INTERMEDIATE
PYTHON

Muhammad Yasoob Ullah Khalid

Contents

Note: You can donate me for my hardwork if you want to by buying the donation version of Intermediate Python from Gumroad. Your help would be greatly appreciated!

You can also sign up to my mailing list so that you remain in sync with any major updates to this book or my future projects!

Part I
PREFACE

Python is an amazing language with a strong and friendly community of programmers. However, there is a lack of documentation on what to learn after getting the basics of Python down your throat. Through this book I aim to solve this problem. I would give you bits of information about some interesting topics which you can further explore.

The topics which are discussed in this book open up your mind towards some nice corners of Python language. This book is an outcome of my desire to have something like this when I was beginning to learn Python.

If you are a beginner, intermediate or even an advanced programmer there is something for you in this book.

Please note that this book is not a tutorial and does not teach you Python. The topics are not explained in depth, instead only the minimum required information is given.

I am sure you are as excited as I am so let's start!

Note: This book is a continuous work in progress. If you find anything which you can further improve (I know you will find a lot of stuff) then kindly submit a pull request!

Part II
AUTHOR

I am Muhammad Yasoob Ullah Khalid. I have been programming extensively in Python for over 3 years now. I have been involved in a lot of Open Source projects. I regularly blog about interesting Python topics over at my blog . In 2014 I also spoke at EuroPython which was held in Berlin. It is the biggest Python conference in Europe. If you have an interesting Internship opportunity for me then I would definitely like to hear from you!

Part III
TABLE OF CONTENTS

*args and **kwargs

I have come to see that most new python programmers have a hard time figuring out the *args and **kwargs magic variables. So what are they ? First of all let me tell you that it is not necessary to write *args or **kwargs. Only the * (asterisk) is necessary. You could have also written *var and **vars. Writing *args and **kwargs is just a convention. So now lets take a look at *args first.

1.1 Usage of *args

*args and **kwargs are mostly used in function definitions. *args and **kwargs allow you to pass a variable number of arguments to a function. What variable means here is that you do not know beforehand how many arguments can be passed to your function by the user so in this case you use these two keywords. *args is used to send a **non-keyworded** variable length argument list to the function. Here's an example to help you get a clear idea:

```python
def test_var_args(f_arg, *argv):
    print("first normal arg:", f_arg)
    for arg in argv:
        print("another arg through *argv:", arg)

test_var_args('yasoob', 'python', 'eggs', 'test')
```

This produces the following result:

```
first normal arg: yasoob
another arg through *argv: python
another arg through *argv: eggs
another arg through *argv: test
```

I hope this cleared away any confusion that you had. So now let's talk about
**kwargs

1.2 Usage of **kwargs

kwargs allows you to pass **keyworded variable length of arguments to a function. You should use **kwargs if you want to handle **named arguments** in a function. Here is an example to get you going with it:

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

>>> greet_me(name="yasoob")
name == yasoob
```

So you can see how we handled a keyworded argument list in our function.
This is just the basics of **kwargs and you can see how useful it is. Now let's
talk about how you can use *args and **kwargs to call a function with a list or
dictionary of arguments.

1.3 Using *args and **kwargs to call a function

So here we will see how to call a function using *args and **kwargs. Just consider that you have this little function:

```
def test_args_kwargs(arg1, arg2, arg3):
    print("arg1:", arg1)
    print("arg2:", arg2)
    print("arg3:", arg3)
```

Now you can use *args or **kwargs to pass arguments to this little function.
Here's how to do it:

```
# first with *args
>>> args = ("two", 3, 5)
>>> test_args_kwargs(*args)
arg1: two
arg2: 3
arg3: 5

# now with **kwargs:
>>> kwargs = {"arg3": 3, "arg2": "two", "arg1": 5}
>>> test_args_kwargs(**kwargs)
arg1: 5
arg2: two
arg3: 3
```

Order of using *args **kwargs and formal args

So if you want to use all three of these in functions then the order is

```
some_func(fargs, *args, **kwargs)
```

1.4 When to use them?

It really depends on what your requirements are. The most common use case is when making function decorators (discussed in another chapter). Moreover it can be used in monkey patching as well. Monkey patching means modifying some code at runtime. Consider that you have a class with a function called `get_info` which calls an API and returns the response data. If we want to test it we can replace the API call with some test data. For instance:

```python
import someclass

def get_info(self, *args):
    return "Test data"

someclass.get_info = get_info
```

I am sure that you can think of some other use cases as well.

Debugging

Debugging is also something which once mastered can greatly enhance your bug hunting skills. Most of the newcomers neglect the importance of the Python debugger (pdb). In this section I am going to tell you only a few important commands. You can learn more about it from the official documentation.

Running from commandline

You can run a script from the commandline using the Python debugger. Here is an example:

```
$ python -m pdb my_script.py
```

It would cause the debugger to stop the execution on the first statement it finds. This is helpful if your script is short. You can then inspect the variables and continue execution line-by-line.

Running from inside a script

You can set break points in the script itself so that you can inspect the variables and stuff at particular points. This is possible using the pdb.set_trace() method. Here is an example:

```python
import pdb

def make_bread():
    pdb.set_trace()
    return "I don't have time"

print(make_bread())
```

Try running the above script after saving it. You would enter the debugger as soon as you run it. Now it's time to learn some of the commands of the debugger.

Commands:

- c: continue execution

- w: shows the context of the current line it is executing.

- a: print the argument list of the current function

- s: Execute the current line and stop at the first possible occasion.

- n: Continue execution until the next line in the current function is reached or it returns.

The difference between next and step is that step stops inside a called function, while next executes called functions at (nearly) full speed, only stopping at the next line in the current function.

These are just a few commands. pdb also supports post mortem. It is also a really handy function. I would highly suggest you to look at the official documentation and learn more about it.

Generators

First lets understand iterators. According to Wikipedia, an iterator is an object that enables a programmer to traverse a container, particularly lists. However, an iterator performs traversal and gives access to data elements in a container, but does not perform iteration. You might be confused so lets take it a bit slow. There are three parts namely:

- Iterable
- Iterator
- Iteration

All of these parts are linked to each other. We will discuss them one by one and later talk about generators.

3.1 Iterable

An `iterable` is any object in Python which has an `__iter__` or a `__getitem__` method defined which returns an **iterator** or can take indexes (Both of these dunder methods are fully explained in a previous chapter). In short an `iterable` is any object which can provide us with an **iterator**. So what is an **iterator**?

3.2 Iterator

An iterator is any object in Python which has a next (Python2) or __next__ method defined. That's it. That's an iterator. Now let's understand **iteration**.

3.3 Iteration

In simple words it is the process of taking an item from something e.g a list. When we use a loop to loop over something it is called iteration. It is the name given to the process itself. Now as we have a basic understanding of these terms let's understand **generators**.

3.4 Generators

Generators are iterators, but you can only iterate over them once. It's because they do not store all the values in memory, they generate the values on the fly. You use them by iterating over them, either with a 'for' loop or by passing them to any function or construct that iterates. Most of the time generators are implemented as functions. However, they do not return a value, they yield it. Here is a simple example of a generator function:

```python
def generator_function():
    for i in range(10):
        yield i

for item in generator_function():
    print(item)

# Output: 0
# 1
# 2
# 3
# 4
# 5
# 6
# 7
# 8
# 9
```

It is not really useful in this case. Generators are best for calculating large sets of results (particularly calculations involving loops themselves) where you don't want to allocate the memory for all results at the same time. Many Standard Library functions that return `lists` in Python 2 have been modified to return `generators` in Python 3 because `generators` require fewer resources.

Here is an example `generator` which calculates fibonacci numbers:

```python
# generator version
def fibon(n):
    a = b = 1
    for i in range(n):
        yield a
        a, b = b, a + b
```

Now we can use it like this:

```python
for x in fibon(1000000):
    print(x)
```

This way we would not have to worry about it using a lot of resources. However, if we would have implemented it like this:

```python
def fibon(n):
    a = b = 1
    result = []
    for i in range(n):
        result.append(a)
        a, b = b, a + b
    return result
```

It would have used up all our resources while calculating a large input. We have discussed that we can iterate over `generators` only once but we haven't tested it. Before testing it you need to know about one more built-in function of Python, `next()`. It allows us to access the next element of a sequence. So let's test out our understanding:

```python
def generator_function():
    for i in range(3):
        yield i

gen = generator_function()
print(next(gen))
# Output: 0
```

```
print(next(gen))
# Output: 1
print(next(gen))
# Output: 2
print(next(gen))
# Output: Traceback (most recent call last):
#           File "<stdin>", line 1, in <module>
#         StopIteration
```

As we can see that after yielding all the values `next()` caused a `StopIteration` error. Basically this error informs us that all the values have been yielded. You might be wondering that why don't we get this error while using a `for` loop? Well the answer is simple. The `for` loop automatically catches this error and stops calling `next`. Do you know that a few built-in data types in Python also support iteration? Let's check it out:

```
my_string = "Yasoob"
next(my_string)
# Output: Traceback (most recent call last):
#     File "<stdin>", line 1, in <module>
#   TypeError: str object is not an iterator
```

Well that's not what we expected. The error says that `str` is not an iterator. Well it is right! It is an iterable but not an iterator. This means that it supports iteration but we can not directly iterate over it. How can we then iterate over it? It's time to learn about one more built-in function, `iter`. It returns an `iterator` object from an iterable. Here is how we can use it:

```
my_string = "Yasoob"
my_iter = iter(my_string)
next(my_iter)
# Output: 'Y'
```

Now that is much better. I am sure that you loved learning about generators. Do bear it in mind that you can fully grasp this concept only when you use it. Make sure that you follow this pattern and use `generators` whenever they make sense to you. You won't be disappointed!

Map & Filter

These are two functions which facilitate a functional approach to programming. We will discuss them one by one and understand their use cases.

4.1 Map

Map applies a function to all the items in an input_list. Here is the blueprint:

Blueprint

```python
map(function_to_apply, list_of_inputs)
```

Most of the times we want to pass all the list elements to a function one-by-one and then collect the output. For instance:

```python
items = [1, 2, 3, 4, 5]
squared = []
for i in items:
    squared.append(i**2)
```

Map allows us to implement this in a much simpler and nicer way. Here you go:

```python
items = [1, 2, 3, 4, 5]
squared = list(map(lambda x: x**2, items))
```

Most of the times we use lambdas with map so I did the same. Instead of a list of inputs we can even have a list of functions!

```python
def multiply(x):
        return (x*x)
def add(x):
        return (x+x)

funcs = [multiply, add]
for i in range(5):
    value = list(map(lambda x: x(i), funcs))
    print(value)

# Output:
# [0, 0]
# [1, 2]
# [4, 4]
# [9, 6]
# [16, 8]
```

4.2 Filter

As the name suggests, filter creates a list of elements for which a function returns true. Here is a short and concise example:

```python
number_list = range(-5, 5)
less_than_zero = list(filter(lambda x: x < 0, number_list))
print(less_than_zero)

# Output: [-5, -4, -3, -2, -1]
```

The filter resembles a for loop but it is a builtin function and faster.

Note: If map & filter do not appear beautiful to you then you can read about list/dict/tuple comprehensions.

set Data Structure

set is a really useful data structure. sets behave mostly like lists with the distinction that they can not contain duplicate values. It is really useful in a lot of cases. For instance you might want to check whether there are duplicates in a list or not. You have two options. The first one involves using a for loop. Something like this:

```python
some_list = ['a', 'b', 'c', 'b', 'd', 'm', 'n', 'n']

duplicates = []
for value in some_list:
    if some_list.count(value) > 1:
        if value not in duplicates:
            duplicates.append(value)

print(duplicates)
# Output: ['b', 'n']
```

But there is a simpler and more elegant solution involving sets. You can simply do something like this:

```python
some_list = ['a', 'b', 'c', 'b', 'd', 'm', 'n', 'n']
duplicates = set([x for x in some_list if some_list.count(x) > 1])
print(duplicates)
# Output: set(['b', 'n'])
```

Sets also have a few other methods. Below are some of them.

Intersection

You can intersect two sets. For instance:

```
valid = set(['yellow', 'red', 'blue', 'green', 'black'])
input_set = set(['red', 'brown'])
print(input_set.intersection(valid))
# Output: set(['red'])
```

Difference

You can find the invalid values in the above example using the difference method. For example:

```
valid = set(['yellow', 'red', 'blue', 'green', 'black'])
input_set = set(['red', 'brown'])
print(input_set.difference(valid))
# Output: set(['brown'])
```

You can also create sets using the new notation:

```
a_set = {'red', 'blue', 'green'}
print(type(a_set))
# Output: <type 'set'>
```

There are a few other methods as well. I would recommend visiting the official documentation and giving it a quick read.

Ternary Operators

Ternary operators are more commonly known as conditional expressions in Python. These operators evaluate something based on a condition being true or not. They became a part of Python in version 2.4

Here is a blueprint and an example of using these conditional expressions.

Blueprint:

```python
condition_is_true if condition else condition_is_false
```

Example:

```python
is_fat = True
state = "fat" if is_fat else "not fat"
```

It allows to quickly test a condition instead of a multiline if statement. Often times it can be immensely helpful and can make your code compact but still maintainable.

Another more obscure and not widely used example involves tuples. Here is some sample code:

Blueprint:

```python
(if_test_is_false, if_test_is_true)[test]
```

Example:

```python
fat = True
fitness = ("skinny", "fat")[fat]
```

```python
print("Ali is ", fitness)
# Output: Ali is fat
```

This works simply because True == 1 and False == 0, and so can be done with lists in addition to tuples.

The above example is not widely used and is generally disliked by Pythonistas for not being Pythonic. It is also easy to confuse where to put the true value and where to put the false value in the tuple.

Another reason to avoid using a tupled ternery is that it results in both elements of the tuple being evaluated, whereas the if-else ternary operator does not.

Example:

```python
condition = True
print(2 if condition else 1/0)
#Output is 2

print((1/0, 2)[condition])
#ZeroDivisionError is raised
```

This happens because with the tupled ternary technique, the tuple is first built, then an index is found. For the if-else ternary operator, it follows the normal if-else logic tree. Thus, if one case could raise an exception based on the condition, or if either case is a computation-heavy method, using tuples is best avoided.

Decorators

Decorators are a significant part of Python. In simple words: they are functions which modify the functionality of another function. They help to make our code shorter and more Pythonic. Most of the beginners do not know where to use them so I am going to share some areas where decorators can make your code more concise.

First, let's discuss how to write your own decorator.

It is perhaps one of the most difficult concepts to grasp. We will take it one step at a time so that you can fully understand it.

7.1 Everything in python is an object:

First of all let's understand functions in python:

```python
def hi(name="yasoob"):
    return "hi " + name

print(hi())
# output: 'hi yasoob'

# We can even assign a function to a variable like
greet = hi
# We are not using parentheses here because we are not calling the function hi
# instead we are just putting it into the greet variable. Let's try to run this
```

```python
print(greet())
# output: 'hi yasoob'

# Let's see what happens if we delete the old hi function!
del hi
print(hi())
#outputs: NameError

print(greet())
#outputs: 'hi yasoob'
```

7.2 Defining functions within functions:

So those are the basics when it comes to functions. Let's take your knowledge one step further. In Python we can define functions inside other functions:

```python
def hi(name="yasoob"):
    print("now you are inside the hi() function")

    def greet():
        return "now you are in the greet() function"

    def welcome():
        return "now you are in the welcome() function"

    print(greet())
    print(welcome())
    print("now you are back in the hi() function")

hi()
#output:now you are inside the hi() function
#       now you are in the greet() function
#       now you are in the welcome() function
#       now you are back in the hi() function

# This shows that whenever you call hi(), greet() and welcome()
# are also called. However the greet() and welcome() functions
# are not available outside the hi() function e.g:

greet()
```

```
#outputs: NameError: name 'greet' is not defined
```

So now we know that we can define functions in other functions. In other words: we can make nested functions. Now you need to learn one more thing, that functions can return functions too.

7.3 Returning functions from within functions:

It is not necessary to execute a function within another function, we can return it as an output as well:

```python
def hi(name="yasoob"):
    def greet():
        return "now you are in the greet() function"

    def welcome():
        return "now you are in the welcome() function"

    if name == "yasoob":
        return greet
    else:
        return welcome

a = hi()
print(a)
#outputs: <function greet at 0x7f2143c01500>

#This clearly shows that `a` now points to the greet() function in hi()
#Now try this

print(a())
#outputs: now you are in the greet() function
```

Just take a look at the code again. In the if/else clause we are returning greet and welcome, not greet() and welcome(). Why is that? It's because when you put a pair of parentheses after it, the function gets executed; whereas if you don't put parenthesis after it, then it can be passed around and can be assigned to other variables without executing it. Did you get it? Let me explain it in a little bit more detail. When we write a = hi(), hi() gets executed and because the name is yasoob by default, the function greet is returned. If we change the

statement to a = hi(name = "ali") then the welcome function will be returned. We can also do print hi()() which outputs *now you are in the greet() function.*

7.4 Giving a function as an argument to another function:

```python
def hi():
    return "hi yasoob!"

def doSomethingBeforeHi(func):
    print("I am doing some boring work before executing hi()")
    print(func())

doSomethingBeforeHi(hi)
#outputs:I am doing some boring work before executing hi()
#        hi yasoob!
```

Now you have all the required knowledge to learn what decorators really are. Decorators let you execute code before and after a function.

7.5 Writing your first decorator:

In the last example we actually made a decorator! Let's modify the previous decorator and make a little bit more usable program:

```python
def a_new_decorator(a_func):

    def wrapTheFunction():
        print("I am doing some boring work before executing a_func()")

        a_func()

        print("I am doing some boring work after executing a_func()")

    return wrapTheFunction

def a_function_requiring_decoration():
    print("I am the function which needs some decoration to remove my foul smell")
```

```
a_function_requiring_decoration()
#outputs: "I am the function which needs some decoration to remove my foul smell"

a_function_requiring_decoration = a_new_decorator(a_function_requiring_decoration)
#now a_function_requiring_decoration is wrapped by wrapTheFunction()

a_function_requiring_decoration()
#outputs:I am doing some boring work before executing a_function_requiring_decoration()
#        I am the function which needs some decoration to remove my foul smell
#        I am doing some boring work after executing a_function_requiring_decoration()
```

Did you get it? We just applied the previously learned principles. This is exactly what the decorators do in python! They wrap a function and modify its behaviour in one way or the another. Now you might be wondering that we did not use the @ anywhere in our code? That is just a short way of making up a decorated function. Here is how we could have run the previous code sample using @.

```
@a_new_decorator
def a_function_requiring_decoration():
    """Hey you! Decorate me!"""
    print("I am the function which needs some decoration to "
          "remove my foul smell")

a_function_requiring_decoration()
#outputs: I am doing some boring work before executing a_func()
#        I am the function which needs some decoration to remove my foul smell
#        I am doing some boring work after executing a_func()

#the @a_new_decorator is just a short way of saying:
a_function_requiring_decoration = a_new_decorator(a_function_requiring_decoration)
```

I hope you now have a basic understanding of how decorators work in Python. Now there is one problem with our code. If we run:

```
print(a_function_requiring_decoration.__name__)
# Output: wrapTheFunction
```

That's not what we expected! Its name is "a_function_requiring_decoration". Well our function was replaced by wrapTheFunction. It overrode the name and docstring of our function. Luckily Python provides us a simple function to solve this problem and that is functools.wraps. Let's modify our previous example to use functools.wraps:

```python
from functools import wraps

def a_new_decorator(a_func):
    @wraps(a_func)
    def wrapTheFunction():
        print("I am doing some boring work before executing a_func()")
        a_func()
        print("I am doing some boring work after executing a_func()")
    return wrapTheFunction

@a_new_decorator
def a_function_requiring_decoration():
    """Hey yo! Decorate me!"""
    print("I am the function which needs some decoration to "
          "remove my foul smell")

print(a_function_requiring_decoration.__name__)
# Output: a_function_requiring_decoration
```

Now that is much better. Let's move on and learn some use-cases of decorators.

Blueprint:

```python
from functools import wraps
def decorator_name(f):
    @wraps(f)
    def decorated(*args, **kwargs):
        if not can_run:
            return "Function will not run"
        return f(*args, **kwargs)
    return decorated

@decorator_name
def func():
    return("Function is running")

can_run = True
print(func())
# Output: Function is running

can_run = False
print(func())
# Output: Function will not run
```

Note: @wraps takes a function to be decorated and adds the functionality of copying over the function name, docstring, arguments list, etc. This allows to access the pre-decorated function's properties in the decorator.

7.5.1 Use-cases:

Now let's take a look at the areas where decorators really shine and their usage makes something really easy to manage.

7.5.2 Authorization

Decorators can help to check whether someone is authorized to use an endpoint in a web application. They are extensively used in Flask web framework and Django. Here is an example to employ decorator based authentication:

Example :

```python
from functools import wraps


def requires_auth(f):
    @wraps(f)
    def decorated(*args, **kwargs):
        auth = request.authorization
        if not auth or not check_auth(auth.username, auth.password):
            authenticate()
        return f(*args, **kwargs)
    return decorated
```

7.5.3 Logging

Logging is another area where the decorators shine. Here is an example:

```python
from functools import wraps


def logit(func):
    @wraps(func)
    def with_logging(*args, **kwargs):
        print(func.__name__ + " was called")
        return func(*args, **kwargs)
    return with_logging
```

```python
@logit
def addition_func(x):
   """Do some math."""
   return x + x


result = addition_func(4)
# Output: addition_func was called
```

I am sure you are already thinking about some clever uses of decorators.

Global & Return

You might have encountered some functions written in python which have a `return` keyword in the end of the function. Do you know what it does? It is similar to return in other languages. Lets examine this little function:

```python
def add(value1, value2):
    return value1 + value2


result = add(3, 5)
print(result)
# Output: 8
```

The function above takes two values as input and then output their addition. We could have also done:

```python
def add(value1,value2):
    global result
    result = value1 + value2


add(3,5)
print(result)
# Output: 8
```

So first lets talk about the first bit of code which involves the `return` keyword. What that function is doing is that it is assigning the value to the variable which is calling that function which in our case is `result`. In most cases and you won't need to use the `global` keyword. However lets examine the other bit of code as well which includes the `global` keyword. So what that function is doing is

that it is making a global variable `result`. What does global mean here? Global variable means that we can access that variable outside the scope of the function as well. Let me demonstrate it with an example :

```
# first without the global variable
def add(value1, value2):
    result = value1 + value2


add(2, 4)
print(result)


# Oh crap we encountered an exception. Why is it so ?
# the python interpreter is telling us that we do not
# have any variable with the name of result. It is so
# because the result variable is only accessible inside
# the function in which it is created if it is not global.
Traceback (most recent call last):
  File "", line 1, in
    result
NameError: name 'result' is not defined


# Now lets run the same code but after making the result
# variable global
def add(value1, value2):
    global result
    result = value1 + value2


add(2, 4)
result
6
```

So hopefully there are no errors in the second run as expected. In practical programming you should try to stay away from `global` keyword as it only makes life difficult by introducing unwanted variables to the global scope.

8.1 Multiple return values

So what if you want to return two variables from a function instead of one? There are a couple of approaches which new programmers take. The most famous approach is to use `global` keyword. Let's take a look at a useless example:

```python
def profile():
    global name
    global age
    name = "Danny"
    age = 30

profile()
print(name)
# Output: Danny

print(age)
# Output: 30
```

Note: Don't try to use the above mentioned method. I repeat, don't try to use the above mentioned method!

Some try to solve this problem by *returning* a tuple, `list` or `dict` with the reqired values after the function terminates. It is one way to do it and works like a charm:

```python
def profile():
    name = "Danny"
    age = 30
    return (name, age)

profile_data = profile()
print(profile_data[0])
# Output: Danny

print(profile_data[1])
# Output: 30
```

But what many programmers don't know is that you can return two separate values as well. Let's take a look at an example so that you can better understand it:

```python
def profile():
    name = "Danny"
    age = 30
    return name, age

name, age = profile()
print(name)
```

```
# Output: Danny
```

```
print(age)
# Output: 30
```

This is a better way to do it along with returning `tuples`, `lists` and `dicts`. Don't use `global` keyword unless you know what you are doing. `global` might be a better option in a few cases but is not in most of them.

Mutation

The mutable and immutable datatypes in Python cause a lot of headache for new programmers. In simple words, mutable means 'able to be changed' and immutable means 'constant'. Want your head to spin? Consider this example:

```python
foo = ['hi']
print(foo)
# Output: ['hi']

bar = foo
bar += ['bye']
print(foo)
# Output: ['hi', 'bye']
```

What just happened? We were not expecting that! We were expecting something like this:

```python
foo = ['hi']
print(foo)
# Output: ['hi']

bar = foo
bar += ['bye']

print(foo)
# Output: ['hi']

print(bar)
```

```
# Output: ['hi', 'bye']
```

It's not a bug. It's mutability in action. Whenever you assign a variable to another variable of mutable datatype, any changes to the data are reflected by both variables. The new variable is just an alias for the old variable. This is only true for mutable datatypes. Here is a gotcha involving functions and mutable data types:

```python
def add_to(num, target=[]):
    target.append(num)
    return target

add_to(1)
# Output: [1]

add_to(2)
# Output: [1, 2]

add_to(3)
# Output: [1, 2, 3]
```

You might have expected it to behave differently. You might be expecting that a fresh list would be created when you call add_to like this:

```python
def add_to(num, target=[]):
    target.append(num)
    return target

add_to(1)
# Output: [1]

add_to(2)
# Output: [2]

add_to(3)
# Output: [3]
```

Well again it is the mutability of lists which causes this pain. In Python the default arguments are evaluated once when the function is defined, not each time the function is called. You should never define default arguments of mutable type unless you know what you are doing. You should do something like this:

```python
def add_to(element, target=None):
    if target is None:
        target = []
    target.append(element)
    return target
```

Now whenever you call the function without the `target` argument, a new list is created. For instance:

```python
add_to(42)
# Output: [42]

add_to(42)
# Output: [42]

add_to(42)
# Output: [42]
```

__slots__ Magic

In Python every class can have instance attributes. By default Python uses a dict to store an object's instance attributes. This is really helpful as it allows setting arbitrary new attributes at runtime.

However, for small classes with known attributes it might be a bottleneck. The dict wastes a lot of RAM. Python can't just allocate a static amount of memory at object creation to store all the attributes. Therefore it sucks a lot of RAM if you create a lot of objects (I am talking in thousands and millions). Still there is a way to circumvent this issue. It involves the usage of `__slots__` to tell Python not to use a dict, and only allocate space for a fixed set of attributes. Here is an example with and without __slots__:

Without `__slots__`:

```python
class MyClass(object):
    def __init__(name, identifier):
        self.name = name
        self.identifier = identifier
        self.set_up()
    # ...
```

With `__slots__`:

```python
class MyClass(object):
    __slots__ = ['name', 'identifier']
    def __init__(name, identifier):
        self.name = name
        self.identifier = identifier
```

```
        self.set_up()
    # ...
```

The second piece of code will reduce the burden on your RAM. Some people have seen almost 40 to 50% reduction in RAM usage by using this technique.

On a sidenote, you might want to give PyPy a try. It does all of these optimizations by default.

Virtual Environment

Have you ever heard of `virtualenv`? If you are a beginner, then you might not have heard about it but if you are a seasoned programmer then it may well be a vital part of your toolset.

So what is `virtualenv`? `Virtualenv` is a tool which allows us to make isolated python environments. Imagine you have an application that needs version 2 of a library, but another application requires version 3. How can you use and develop both these applications?

If you install everything into `/usr/lib/python2.7/site-packages` (or whatever your platform's standard location is), it's easy to end up in a situation where you unintentionally upgrade a package.

In another case, imagine that you have an application which is fully developed and you do not want to make any change to the libraries it is using but at the same time you start developing another application which requires the updated versions of those libraries.

What will you do? Use `virtualenv`! It creates isolated environments for your python application and allows you to install Python libraries in that isolated environment instead of installing them globally.

To install it, just type this command in the shell:

```
$ pip install virtualenv
```

The most important commands are:

- `$ virtualenv myproject`

- `$ source bin/activate`

This first one makes an isolated virtualenv environment in the `myproject` folder and the second command activates that isolated environment.

While creating the virtualenv you have to make a decision. Do you want this virtualenv to use packages from your system `site-packages` or install them in the virtualenv's site-packages? By default, virtualenv will not give access to the global `site-packages`.

If you want your `virtualenv` to have access to your systems `site-packages`, use the `--system-site-packages` switch when creating your virtualenv like this:

```
$ virtualenv --system-site-packages mycoolproject
```

You can turn off the `env` by typing:

```
$ deactivate
```

Running *python* after deactivating will use your system installation of Python again.

Bonus

You can use `smartcd` which is a library for bash and zsh and allows you to alter your bash (or zsh) environment as you cd. It can be really helpful to activate and deactivate a `virtualenv` when you change directories. I have used it quite a lot and love it. You can read more about it on GitHub

This was just a short intro to virtualenv. There's a lot more to it; this link has more information.

Collections

Python ships with a module that contains a number of container data types called Collections. We will talk about a few of them and discuss their usefulness.

The ones which we will talk about are:

- `defaultdict`

- `counter`

- `deque`

- `namedtuple`

- `enum.Enum` (outside of the module; Python 3.4+)

12.1 `defaultdict`

I personally use defaultdict quite a bit. Unlike `dict`, with `defaultdict` you do not need to check whether a key is present or not. So we can do:

```python
from collections import defaultdict

colours = (
    ('Yasoob', 'Yellow'),
    ('Ali', 'Blue'),
    ('Arham', 'Green'),
    ('Ali', 'Black'),
```

```
    ('Yasoob', 'Red'),
    ('Ahmed', 'Silver'),
)

favourite_colours = defaultdict(list)

for name, colour in colours:
    favourite_colours[name].append(colour)

print(favourite_colours)

# output
# defaultdict(<type 'list'>,
#    {'Arham': ['Green'],
#     'Yasoob': ['Yellow', 'Red'],
#     'Ahmed': ['Silver'],
#     'Ali': ['Blue', 'Black']
# })
```

One other very important use case is when you are appending to nested lists inside a dictionary. If a key is not already present in the dictionary then you are greeted with a KeyError. defaultdict allows us to circumvent this issue in a clever way. First let me share an example using dict which raises KeyError and then I will share a solution using defaultdict.

Problem:

```
some_dict = {}
some_dict['colours']['favourite'] = "yellow"
# Raises KeyError: 'colours'
```

Solution:

```
import collections
tree = lambda: collections.defaultdict(tree)
some_dict = tree()
some_dict['colours']['favourite'] = "yellow"
# Works fine
```

You can print some_dict using json.dumps. Here is some sample code:

```
import json
print(json.dumps(some_dict))
# Output: {"colours": {"favourite": "yellow"}}
```

12.2 counter

Counter allows us to count the occurrences of a particular item. For instance it can be used to count the number of individual favourite colours:

```python
from collections import Counter

colours = (
    ('Yasoob', 'Yellow'),
    ('Ali', 'Blue'),
    ('Arham', 'Green'),
    ('Ali', 'Black'),
    ('Yasoob', 'Red'),
    ('Ahmed', 'Silver'),
)

favs = Counter(name for name, colour in colours)
print(favs)
# Output: Counter({
#     'Yasoob': 2,
#     'Ali': 2,
#     'Arham': 1,
#     'Ahmed': 1
# })
```

We can also count the most common lines in a file using it. For example:

```python
with open('filename', 'rb') as f:
    line_count = Counter(f)
print(line_count)
```

12.3 deque

deque provides you with a double ended queue which means that you can append and delete elements from either side of the queue. First of all you have to import the deque module from the collections library:

```python
from collections import deque
```

Now we can instantiate a deque object.

```
d = deque()
```

It works like python lists and provides you with somewhat similar methods as well. For example you can do:

```
d = deque()
d.append('1')
d.append('2')
d.append('3')

print(len(d))
# Output: 3

print(d[0])
# Output: '1'

print(d[-1])
# Output: '3'
```

You can pop values from both sides of the deque:

```
d = deque(range(5))
print(len(d))
# Output: 5

d.popleft()
# Output: 0

d.pop()
# Output: 4

print(d)
# Output: deque([1, 2, 3])
```

We can also limit the amount of items a deque can hold. By doing this when we achieve the maximum limit of our deque it will simply pop out the items from the opposite end. It is better to explain it using an example so here you go:

```
d = deque(maxlen=30)
```

Now whenever you insert values after 30, the leftmost value will be popped from the list. You can also expand the list in any direction with new values:

```
d = deque([1,2,3,4,5])
d.extendleft([0])
d.extend([6,7,8])
print(d)
# Output: deque([0, 1, 2, 3, 4, 5, 6, 7, 8])
```

This was just a quick drive through the `collections` module. Make sure you read the official documentation after reading this.

12.4 `namedtuple`

You might already be acquainted with tuples. A tuple is a lightweight object type which allows to store a sequence of immutable Python objects. They are just like lists but have a few key differences. The major one is that unlike lists, **you can not change a value in a tuple**. In order to access the value in a tuple you use integer indexes like:

```
man = ('Ali', 30)
print(man[0])
# Output: Ali
```

Well, so now what are `namedtuples`? They turn tuples into convenient containers for simple tasks. With namedtuples you don't have to use integer indexes for accessing members of a tuple. You can think of namedtuples like dictionaries but unlike dictionaries they are immutable.

```
from collections import namedtuple

Animal = namedtuple('Animal', 'name age type')
perry = Animal(name="perry", age=31, type="cat")

print(perry)
# Output: Animal(name='perry', age=31, type='cat')

print(perry.name)
# Output: 'perry'
```

You can now see that we can access members of a tuple just by their name using a `.`. Let's dissect it a little more. A named tuple has two required arguments. They are the tuple name and the tuple field_names. In the above example our tuple name was 'Animal' and the tuple field_names were 'name',

'age' and 'cat'. Namedtuple makes your tuples **self-document**. You can easily understand what is going on by having a quick glance at your code. And as you are not bound to use integer indexes to access members of a tuple, it makes it more easy to maintain your code. Moreover, as **'namedtuple' instances do not have per-instance dictionaries**, they are lightweight and require no more memory than regular tuples. This makes them faster than dictionaries. However, do remember that as with tuples, **attributes in namedtuples are immutable**. It means that this would not work:

```python
from collections import namedtuple

Animal = namedtuple('Animal', 'name age type')
perry = Animal(name="perry", age=31, type="cat")
perry.age = 42

# Output: Traceback (most recent call last):
#           File "", line 1, in
#           AttributeError: can't set attribute
```

You should use named tuples to make your code self-documenting. **They are backwards compatible with normal tuples**. It means that you can use integer indexes with namedtuples as well:

```python
from collections import namedtuple

Animal = namedtuple('Animal', 'name age type')
perry = Animal(name="perry", age=31, type="cat")
print(perry[0])
# Output: perry
```

Last but not the least, you can convert a namedtuple to a dictionary. Like this:

```python
from collections import namedtuple

Animal = namedtuple('Animal', 'name age type')
perry = Animal(name="Perry", age=31, type="cat")
print(perry._asdict())
# Output: OrderedDict([('name', 'Perry'), ('age', 31), ...
```

12.5 enum.Enum (Python 3.4+)

Another useful collection is the enum object. It is available in the enum module, in Python 3.4 and up (also available as a backport in PyPI named enum34.) Enums (enumerated type) are basically a way to organize various things.

Let's consider the Animal namedtuple from the last example. It had a type field. The problem is, the type was a string. This poses some problems for us. What if the user types in Cat because they held the Shift key? Or CAT? Or kitten?

Enumerations can help us avoid this problem, by not using strings. Consider this example:

```
from collections import namedtuple
from enum import Enum

class Species(Enum):
    cat = 1
    dog = 2
    horse = 3
    aardvark = 4
    butterfly = 5
    owl = 6
    platypus = 7
    dragon = 8
    unicorn = 9
    # The list goes on and on...

    # But we don't really care about age, so we can use an alias.
    kitten = 1
    puppy = 2

Animal = namedtuple('Animal', 'name age type')
perry = Animal(name="Perry", age=31, type=Species.cat)
drogon = Animal(name="Drogon", age=4, type=Species.dragon)
tom = Animal(name="Tom", age=75, type=Species.cat)
charlie = Animal(name="Charlie", age=2, type=Species.kitten)

# And now, some tests.
>>> charlie.type == tom.type
True
>>> charlie.type
<Species.cat: 1>
```

This is much less error-prone. We have to be specific, and we should use only the enumeration to name types.

There are three ways to access enumeration members. For example, all three methods will get you the value for cat:

```
Species(1)
Species['cat']
Species.cat
```

Enumerate

Enumerate is a built-in function of Python. It's usefulness can not be summarized in a single line. Yet most of the newcomers and even some advanced programmers are unaware of it. It allows us to loop over something and have an automatic counter. Here is an example:

```python
for counter, value in enumerate(some_list):
    print(counter, value)
```

This is not it. enumerate also accepts some optional arguments which make it even more useful.

```python
my_list = ['apple', 'banana', 'grapes', 'pear']
for c, value in enumerate(my_list, 1):
    print(c, value)

# Output:
# 1 apple
# 2 banana
# 3 grapes
# 4 pear
```

The optional argument allows us to tell enumerate from where to start the index. You can also create tuples containing the index and list item using a list. Here is an example:

```python
my_list = ['apple', 'banana', 'grapes', 'pear']
counter_list = list(enumerate(my_list, 1))
```

```
print(counter_list)
# Output: [(1, 'apple'), (2, 'banana'), (3, 'grapes'), (4, 'pear')]
```

Object introspection

In computer programming, introspection is the ability to determine the type of an object at runtime. It is one of Python's strengths. Everything in Python is an object and we can examine those objects. Python ships with a few built-in functions and modules to help us.

14.1 dir

In this section we will learn about `dir` and how it facilitates us in introspection.

It is one of the most important functions for introspection. It returns a list of attributes and methods belonging to an object. Here is an example:

```python
my_list = [1, 2, 3]
dir(my_list)
# Output: ['__add__', '__class__', '__contains__', '__delattr__', '__delitem__',
# '__delslice__', '__doc__', '__eq__', '__format__', '__ge__', '__getattribute__',
# '__getitem__', '__getslice__', '__gt__', '__hash__', '__iadd__', '__imul__',
# '__init__', '__iter__', '__le__', '__len__', '__lt__', '__mul__', '__ne__',
# '__new__', '__reduce__', '__reduce_ex__', '__repr__', '__reversed__', '__rmul__',
# '__setattr__', '__setitem__', '__setslice__', '__sizeof__', '__str__',
# '__subclasshook__', 'append', 'count', 'extend', 'index', 'insert', 'pop',
# 'remove', 'reverse', 'sort']
```

Our introspection gave us the names of all the methods of a list. This can be handy when you are not able to recall a method name. If we run `dir()` without any argument then it returns all names in the current scope.

14.2 `type` and `id`

The type function returns the type of an object. For example:

```
print(type(''))
# Output: <type 'str'>

print(type([]))
# Output: <type 'list'>

print(type({}))
# Output: <type 'dict'>

print(type(dict))
# Output: <type 'type'>

print(type(3))
# Output: <type 'int'>
```

`id` returns the unique ids of various objects. For instance:

```
name = "Yasoob"
print(id(name))
# Output: 139972439030304
```

14.3 inspect module

The inspect module also provides several useful functions to get information about live objects. For example you can check the members of an object by running:

```
import inspect
print(inspect.getmembers(str))
# Output: [('__add__', <slot wrapper '__add__' of ... ...
```

There are a couple of other methods as well which help in introspection. You can explore them if you wish.

Comprehensions

Comprehensions are a feature of Python which I would really miss if I ever have to leave it. Comprehensions are constructs that allow sequences to be built from other sequences. Three types of comprehensions are supported in both Python 2 and Python 3:

- list comprehensions
- dictionary comprehensions
- set comprehensions

We will discuss them one by one. Once you get the hang of using `list` comprehensions then you can use any of them easily.

15.1 `list` comprehensions

List comprehensions provide a short and concise way to create lists. It consists of square brackets containing an expression followed by a `for` clause, then zero or more `for` or `if` clauses. The expressions can be anything, meaning you can put in all kinds of objects in lists. The result would be a new list made after the evaluation of the expression in context of the `if` and `for` clauses.

Blueprint

```python
variable = [out_exp for out_exp in input_list if out_exp == 2]
```

Here is a short example:

```python
multiples = [i for i in range(30) if i % 3 is 0]
print(multiples)
# Output: [0, 3, 6, 9, 12, 15, 18, 21, 24, 27]
```

This can be really useful to make lists quickly. It is even preferred by some instead of the `filter` function. list comprehensions really shine when you want to supply a list to a method or function to make a new list by appending to it in each iteration of the `for` loop. For instance you would usually do something like this:

```python
squared = []
for x in range(10):
    squared.append(x**2)
```

You can simplify it using list comprehensions. For example:

```python
squared = [x**2 for x in range(10)]
```

15.2 `dict` comprehensions

They are used in a similar way. Here is an example which I found recently:

```python
mcase = {'a': 10, 'b': 34, 'A': 7, 'Z': 3}

mcase_frequency = {
    k.lower(): mcase.get(k.lower(), 0) + mcase.get(k.upper(), 0)
    for k in mcase.keys()
}

# mcase_frequency == {'a': 17, 'z': 3, 'b': 34}
```

In the above example we are combining the values of keys which are same but in different typecase. I personally do not use `dict` comprehensions a lot. You can also quickly switch keys and values of a dictionary:

```python
{v: k for k, v in some_dict.items()}
```

15.3 set comprehensions

They are also similar to list comprehensions. The only difference is that they use braces {}. Here is an example:

```python
squared = {x**2 for x in [1, 1, 2]}
print(squared)
# Output: {1, 4}
```

Exceptions

Exception handling is an art which once you master grants you immense powers. I am going to show you some of the ways in which we can handle exceptions.

In basic terminology we are aware of `try`/`except` clause. The code which can cause an exception to occur is put in the `try` block and the handling of the exception is implemented in the `except` block. Here is a simple example:

```python
try:
    file = open('test.txt', 'rb')
except IOError as e:
    print('An IOError occurred. {}'.format(e.args[-1]))
```

In the above example we are handling only the IOError exception. What most beginners do not know is that we can handle multiple exceptions.

16.1 Handling multiple exceptions:

We can use three methods to handle multiple exceptions. The first one involves putting all the exceptions which are likely to occur in a tuple. Like so:

```python
try:
    file = open('test.txt', 'rb')
except (IOError, EOFError) as e:
    print("An error occurred. {}".format(e.args[-1]))
```

Another method is to handle individual exceptions in separate except blocks. We can have as many except blocks as we want. Here is an example:

```python
try:
    file = open('test.txt', 'rb')
except EOFError as e:
    print("An EOF error occurred.")
    raise e
except IOError as e:
    print("An error occurred.")
    raise e
```

This way if the exception is not handled by the first except block then it may be handled by a following block, or none at all. Now the last method involves trapping ALL exceptions:

```python
try:
    file = open('test.txt', 'rb')
except Exception:
    # Some logging if you want
    raise
```

This can be helpful when you have no idea about the exceptions which may be thrown by your program.

16.1.1 finally clause

We wrap our main code in the try clause. After that we wrap some code in an except clause which gets executed if an exception occurs in the code wrapped in the try clause. In this example we will use a third clause as well which is the finally clause. The code which is wrapped in the finally clause will run whether or not an exception occurred. It might be used to perform clean-up after a script. Here is a simple example:

```python
try:
    file = open('test.txt', 'rb')
except IOError as e:
    print('An IOError occurred. {}'.format(e.args[-1]))
finally:
    print("This would be printed whether or not an exception occurred!")

# Output: An IOError occurred. No such file or directory
# This would be printed whether or not an exception occurred!
```

16.1.2 `try/else` clause

Often times we might want some code to run if **no** exception occurs. This can easily be achieved by using an `else` clause. Most people don't use it and honestly I have myself not used it widely. Here is an example:

```python
try:
    print('I am sure no exception is going to occur!')
except Exception:
    print('exception')
else:
    print('This would only run if no exception occurs.')
finally:
    print('This would be printed in every case.')

# Output: I am sure no exception is going to occur!
# This would only run if no exception occurs.
# This would be printed in every case.
```

The `else` clause would only run if no exception occurs and it would run before the `finally` clause.

Lambdas

Lambdas are one line functions. They are also known as anonymous functions in some other languages. You might want to use lambdas when you don't want to use a function twice in a program. They are just like normal functions and even behave like them.

Blueprint

```python
lambda argument: manipulate(argument)
```

Example

```python
add = lambda x, y: x + y

print(add(3, 5))
# Output: 8
```

Here are a few useful use cases for lambdas and just a few way in which they are used in the wild:

List sorting

```python
a = [(1, 2), (4, 1), (9, 10), (13, -3)]
a.sort(key=lambda x: x[1])

print(a)
# Output: [(13, -3), (4, 1), (1, 2), (9, 10)]
```

Parallel sorting of lists

```
data = zip(list1, list2)
data.sort()
list1, list2 = map(lambda t: list(t), zip(*data))
```

One-Liners

In this chapter I will show you some one-liner Python commands which can be really helpful.

Simple Web Server

Ever wanted to quickly share a file over a network? Well you are in luck. Python has a feature just for you. Go to the directory which you want to serve over the network and write the following code in your terminal:

```
# Python 2
python -m SimpleHTTPServer

# Python 3
python -m http.server
```

Pretty Printing

You can print a list and dictionary in a beautiful format in the Python repl. Here is the relevant code:

```
from pprint import pprint

my_dict = {'name': 'Yasoob', 'age': 'undefined', 'personality': 'awesome'}
pprint(my_dict)
```

This is more effective on "dict"s. Moreover, if you want to pretty print json quickly from a file then you can simply do:

```
cat file.json | python -m json.tool
```

Profiling a script

This can be extremely helpful in pinpointing the bottlenecks in your scripts:

```
python -m cProfile my_script.py
```

Note: `cProfile` is a faster implementation of `profile` as it is written in c

CSV to json

Run this in the terminal:

```
python -c "import csv,json;print json.dumps(list(csv.reader(open('csv_file.csv'))))"
```

Make sure that you replace `csv_file.csv` to the relevant file name.

List Flattening

You can quickly and easily flatten a list using `itertools.chain.from_iterable` from the `itertools` package. Here is a simple example:

```python
a_list = [[1, 2], [3, 4], [5, 6]]
print(list(itertools.chain.from_iterable(a_list)))
# Output: [1, 2, 3, 4, 5, 6]

# or
print(list(itertools.chain(*a_list)))
# Output: [1, 2, 3, 4, 5, 6]
```

One-Line Constructors

Avoid a lot of boilerplate assignments when initializing a class

```python
class A(object):
    def __init__(self, a, b, c, d, e, f):
        self.__dict__.update({k: v for k, v in locals().items() if k != 'self'})
```

Additional one-liners can be found on the Python website.

For - Else

Loops are an integral part of any language. Likewise for loops are an important part of Python. However there are a few things which most beginners do not know about them. We will discuss a few of them one by one.

Let's first start of by what we know. We know that we can use for loops like this:

```python
fruits = ['apple', 'banana', 'mango']
for fruit in fruits:
    print(fruit.capitalize())

# Output: Apple
#         Banana
#         Mango
```

That is the very basic structure of a for loop. Now let's move on to some of the lesser known features of for loops in Python.

19.1 else clause:

For loops also have an else clause which most of us are unfamiliar with. The else clause executes when the loop completes normally. This means that the loop did not encounter any break. They are really useful once you understand where to use them. I myself came to know about them a lot later.

The common construct is to run a loop and search for an item. If the item is found, we break the loop using `break`. There are two scenarios in which the loop may end. The first one is when the item is found and `break` is encountered. The second scenario is that the loop ends. Now we may want to know which one of these is the reason for a loops completion. One method is to set a flag and then check it once the loop ends. Another is to use the `else` clause.

This is the basic structure of a `for/else` loop:

```python
for item in container:
    if search_something(item):
        # Found it!
        process(item)
        break
else:
    # Didn't find anything..
    not_found_in_container()
```

Consider this simple example which I took from the official documentation:

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

It finds factors for numbers between 2 to 10. Now for the fun part. We can add an additional `else` block which catches the numbers which are prime and tells us so:

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

Open function

open opens a file. Pretty simple, eh? Most of the time, we see it being used like this:

```
f = open('photo.jpg', 'r+')
jpgdata = f.read()
f.close()
```

The reason I am writing this article is that most of the time, I see open used like this. There are **three** errors in the above code. Can you spot them all? If not, read on. By the end of this article, you'll know what's wrong in the above code, and, more importantly, be able to avoid these mistakes in your own code. Let's start with the basics:

The return of open is a file handle, given out from the operating system to your Python application. You will want to return this file handle once you're finished with the file, if only so that your application won't reach the limit of the number of open file handle it can have at once.

Explicitly calling close closes the file handle, but only if the read was successful. If there is any error just after f = open(...), f.close() will not be called (depending on the Python interpreter, the file handle may still be returned, but that's another story). To make sure that the file gets closed whether an exception occurs or not, pack it into a 'with statement:

```
with open('photo.jpg', 'r+') as f:
    jpgdata = f.read()
```

The first argument of open is the filename. The second one (the *mode*) determines

how the file gets opened.

- If you want to read the file, pass in `r`

- If you want to read and write the file, pass in `r+`

- If you want to overwrite the file, pass in `w`

- If you want to append to the file, pass in `a`

While there are a couple of other valid mode strings, chances are you won't ever use them. The mode matters not only because it changes the behavior, but also because it may result in permission errors. For example, if we were to open a jpg-file in a write-protected directory, `open(.., 'r+')` would fail. The mode can contain one further character; we can open the file in binary (you'll get a string of bytes) or text mode (a string of characters).

In general, if the format is written by humans, it tends to be text mode. `jpg` image files are not generally written by humans (and are indeed not readable to humans), and you should therefore open them in binary mode by adding a b to the text string (if you're following the opening example, the correct mode would be `rb`). If you open something in text mode (i.e. add a `t`, or nothing apart from `r`/`r+`/`w`/`a`), you must also know which encoding to use - for a computer, all files are just bytes, not characters.

Unfortunately, open does not allow explicit encoding specification in Python 2.x. However, the function io.open is available in both Python 2.x and 3.x (where it is an alias of open), and does the right thing. You can pass in the encoding with the `encoding` keyword. If you don't pass in any encoding, a system- (and Python-) specific default will be picked. You may be tempted to rely on these defaults, but the defaults are often wrong, or the default encoding cannot actually express all characters (this will happen on Python 2.x and/or Windows). So go ahead and pick an encoding. `utf-8` is a terrific one. When you write a file, you can just pick the encoding to your liking (or the liking of the program that will eventually read your file).

How do you find out which encoding a file you read has? Well, unfortunately, there is no foolproof way to detect the encoding - the same bytes can represent different, but equally valid characters in different encodings. Therefore, you must rely on metadata (for example, in HTTP headers) to know the encoding. Increasingly, formats just define the encoding to be UTF-8.

Armed with this knowledge, let's write a program that reads a file, determines whether it's JPG (hint: These files start with the bytes `FF D8`), and writes a text file that describe the input file.

```python
import io

with open('photo.jpg', 'rb') as inf:
    jpgdata = inf.read()

if jpgdata.startswith(b'\xff\xd8'):
    text = u'This is a jpeg file (%d bytes long)\n'
else:
    text = u'This is a random file (%d bytes long)\n'

with io.open('summary.txt', 'w', encoding='utf-8') as outf:
    outf.write(text % len(jpgdata))
```

I am sure that now you would use open correctly!

Targeting Python 2+3

In a lot of cases you might want to develop programs which can be run in both Python 2+ and 3+.

Just imagine that you have a very popular Python module which is use by hundreds of people but not all of them have Python 2 or 3. In that case you have two choices. The first one is to distribute 2 modules, one for Python 2 and the other for Python 3. The other choice is to modify your current code and make is compatible with both Python 2 and 3.

In this section I am going to highlight some of the tricks which you can employ to make a script compatible with both of them.

Future imports

The first and most important method is to use `__future__` imports. It allows you to import Python 3 functionality in Python 2. Here is an example:

- Context managers were new in Python 2.6+. For using them in Python 2.5 you can use:

```python
from __future__ import with_statement
```

- print function

`print` was changed to a function in Python 3. If you want to use it in Python 2 you can import it from `__future__`.

```python
print
# Output:
```

```python
from __future__ import print_function
print(print)
# Output: <built-in function print>
```

Dealing with module renaming

First tell me how you import packages in your script ? Most of us do this :

```python
import foo
# or
from foo import bar
```

Do you know that you can do something like this as well?

```python
import foo as foo
```

I know its function is the same as the above listed code but it is vital for making your script compatible with Python 2 and 3. Now examine the code below :

```python
try:
    import urllib.request as urllib_request  # for Python 3
except ImportError:
    import urllib2 as urllib_request  # for Python 2
```

So let me explain the above code a little. We are wrapping our importing code in a try except clause. We are doing it because in Python 2 there is no urllib.request module and will result in an ImportError. The functionality of urllib.request is provided by urllib2 module in Python 2. So now when in Python 2 we try to import urllib.request and get an ImportError we tell Python to import urllib2 instead.

The final thing you need to know about is the as keyword. It is mapping the imported module to urllib_request. So that now all of the Classes and methods of urllib2 are available to us by urllib_request.

Obsolete Python 2 builtins

Another thing to keep in mind is that there are 12 Python 2 builtins which have been removed from Python 3. Make sure that you don't use them in Python 2 as well in order to make your code compatible with Python 3. Here is a way to enforce you to abandon these 12 builtins in Python 2 as well.

```python
from future.builtins.disabled import *
```

Now whenever you try to use the modules which are abandoned in Python 3, it raises a NameError like this:

```python
from future.builtins.disabled import *

apply()
# Output: NameError: obsolete Python 2 builtin apply is disabled
```

External standard-library backports

There are a few packages in the wild which provide Python 3 functionality in Python 2. For instance we have:

- enum `pip install enum34`

- singledispatch `pip install singledispatch`

- pathlib `pip install pathlib`

For further reading, the Python documentation has a comprehensive guide of steps you need to take to make your code compatible with both Python 2 and 3.

Coroutines

Coroutines are similar to generators with a few differences. The main differences are:

- generators are data producers
- coroutines are data consumers

First of all let's review the generator creation process. We can make generators like this:

```python
def fib():
    a, b = 0, 1
    while True:
        yield a
        a, b = b, a+b
```

We then commonly use it in a `for` loop like this:

```python
for i in fib():
    print(i)
```

It is fast and does not put a lot of pressure on memory because it **generates** the values on the fly rather then storing them in a list. Now if we use `yield` in the above example more generally we get a coroutine. Coroutines consume values which are sent to it. A very basic example would be a `grep` alternative in Python:

```python
def grep(pattern):
    print("Searching for", pattern)
```

```python
    while True:
        line = (yield)
        if pattern in line:
            print(line)
```

Wait! What does yield return? Well we have turned it into a coroutine. It does not contain any value initially instead we supply it values externally. We supply values by using the .send() method. Here is an example:

```python
search = grep('coroutine')
next(search)
# Output: Searching for coroutine
search.send("I love you")
search.send("Don't you love me?")
search.send("I love coroutines instead!")
# Output: I love coroutines instead!
```

The sent values are accessed by yield. Why did we run next()? It is done to start the coroutine. Just like generators coroutines do not start the function immediately. Instead they run it in response to __next__() and .send() methods. Therefore you have to run next() so that the execution advances to the yield expression.

We can close a coroutine by calling the .close() method. Like:

```python
search = grep('coroutine')
# ...
search.close()
```

There is a lot more to coroutines. I suggest you check out this awesome presentation by David Beazley.

Function caching

Function caching allows us to cache the return values of a function depending on the arguments. It can save time when an I/O bound function is periodically called with the same arguments. Before Python 3.2 we had to write a custom implementation. In Python 3.2+ there is an `lru_cache` decorator which allows us to quickly cache and uncache the return values of a function.

Let's see how we can use it in Python 3.2+ and the versions before it.

23.1 Python 3.2+

Let's implement a Fibonacci calculator and use `lru_cache`.

```
from functools import lru_cache

@lru_cache(maxsize=32)
def fib(n):
    if n < 2:
        return n
    return fib(n-1) + fib(n-2)

>>> print([fib(n) for n in range(10)])
# Output: [0, 1, 1, 2, 3, 5, 8, 13, 21, 34]
```

The `maxsize` argument tells `lru_cache` about how many recent return values to cache.

We can easily uncache the return values as well by using:

```
fib.cache_clear()
```

23.2 Python 2+

There are a couple of ways to achieve the same effect. You can create any type of caching mechanism. It entirely depends upon your needs. Here is a generic cache:

```python
from functools import wraps

def memoize(function):
    memo = {}
    @wraps(function)
    def wrapper(*args):
        if args in memo:
            return memo[args]
        else:
            rv = function(*args)
            memo[args] = rv
            return rv
    return wrapper

@memoize
def fibonacci(n):
    if n < 2: return n
    return fibonacci(n - 1) + fibonacci(n - 2)

fibonacci(25)
```

Here is a fine article by Caktus Group in which they caught a bug in Django which occurred due to `lru_cache`. It's an interesting read. Do check it out.

Context managers

Context managers allow you to allocate and release resources precisely when you want to. The most widely used example of context managers is the `with` statement. Suppose you have two related operations which you'd like to execute as a pair, with a block of code in between. Context managers allow you to do specifically that. For example:

```python
with open('some_file', 'w') as opened_file:
    opened_file.write('Hola!')
```

The above code opens the file, writes some data to it and then closes it. If an error occurs while writing the data to the file, it tries to close it. The above code is equivalent to:

```python
file = open('some_file', 'w')
try:
    file.write('Hola!')
finally:
    file.close()
```

While comparing it to the first example we can see that a lot of boilerplate code is eliminated just by using `with`. The main advantage of using a `with` statement is that it makes sure our file is closed without paying attention to how the nested block exits.

A common use case of context managers is locking and unlocking resources and closing opened files (as I have already showed you).

Let's see how we can implement our own Context Manager. This would allow

us to understand exactly what's going on behind the scenes.

24.1 Implementing Context Manager as a Class:

At the very least a context manager has an __enter__ and __exit__ methods defined. Let's make our own file opening Context Manager and learn the basics.

```python
class File(object):
    def __init__(self, file_name, method):
        self.file_obj = open(file_name, method)
    def __enter__(self):
        return self.file_obj
    def __exit__(self, type, value, traceback):
        self.file_obj.close()
```

Just by defining __enter__ and __exit__ methods we can use it in a with statement. Let's try:

```python
with File('demo.txt', 'w') as opened_file:
    opened_file.write('Hola!')
```

Our __exit__ function accepts three arguments. They are required by every __exit__ method which is a part of a Context Manager class. Let's talk about what happens under-the-hood.

1. The with statement stores the __exit__ method of File class.

2. It calls the __enter__ method of File class.

3. __enter__ method opens the file and returns it.

4. the opened file handle is passed to opened_file.

5. we write to the file using .write()

6. with statement calls the stored __exit__ method.

7. the __exit__ method closes the file.

24.2 Handling exceptions

We did not talk about the `type`, `value` and `traceback` arguments of the `__exit__` method. Between the 4th and 6th step, if an exception occurs, Python passes the type, value and traceback of the exception to the `__exit__` method. It allows the `__exit__` method to decide how to close the file and if any further steps are required. In our case we are not paying any attention to them.

What if our file object raises an exception? We might be trying to access a method on the file object which it does not supports. For instance:

```python
with File('demo.txt', 'w') as opened_file:
    opened_file.undefined_function('Hola!')
```

Let's list down the steps which are taken by the `with` statement when an error is encountered.

1. It passes the type, value and traceback of the error to the `__exit__` method.

2. It allows the `__exit__` method to handle the exception.

3. If `__exit__` returns True then the exception was gracefully handled.

4. If anything else than True is returned by `__exit__` method then the exception is raised by `with` statement.

In our case the `__exit__` method returns `None` (when no return statement is encountered then the method returns `None`). Therefore, `with` statement raises the exception.

```
Traceback (most recent call last):
  File "<stdin>", line 2, in <module>
AttributeError: 'file' object has no attribute 'undefined_function'
```

Let's try handling the exception in the `__exit__` method:

```python
class File(object):
    def __init__(self, file_name, method):
        self.file_obj = open(file_name, method)
    def __enter__(self):
        return self.file_obj
    def __exit__(self, type, value, traceback):
        print("Exception has been handled")
        self.file_obj.close()
        return True
```

```python
with File('demo.txt', 'w') as opened_file:
    opened_file.undefined_function()

# Output: Exception has been handled
```

Our __exit__ method returned True, therefore no exception was raised by the with statement.

This is not the only way to implement context managers. There is another way and we will be looking at it in this next section.

24.3 Implementing a Context Manager as a Generator

We can also implement Context Managers using decorators and generators. Python has a contextlib module for this very purpose. Instead of a class, we can implement a Context Manager using a generator function. Let's see a basic, useless example:

```python
from contextlib import contextmanager

@contextmanager
def open_file(name):
    f = open(name, 'w')
    yield f
    f.close()
```

Okay! This way of implementing Context Managers appear to be more intuitive and easy. However, this method requires some knowledge about generators, yield and decorators. In this example we have not caught any exceptions which might occur. It works in mostly the same way as the previous method.

Let's dissect this method a little.

1. Python encounters the yield keyword. Due to this it creates a generator instead of a normal function.

2. Due to the decoration, contextmanager is called with the function name (open_file) as it's argument.

3. The contextmanager function returns the generator wrapped by the GeneratorContextManager object.

4. The `GeneratorContextManager` is assigned to the `open_file` function. Therefore, when we later call `open_file` function, we are actually calling the `GeneratorContextManager` object.

So now that we know all this, we can use the newly generated Context Manager like this:

```python
with open_file('some_file') as f:
    f.write('hola!')
```