritance and build more complex systems of classes, you'll find the need to redefine methods and functionality. This redefinition is known as **overriding** methods. Ultimately, it is as simple as re-writing a method body. You may need to override a method when:

* You are realizing an abstract class
* You are extending a class and the method definition is now dependent on new information in the child class
* You want to ensure an object fully realizes the base object's interface (i.e. **init**, **repr**, **str**)

Keep in mind, you can always access the parent method's implementation of a specific method using the super() method. This will allow you to extend a method without repeating code. For example:

**class** **BaseClass**():

**def** **simple\_method**():

**return** 'hello'

**class** **SimpleClass**(BaseClass):

**def** **simple\_method**():

**return** super().simple\_method() + ' world'

## **Class Methods**

Dynamic creation of specific instances. They’re not dependant on instantionating any objects.

<https://www.youtube.com/watch?time_continue=132&v=UzfrWvvIRuQ&feature=emb_logo>

**Exercise:**

class Dog():

def \_\_init\_\_(self, name:str, age:int,

breed:str, weight:int):

"""Create a new dog"""

self.breed = breed

self.weight = weight

self.name = name

self.age = age

def speak(self) -> None:

"""Make the dog bark"""

print(f'{self.name} says, "woof"')

# TODO: Override the \_\_gt\_\_ method to compare using the greater than operator

def \_\_gt\_\_(self, other):

return self.age > other.age

# TODO: Override the \_\_str\_\_ method to print the Dog's name instance variable

def \_\_str\_\_(self):

return self.name

if \_\_name\_\_ == "\_\_main\_\_":

sally = Dog('Sally', 6, 'chihuahua', 7)

henry = Dog('Henry', 7, 'terrier', 15)

if(sally > henry):

print(f'{sally} is older than {henry}')

else:

print(f'{henry} is older than {sally}')

# **Strategy Object Design Pattern**

## **Combining Our Skills - Encapsulating Third-Party Libraries**

As our system grows and implements more complex code, we'll want a way to organize complex code into a simpler interface. One way of achieving this goal is to *encapsulate* a more complex class or library into a simple class with limited functionality. We'll call this class a **strategy object** because it represents one possible strategy for achieving an action.

<https://www.youtube.com/watch?time_continue=143&v=jAuYdDRw1Vw&feature=emb_logo>

The strategy object design pattern provides many benefits, including:

* **Decreasing the complexity for developers consuming the library who do not need the entire functionality of the library.** For example, an encapsulated class may be designed to only provide one simplified method for working with the relevant data.
* **Easily replace a library without needing to make changes throughout a large codebase.** You can easily write a second strategy object and simply replace the old object with the new object. Since the object's methods are the same, no additional refactoring will be required.
* **Create additional classes to solve similar problems, and dynamically select the most appropriate class without needing to write additional code.** Similar to replacing an object, you can add additional objects to perform the same action in multiple ways. For example, you may need a different strategy object for different file types.

## **Static Methods**

There may be times when you'd like to define a method signature that will be consistently implemented in multiple ways.

For example, suppose we're implementing a conversion between two units (e.g. *inches to centimeters*, or *Fahrenheit to Celsius*). In these cases, we will not need to know any additional information outside of these inputs (i.e. instance variables), but would like a consistent signature so we can easily add a different strategy later (e.g., replace *Fahrenheit to Celsius* with *Fahrenheit to Kelvin*). In this scenario, it would be ideal to have an additional structure to ensure the interface is implemented consistently for each method. More concretely, we can define our method signature as convert(input) -> output, which is general enough that it can be the same across our conversion equations.

We could use an *abstract class* with an instance method, but this would require us to first create an object in memory. Instead, we can use a feature of Python called a **static method**. With a static method, we can define the method and make it accessible *before* we instantiate the class object.

A *static method* is similar to a *class method* in many ways, except it has no reference to the class or any of its default instance variables. Although static methods are not the most commonly used features in Python, they can be a helpful thing to have in your tool belt when the need arises.

Implementing the conversion example from before the video, we can create a 3 class relationship with a well defined abstract, static method. Using a different strategy object is as simple as re-assigning the strategy variable.

<https://www.youtube.com/watch?time_continue=176&v=CR6HNhi2bvs&feature=emb_logo>

from abc import ABC, abstractmethod

class ConversionStrategy(ABC):

@staticmethod

@abstractmethod

def convert(x):

pass

class FahrenheitToCelsiusConverter(ConversionStrategy):

@staticmethod

def convert(x):

return (x-32) \* 5 / 9

class CelsiusToFahrenheitConverter(ConversionStrategy):

@staticmethod

def convert(x):

return ( x \* 9 / 5 ) + 32

result = FahrenheitToCelsiusConverter.convert(32)

print(result)

Geeks for geeks: class methods vs static methods: <https://www.geeksforgeeks.org/class-method-vs-static-method-python/>

**Python in systems**

## **Outline**

In this lesson, we will learn how to:

* Package our application into a command line tool
* Explore advanced CLI tools in Python using *argparse*
* Execute other CLI tools in Python to expand capabilities using *subprocess*

Packaging our application into a command line tool will allow us to develop tools for users who are tech savvy and can execute commands, but may not be developers. Consuming command line tools in our applications will enable us to interface with other languages including legacy software.

The command line is a very useful tool with a tremendous number of applications. We've just gone over some of the basics so you can start using it in your Python projects. A great resource to continue practicing these skills is the [Bandit CLI Trainer](https://overthewire.org/wargames/bandit/). This game is designed to teach basic security principles and command-line tools. It is a fairly involved game and will take some time to complete, but it will provide a solid foundation.

NEXT

Argparse is better than sys:

import sys

name = sys.argv[1]

city = sys.argv[2]

print(f'hello, {name} from {city}')

# $ python argpars.py Lucia 'Horne Oresany'

VERSUS

import argparse

if \_\_name\_\_ == "\_\_main\_\_":

parser = argparse.ArgumentParser(description="Say a Greeting.")

parser.add\_argument('name', type=str)

parser.add\_argument('--city', type=str, # specifies an optional arg

default="San Fran",

help="where is the person from?")

args = parser.parse\_args()

name = args.name

city = args.city

print(f'Hello, {name} from {city}')

# python argparser.py --help

# python argparser.py Monika, Lisbon

# **Advanced CLI - Subprocess**

We now understand how we can make our software accessible from the command line, but what if we wanted to use other software that is available from the command line *within our Python application*? To accomplish this goal, we can use a Python module called subprocess, which will allow us to interface with software that we'd normally execute from a terminal window.

### **QUIZ QUESTION**

Why might we wish to use command-line tools inside of Python code?

(Select all that apply.)

* Legacy, existing software exists but we need to use it in a new system
* Other languages may be better equipped for the task (e.g. a complex math function may be faster using c)
* We like the other language more
* The problem is complex and a solution exists but there is no Python solution
* Our development team does not know python

**Some Useful Command Line Tools:**

* [FFMPEG](https://www.ffmpeg.org/) - a useful command-line tool to manipulate videos
* [Emoj](https://github.com/sindresorhus/emoj) - a command-line tool to select emojis
* [ImageMagick](https://imagemagick.org/index.php) - a command-line tool to manipulate images
* [List of Awesome CLI Apps](https://github.com/agarrharr/awesome-cli-apps) - a list of many different command-line tools in many different languages

## **Using subprocess.run**

Popen has been supported since the start of Python3, and you'll see it used quite often. However, since Python 3.5 a simpler run interface has been introduced ([docs](https://docs.python.org/3/library/subprocess.html#subprocess.run)). You should be aware of (and know how to use) both of these.

Try refactoring your code to use the simpler subprocess.run interface!

**import** subprocess

**import** random

p **=** subprocess.run(['emoj', 'dog'], stdout**=**subprocess.PIPE)

emoji **=** p.stdout.decode('utf-8').split(' ')

print(random.choice(emoji))

# 

# **Using Subprocess to Interface with CLI Tools**[**¶**](https://r950555c959637xjupyterlq5a2zquk.udacity-student-workspaces.com/lab?#Using-Subprocess-to-Interface-with-CLI-Tools)

We will be building upon a prior exercise. In those exercises we installed the following dependenencies:

pip install -U setuptools

pip install python-docx

pip install pandas

# **Try it!**[**¶**](https://r950555c959637xjupyterlq5a2zquk.udacity-student-workspaces.com/lab?#Try-it!)

Before starting, make sure you have the xpdf tool installed by running:

sudo apt-get install -y xpdf

Next, try using the command-line tool to convert the pdf into a text file. In the terminal, run:

pdftotext data/cats.pdf tmp/a.txt

cat tmp/a.txt

The first command will convert from pdf to a text file. The second line (which is confusingly [overloaded](https://en.wikipedia.org/wiki/Function_overloading) as cat) will concatenate the argument files and print on the standard output. In this case, we provide only the a.txt file so this file's content will be printed to the terminal window.

Finally, create your new PDFImporter class that performs the following steps:

1. Creates a random filename for the output.
2. Uses supbrocess to call the pdftotext tool on the input path, saving to the random file.
3. Uses the Python language reference to open the text file and read it [line-by-line](https://stackabuse.com/read-a-file-line-by-line-in-python/).
4. For each line, parses a new Cat object.
5. Removes the temporary text file.
6. Returns the list of cats.

**from** typing **import** List

**import** subprocess

**import** os

**import** random

**from** .ImportInterface **import** ImportInterface

**from** .Cat **import** Cat

**class** PDFImporter(ImportInterface):

allowed\_extensions **=** ['pdf']

@classmethod

**def** parse(cls, path: str) **->** List[Cat]:

**if** **not** cls.can\_ingest(path):

**raise** Exception('Cannot Ingest Exception')

tmp **=** f'./tmp/{random.randint(0,1000000)}.txt'

call **=** subprocess.call(['pdftotext', path, tmp])

file\_ref **=** open(tmp, "r")

cats **=** []

**for** line **in** file\_ref.readlines():

line **=** line.strip('\n\r').strip()

**if** len(line) **>** 0:

parsed **=** line.split(',')

new\_cat **=** Cat(parsed[0],

int(parsed[1]),

bool(parsed[2]))

cats.append(new\_cat)

file\_ref.close()

os.remove(tmp)

**return** cats

These skills allow us to package our applications into tools that other developers or users can use without needing to understand how we've implemented our code. For example, we may write a utility to convert a document from a PDF to plain text for a business team. Instead of asking the user to open a Python script, change variables, and then execute the script, we can use Argparse to request file paths and use subprocess internally to perform the conversion. In this case, we can simply train the user to execute a command like $ python3 convert.py file.pdf to perform what would otherwise be a complex task.