Chapter 1. Python Essentials for DevOps

DevOps, the combination of software development with information technology operations, has been a hot field during the last decade. Traditional boundaries between software development, deployment, maintenance, and quality assurance are broken, enabling more integrated teams. Python has been a popular language both in traditional IT operations and in DevOps due to its combination of flexibility, power, and ease of use.

The Python programming language was publically released in the early 1990s for use in system administration. It has been a great success in this area and has gained wide adoption. Python is a general-purpose programming language used in just about every domain. Visual effects and the motion picture industry have embraced it. More recently, it has become the de facto language of data science and machine learning (ML). It has been used across industries from aviation to bioinformatics. Python has an extensive arsenal of tools to cover the wide-ranging needs of its users. Learning the whole Python Standard Library (the capabilities that come with any Python installation) would be a daunting task. Trying to learn all the third-party packages that enliven the Python ecosystem, would be an immense undertaking. The good news is that you don’t need to do those things. You can become a powerful DevOps practitioner knowing only a small subset of Python.

In this chapter, we draw on our decades of Python DevOps experience to teach only the elements of the language that you need. These are the parts of Python DevOps uses daily. They form the essential toolbox to get things done. Once you have these core concepts down, you can add more complicated tools, as you'll see in later chapters.

Installing and Running Python

If you want to try the code in this overview, you need version Python 3.7 or later installed (the latest release is 3.7.4 as of this writing), and access to a shell. In macOs X, Windows, and most Linux distributions, you can open the terminal application to access a shell. To check the version of Python you are using, open a shell, and type `python --version`.

```
$ python --version

Python 3.7.4
```

Python installers can be downloaded directly from the Python.org website. Alternatively, you can use a package manager such as Apt, RPM, MacPorts, Homebrew, Chocolatey, or many others.

The Python Shell

The simplest way to run Python is to use the built-in interactive interpreter. Just type `python` in a shell. You can then interactively run Python statements. Type `exit()` to exit the shell.

```
$ python

Python 3.7.4 (default, Sep 23 2018, 09:47:03)

[Clang 9.0.0 (clang-900.0.38)] on darwin

Type "help", "copyright", "credits" or
"license" for more information.

>>> 1 + 2

3

>>> exit()
```

PYTHON SCRIPTS

Python code runs from a file with the `.py` extension.

```
# This is my first Python script

print('Hello world!')
```

Save this code to a file named *hello.py*. To invoke the script, in a shell run `python` followed by the filename.

```
$ python hello.py
```

```
Hello world!
```

Python scripts are the way most production Python code runs.

IPYTHON

Besides the built-in interactive shell, several third-party interactive shells run Python code. One of the most popular is IPython. IPython offers *introspection* (the ability to dynamically get information about objects), syntax highlighting, special *magic* commands (which we touch on later in this chapter), and many more features making it a pleasure to use for exploring Python. To install IPython, use the Python package manager, `pip`:

```
$ pip install ipython
```

Running is similar to running the built-in an interactive shell of the previous section:

```
$ ipython

Python 3.7.4 (default, Sep 23 2018, 09:47:03)

Type 'copyright', 'credits' or 'license' for
more information

IPython 7.5.0 -- An enhanced Interactive
Python. Type '?' for help.

In [1]: print('Hello')
```

```
Hello
```

```
In [2]: exit()
```

Jupyter Notebooks

A spin-off from the iPython project, the Jupyter project allows documents containing text, code, and visualizations. These documents are a powerful tool for combining running code, output, and formatted text. Jupyter enables the delivery of documentation along with the code. It has achieved widespread popularity, especially in the data science world. To install and run Jupyter notebooks:

```
$ pip install jupyter
```

```
$ jupyter notebook
```

This command opens a web browser tab showing the current working directory. From here, you can open existing notebooks in the current project or create new ones.

Procedural Programming

If you’ve been around programming at all, you’ve probably heard terms like object oriented programming

(OOP) and functional programming. These are different architectural paradigms used to organizing programs. One of the most basic paradigms, procedural Programming, is an excellent place to start. *Procedural programming* is the issuing instructions to a computer in an ordered sequence:

```
>>> i = 3
>>> j = i +1
>>> i + j
7
```

As you can see in this example, there are three statements, which execute in order from the first line to the last. Each statement uses the state produced by the previous ones. In this case, the first statement assigns the value 3 to a variable named `i`. In the second statement, this variable's value is used to assign a value to a variable named `j`, and in the third statement, the values from both variables add together. Don't worry about the details of these statements yet, notice that they are executed in order and rely on the state left by the previous statements.

Variables

A variable is a name that points to some value. In the previous example, these variables are `i` and `j` . Variables in Python can assign to new values:

```
>>> dog_name = 'spot'

>>> dog_name
```

```
'spot'

>>> dog_name = 'rex'

>>> dog_name

'rex'

>>> dog_name = 't-' + dog_name

>>> dog_name

't-rex'

>>>
```

Python variables use dynamic typing. In practice, this means that they can be reassigned to values of different types or classes:

```
>>> big = 'large'

>>> big

'large'

>>> big = 1000*1000

>>> big

1000000
```

```
>>> big = {}

>>> big

{}

>>>
```

Here the same variable is set to a string, a number, and a dictionary. Variables can be reassigned to values of any type.

Basic Math

Basic math operations such as addition, subtraction, multiplication, and division can all be performed using built-in math operators:

```
>>> 1 + 1

2

>>> 3 - 4

-1

>>> 2*5

10

>>> 2/3
```

```
0.6666666666666666
```

Note that a `//` symbol is for integer division. The symbol `**` creates an exponent, and `%` is the modulo operator:

```
>>> 5/2

2.5

>>> 5//2

2

>>> 3**2

9

>>> 5%2

1
```

Comments

Comments are text ignored by the Python interpreter. They are useful for documentation of code and can be mined by some services to provide standalone documentation. Single line comments are delineated by prepending with . A single line comment can start at the beginning of a line, or any point following. Everything after the is part of the comment until a new line break.

```
# This is a comment
1 + 1 # This comment follows a statement
```

Multi-line comments enclose themselves in blocks beginning and ending with either `"""` or `'`.

```
"""
This statement is a block comment.
It can run for multiple lines
"""

'''
This statement is also a block comment
'''
```

Built-in Functions

Functions are statements grouped to be called as a unit. You invoke a function by calling the function name followed by parentheses. If the function takes arguments, the arguments appear within the parenthesis. Python has many built-in functions. Two of the most widely used build-in functions are `print` and `range`.

Print

The `print` function produces output that a user of a program can view. It is less relevant in interactive environments but is a fundamental tool when writing Python scripts. In the previous example, the argument to the `print` function is written as output when the script runs:

```
# This is my first Python script

print("Hello world!")
```

```
$ python hello.py

Hello world!
```

`print` can be used to see the value of a variable or to give feedback as to the state of a program. `print` generally outputs to standard out and is visible as program output in a shell.

Range

Though `range` is a built-in function, it is technically not a function at all. It is a type representing a sequence of numbers. When calling the `range()` constructor, an object representing a sequence of numbers is returned. Range objects count through a sequence of numbers.
The `range` function takes up to three integer arguments. If only one argument appears, then the sequence is represented by the numbers from zero up to, but not including that number. If a second argument appears, it represents the starting point, rather than the default of starting from 0. The third argument can be used to specify the step distance, and it defaults to 1.

```
>>> range(10)
range(0, 10)
>>> list(range(10))
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> list(range(5, 10))
[5, 6, 7, 8, 9]
>>> list(range(5, 10, 3))
[5, 8]
>>>
```

`range` maintains a small memory footprint, even over extended sequences as it only stores the start, stop, and step values. The `range` function can iterate through long sequences of numbers without performance constraints.

Execution Control

Python has many constructs to control the flow of statement execution. You can group statements you wish to run together as a block of code. These blocks can be run multiple times using `for` and `while` loops, or only run under certain conditions using `if` statements, `while` loops, or `try-except` blocks. Using these constructs is the first step to taking advantage of the power of programming. Different languages demarcate blocks of code using different conventions. Many languages with syntax similar to the C language (a very influential language used in writing Unix) use curly brackets around a group of statements to define a block. In Python, indentation is used to indicate a block. Statements are grouped by indentation into blocks which execute as a unit.

NOTE

The Python interpreter does not care if you use tabs or spaces to indent, as long as you are consistent. The Python style guide, PEP-8, however, recommends using four whitespaces for each level of indentation.

if/elif/else

`if/elif/else` statements are a common way to branch between decisions in code. A block directly after an `if` statement runs if that statement evaluates to `True`:

```
>>> i = 45
>>> if i == 45:
...         print('i is 45')
...
...
i is 45
>>>
```

Here we used the `==` operator which returns `True` if items are equal and `False` if not. This block can optionally follow an `elif` or `else` statement with an accompanying block. In the case of an `elif` statement, this block only executes if the `elif` evaluated to `True`.

```
>>> i = 35
>>> if i == 45:
...         print('i is 45')
... elif i == 35:
...         print('i is 35')
...
...
i is 35
>>>
```

Multiple `elif` loops can append together. If you are familiar with `switch` statements in other languages, this simulates that same behavior of choosing from multiple choices. Adding an `else` statement at the end runs a block if none of the other conditions evaluate as `True`:

```
>>> i = 0
>>> if i == 45:
...         print('i is 45')
... elif i == 35:
...         print('i is 35')
... elif i > 10:
...         print('i is greater than 10')
... elif i%3 == 0:
...         print('i is a multiple of 3')
... else:
...         print('I don't know much about i...')
...
...
```

```
i is a multiple of 3
>>>
```

You can nest if statements, creating blocks containing `if` statements which only execute if an outer `if` statement is `True`.

```
>>> cat = 'spot'
>>> if 's' in cat:
...         print("Found an 's' in a cat")
...         if cat == 'Sheba':
...                 print("I found Sheba")
...         else:
...                 print("Some other cat")
... else:
...         print(" a cat without 's'")
...
...
Found an 's' in a cat
Some other cat
>>>
```

For Loops

for loops allow you to repeat a block of statements (a code block) once for each member of a *sequence* (ordered groups of items). As you iterate through the sequence, the current item can be accessed by the code block. One of most common uses of for loops is to iterate through a `range` object to do a task a set number of times.

```
>>> for i in range(10):
...         x = i*2
...         print(x)
...
...
0
2
4
6
8
10
12
14
16
18
>>>
```

In this example, our block of code is as follows:

```
...     x = i*2
...     print(x)
```

We repeat this code 10 times, each time assigning the variable `i` the next number in the sequence of integers 0–9. `for` loops can be used to iterate through any of the Python Sequence types. You will see these later in this chapter.

CONTINUE

The `continue` statement skips a step in a loop, jumping to the next item in the sequence.

```
>>> for i in range(6):
...     if i == 3:
...         continue
...     print(i)
...
...
0
1
2
4
5
>>>
```

While Loops

`while` loops repeat a block as long as a condition evaluates to `True`.

```
>>> count = 0
>>> while count < 3:
...     print(f"The count is {count}")
...     count += 1
...
```

```
...
The count is 0
The count is 1
The count is 2
>>>
```

It is essential to define a way for your loop to end. Otherwise, you will be stuck in the loop until your program crashes. One way to handle this is to define your conditional statement such that it eventually evaluates to `False`. An alternative pattern uses the `break` statement to exit a loop using a nested conditional.

```
>>> count = 0
>>> while True:
...         print(f"The count is {count}")
...         if count > 5:
...                 break
...         count += 1
...
...
The count is 0
The count is 1
The count is 2
The count is 3
The count is 4
The count is 5
The count is 6
>>>
```

Handling Exceptions

Exceptions are a type of error causing your program to crash if not handled (caught). Catching them with a `try-except` block allows the program to continue. These blocks are created by indenting the block in which the exception might be raised and putting a `try` statement before it and an `except` statement after, followed by a code block which should run when the error occurs:

```
>>> thinkers = ['Plato', 'PlayDo', 'Gumby']
>>> while True:
```

```
...         try:
...             thinker = thinkers.pop()
...             print(thinker)
...         except IndexError as e:
...             print("We tried to pop too many thinkers")
...             print(e)
...             break
...
...
...
Gumby
PlayDo
Plato
We tried to pop too many thinkers
pop from empty list
>>>
```

There are many built-in exceptions, such as `IOError`, `KeyError`, and `ImportError`. Many third-party packages also define their own exception classes. They indicate that something has gone very wrong, so it pays only to catch them if you are confident that the problem should not be fatal to your software. You can specify explicitly which exception type you will catch. Ideally, you should catch the exact exception type (in our example this was the exception `IndexError`).

Built-in Objects

In this overview, we will not be covering object oriented programming. The Python language, however, comes with quite a few built-in classes.

What Is an Object?

In object-oriented programming (OOP), data or state and functionality appear together. The essential concepts to understand when working with objects are *class instantiation* (creating objects from classes) and *dot* *syntax* (the syntax for accessing an object’s attributes and methods). A class defines attributes and methods shared by its objects. Think of it as the technical drawing describing a car model. The class can then be instantiated to create an instance. The instance, or object, is a single car built based on those drawings.

```
>>> # Define a class for fancy defining fancy
cars

>>> class FancyCar():

...     pass

...

>>> type(FancyCar)

<class 'type'>

>>> # Instantiate a fancy car

>>> my_car = FancyCar()

>>> type(my_car)

<class '__main__.FancyCar'>
```

You don’t need to worry about creating your own classes at this point. Just understand that each object is an instantiation of a class.

Object Methods and Attributes

Objects store data in attributes. These attributes are variables attached to the object or object class. Objects define functionality in *object methods* (methods defined for all objects in a class) and *class methods* (methods attached to a class, and shared by all objects in the class), which are functions attached to the object.

NOTE

In Python documentation functions attached to Objects and classes are referred to as methods.

These functions have access to the object’s attributes and can modify and use the object’s data. To call an object’s method or access one of its attributes, we use dot syntax:

```
>>> # Define a class for fancy defining fancy cars
>>> class FancyCar():
...         # Add a class variable
...         wheels = 4
...         # Add a method
...         def driveFast(self):
...                 print("Driving so fast")
...
...
...
>>> # Instantiate a fancy car
>>> my_car = FancyCar()
>>> # Access the class attribute
>>> my_car.wheels
4
>>> # Invoke the method
>>> my_car.driveFast()
Driving so fast
>>>
```

So here our `FancyCar` class defines a method called `driveFast` and an attribute `wheels`. When you instantiate an instance of `FancyCar` named `my_car`, you can access the attribute and invoke the method using the dot syntax.

Sequences

Sequences are a family of built-in types including the *list*, *tuple*, *range*, *string*, and *binary* types. Sequences represent ordered and finite collections of items.

SEQUENCE OPERATIONS

There are many operations which work across all of the types of sequences. We cover some of the most commonly used here.

You can use the `in` and `not in` operators to test if an item exists in a sequence:

```
>>> 2 in [1,2,3]
True
>>> 'a' not in 'cat'
False
>>> 10 in range(12)
True
>>> 10 not in range(2, 4)
True
```

You can reference the contents of a sequence by using its index number. To access the item at some index, use square brackets with the index number as an argument. The first item indexed is at position 0, the second at 1 and so forth up the number one less than the number of items:

```
>>> my_sequence = 'Bill Cheatham'
>>> my_sequence[0]
'B'
>>> my_sequence[2]
'l'
>>> my_sequence[12]
'm'
```

Indexing can appear from the end of a sequence rather than the front using negative numbers. The last item has the index of –1, the second to last –2 and so forth:

```
>>> my_sequence = "Bill Cheatham"
>>> my_sequence[-1]
'm'
>>> my_sequence[-2]
'a'
>>> my_sequence[-13]
'B'
```

The index of an item results from the `index` method. By default, it returns the index of the first occurrence of the item, but optional arguments can define a sub-range in which to search:

```
>>> my_sequence = "Bill Cheatham"
>>> my_sequence.index('C')
5
>>> my_sequence.index('a')
8
>>> my_sequence.index('a',9, 12)
11
>>> my_sequence[11]
'a'
>>>
```

You can produce a new sequence from a sequence using slicing. A slice appears by invoking a sequence with brackets containing optional `start`, `stop`, and `step` arguments:

```
my_sequence[start:stop:step]
```

`start` is the index of the first item to use in the new sequence, `stop` the first index beyond that point, and `step`, the distance between items. These arguments are all optional and are replaced with default values if omitted. This statement produces a copy of the original sequence. The default value for `start` is 0, for `stop` is the length of the sequence, and `step` is 1. Note that if the step does not appear, the corresponding : can also be dropped:

```
>>> my_sequence = ['a', 'b', 'c', 'd', 'e', 'f', 'g']
>>> my_sequence[2:5]
['c', 'd', 'e']
>>> my_sequence[:5]
['a', 'b', 'c', 'd', 'e']
>>> my_sequence[3:]
['d', 'e', 'f', 'g']
>>>
```

Negative numbers can be used to index backward:

```
>>> my_sequence[-6:]
['b', 'c', 'd', 'e', 'f', 'g']
>>> my_sequence[3:-1]
['d', 'e', 'f']
>>>
```

Sequences share many operations for getting information about them and their contents. `len` returns the lengths of the sequence, `min` the smallest member, `max` the largest, and `count` the number of a particular item. `min` and `max` work only on sequences with items that are comparable. Remember that these work with any sequence type:

```
>>> my_sequence = [0, 1, 2, 0, 1, 2, 3, 0, 1, 2, 3, 4]
>>> len(my_sequence)
12
>>> min(my_sequence)
0
>>> max(my_sequence)
4
>>> my_sequence.count(1)
3
>>>
```

LISTS

Lists, one of the most commonly used Python data structures, represent an ordered collection of items of any type. The use of square brackets indicates a list syntax.

The function `list()` can be used to create an empty list or a list based on another finite iterable object (such as another sequence):

```
>>> list()
[]
>>> list(range(10))
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> list("Henry Miller")
['H', 'e', 'n', 'r', 'y', ' ', 'M', 'i', 'l', 'l', 'e', 'r']
>>>
```

Lists created by using square brackets directly are the most common form. Items in the list need to be enumerated explicitly in this case. Remember that the items in a list can be of different types:

```
>>> empty = []
>>> empty
[]
>>> nine = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> nine
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> mixed = [0, 'a', empty, 'WheelHoss']
>>> mixed
[0, 'a', [], 'WheelHoss']
>>>
```

The most efficient way to add a single item to a list is to `append` the item to the end of the list. A less efficient method, `insert`, allows you to insert an item at the index position of your choice:

```
>>> pies = ['cherry', 'apple']
>>> pies
['cherry', 'apple']
>>> pies.append('rhubarb')
>>> pies
['cherry', 'apple', 'rhubarb']
>>> pies.insert(1, 'cream')
>>> pies
['cherry', 'cream', 'apple', 'rhubarb']
>>>
```

The contents of one list can be added to another using the `extend` method:

```
>>> pies
['cherry', 'cream', 'apple', 'rhubarb']
>>> desserts = ['cookies', 'paste']
>>> desserts
['cookies', 'paste']
>>> desserts.extend(pies)
>>> desserts
['cookies', 'paste', 'cherry', 'cream', 'apple', 'rhubarb']
>>>
```

The most efficient and common way of removing the last item from a list and returning its value is to `pop` it. An index can be passed to this method, removing and returning the item at that index. This technique is less efficient as the list needs to be re-indexed:

```
>>> pies
['cherry', 'cream', 'apple', 'rhubarb']
>>> pies.pop()
'rhubarb'
>>> pies
['cherry', 'cream', 'apple']
>>> pies.pop(1)
'cream'
>>> pies
['cherry', 'apple']
```

There is also a `remove` method, which removes the first occurrence of an item.

```
>>> pies.remove('apple')
>>> pies
['cherry']
>>>
```

One of the most potent and idiomatic Python features, list comprehensions, allow you to use the functionality of a `for` loop in a single line. Lets look at a simple example, starting with a `for` loop squaring all of the numbers 0–9 and appending them to a list:

```
>>> squares = []
>>> for i in range(10):
...     squared = i*i
...     squares.append(squared)
...
...
>>> squares
```

```
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
>>>
```

In order to replace this with a list comprehension, we do the following:

```
>>> squares = [i*i for i in range(10)]
>>> squares
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
>>>
```

Note that the functionality of the inner block is put first, followed for the `for` statement. You can also add conditionals to list comprehensions, filtering the results:

```
>>> squares = [i*i for i in range(10) if i%2==0]
>>> squares
[0, 4, 16, 36, 64]
>>>
```

Other techniques for list comprehensions include nesting them and using multiple variables, but the more straightforward form shown here is the most common.

STRINGS

The string sequence type is a collection of ordered characters surrounded by quotation marks. As of Python 3, strings default to using *UTF-8* encoding.

You can create strings either using the string constructor method, `str()`, or by directly enclosing the text in quotation marks:

```
>>> str()
''
>>> "some new string!"
'some new string!'
>>> 'or with single quotes'
'or with single quotes'
```

The string constructor can be used to make strings from other objects.

```
>>> my_list = list()
>>> str(my_list)
'[]'
```

You can create multiple line strings by using triple quotes around the content:

```
>>> multi_line = """This is a
... multi-line string,
... which includes linebreaks.
... """
>>> print(multi_line)
This is a
multi-line string,
which includes linebreaks.
>>>
```

In addition to the methods shared by all sequences, strings have quite a few methods distinct to their class.

It is relatively common for user text to have trailing or leading whitespace. If someone types ' yes ' in a form instead of *yes*, you usually want to treat them the same. Python strings have a `strip` method just for this case. It returns a string with the whitespace removed from the beginning and end. There are also methods to remove the whitespace from only the right or left side of the string:

```
>>> input = "  I want more  "
>>> input.strip()
'I want more'
>>> input.rstrip()
'  I want more'
>>> input.lstrip()
'I want more  '
```

On the other hand, if you want to add padding to a string, you can use the `ljust` or `rjust` methods. Either one pads with whitespace by default, or takes a character argument:

```
>>> output = 'Barry'
>>> output.ljust(10)
'Barry     '
>>> output.rjust(10, '*')
'*****Barry'
```

Sometimes you want to break a string up into a list of sub-strings. Perhaps you have a sentence you want to turn into a list of words or string of words separated by commas. The `split` method breaks a string broken into a list of strings. By default, it uses whitespace as the token to make the breaks. An optional argument can be used to pass in another character to break on:

```
>>> text = "Mary had a little lamb"
>>> text.split()
['Mary', 'had', 'a', 'little', 'lamb']
>>> url = "gt.motomomo.io/v2/api/asset/143"
>>> url.split('/')
['gt.motomomo.io', 'v2', 'api', 'asset', '143']
```

You can easily create a new string from a sequence of strings and `join` them into a single string. This method inserts a string as a separator between a list of other strings:

```
>>> items = ['cow', 'milk', 'bread', 'butter']
>>> " and ".join(items)
'cow and milk and bread and butter'
```

Changing the case of text is a common occurrence, whether it is making the case uniform for comparison or changing in preparation for user consumption. Python strings have several methods to make this an easy process:

```
>>> name = "bill monroe"
>>> name.capitalize()
'Bill monroe'
```

```
>>>  name.upper()
'BILL MONROE'
>>>  name.title()
'Bill Monroe'
>>>  name.swapcase()
'BILL MONROE'
>>>  name  =  "BILL MONROE"
>>>  name.lower()
'bill monroe'
```

Python also provides methods to understand a strings content. Whether it’s checking the case of the text, or seeing if it represents a number, there are quite a few built-in methods for interrogation. Here are just a few of the most commonly used ones:

```
>>>  "William".startswith('W')
True
>>>  "William".startswith('Bill')
False
>>>  "Molly".endswith('olly')
True
>>>  "abc123".isalnum()
True
>>>  "abc123".isalpha()
False
>>>  "abc".isalnum()
True
>>>  "123".isnumeric()
True
>>>  "Sandy".istitle()
True
>>>  "Sandy".islower()
False
>>>  "SANDY".isupper()
True
```

You can insert content into a string and control its format at run-time. Your program can use the values of variables or other calculated content in strings. This approach is used in both creating user consumed text and for writing software logs.

The older form of string formatting in Python comes from the *C* language `printf` function. You can use the modulus operator, `%`, to insert formatted values into a string. This technique applies to the form `string % values`, where values can be a single non-tuple or a tuple of multiple values. The string itself must have a conversion specifier for each value. The conversion specifier, at a minimum, starts with a `%` and is followed by a character representing the type of value inserted:

```
>>> "%s + %s = %s" % (1, 2, "Three")
'1 + 2 = Three'
>>>
```

Additional format arguments include the conversion specifier. For example, you can control the number of places a float, `%f` prints:

```
>>> "%.3f" % 1.234567
'1.235'
```

This mechanism for string formatting was the dominant one in Python for years, and you encounter it in legacy code. This approach offers some compelling features, such as sharing syntax with other languages. It also has some pitfalls. In particular, due to the use of a sequence to hold the arguments, errors related to displaying `tuple` and `dict` objects are common. We recommend adopting newer formatting options, such as the string `format` method, template strings, and f-strings, to both avoid these errors and increase the simplicity and readability of your code.

Python 3 introduced a new way of formatting strings using the string method `format`. This way of formatting has been backported to Python 2 as well. This specification uses curly brackets in the string to indicate replacement fields rather than the modulus based conversion specifiers of the old-style formatting. The insert values become arguments to the string `format` method. The order of the arguments determines their placement order in the target string:

```
>>> '{} comes before {}'.format('first', 'second')
'first comes before second'
>>>
```

You can specify index numbers in the brackets to insert values in an order different than that in the argument list. You can also repeat a value by specifying the same index number in multiple replacement fields:

```
>>> '{1} comes after {0}, but {1} comes before {2}'.format('first',

'second',

'third')
'second comes after first, but second comes before third'
>>>
```

An even more powerful feature is that the insert values can be specified by name:

```
>>> '''{country} is an island.
... {country} is off of the coast of
... {continent} in the {ocean}'''.format(ocean='Indian Ocean',
...
continent='Africa',
...
country='Madagascar')
'Madagascar is an island.
Madagascar is off of the coast of
Africa in the Indian Ocean'
```

Here a `dict` works to supply the key values for name–based replacement fields:

```
>>> values = {'first': 'Bill', 'last': 'Bailey'}
>>> "Won't you come home {first} {last}?".format(**values)
"Won't you come home Bill Bailey?"
```

You can also specify format specification arguments. Here they add left and right padding using `>` and `<`. In the second example, we specify a character to use in the padding:

```
>>> text = "|{0:>22}||{0:<22}|"
>>> text.format('O','O')
'|                     O||O                     |'
>>> text = "|{0:<>22}||{0:><22}|"
>>> text.format('O','O')
'|<<<<<<<<<<<<<<<<<<<<<O||O>>>>>>>>>>>>>>>>>>>>>|'
```

Format specifications are done using the format specification mini-language. Our topic also uses this type of language, *F-strings*.

Python f-strings use the same formatting language as the `format` method, but offer a more straightforward and intuitive mechanism for using it. F-strings are pre-pended with either *f* or *F* before the first quotation mark. Like the `format` string previously described, F-strings use curly braces to demarcate replacement fields. In an F-string, however, the content of the replacement field is an expression. This approach means it can refer to variables defined in the current scope or involve calculations:

```
>>> a = 1
>>> b = 2
>>> f"a is {a}, b is {b}. Adding them results in {a + b}"
'a is 1, b is 2. Adding them results in 3'
```

As in `format` strings, format specifications in F-strings happen within the curly brackets after the value expression and start with a `:`:

```
>>> count = 43
>>> f"|{count:5d}"
'|   43'
```

The value expression can contain nested expressions, referencing variables, and expressions in the construction of the parent expression:

```
>>> padding = 10
>>> f"|{count:{padding}d}"
'|        43'
```

TIP

We highly recommend using F-strings for the majority of your string formatting. They combine the power of the specification mini-language with a simple and intuitive syntax.

Template strings are designed to offer a straightforward string substitution mechanism. These built-in methods work for tasks such as internationalization where simple word substitutions are necessary. They use `$` as a substitution character, with optional curly braces surrounding. The characters directly following the `$` identify the value to be inserted. When the `substitute` method of the string template executes, these names are used to assign values.

NOTE

Built-in types and functions are available whenever you run Python code, but to access the broader world of functionality available in the Python ecosystem, you need to use the `import` statement. This approach lets you add functionality from the Python Standard Library or third-party services into your environment. You can selectively import parts of a package by using the `from` keyword.

```
>>> from string import Template
>>> greeting = Template("$hello Mark Anthony")
>>> greeting.substitute(hello="Bonjour")
'Bonjour Mark Anthony'
>>> greeting.substitute(hello="Zdravstvuyte")
'Zdravstvuyte Mark Anthony'
>>> greeting.substitute(hello="Nǐn hǎo")
'Nǐn hǎo Mark Anthony'
```

DICTS

Aside from strings and lists, dicts may be the most used of the Python built-in classes. A *dict* is a mapping of keys to values. The lookup of any particular value using a key is highly efficient and fast. The keys can be strings, numbers, custom objects, or any other non-mutable type.

NOTE

A *mutable* object is one whose contents can change in place. Lists are a primary example; the contents of the list can change without the list’s identity changing. Strings are not mutable. You create a new string each time you change the contents of an existing one.

Dicts are represented as comma–separated key/value pairs surrounded by curly braces. The key/value pairs consist of a key, a colon (:), and then a value.

You can create a dict object using the `dict()` constructor. With no arguments, it creates an empty dict. It takes a sequence of key/value pairs as an argument as well:

```
>>> map = dict()
>>> type(map)
<class 'dict'>
>>> map
{}
>>> kv_list = [['key-1', 'value-1'], ['key-2', 'value-2']]
>>> dict(kv_list)
{'key-1': 'value-1', 'key-2': 'value-2'}
```

You can also create a *dict* directly using curly braces:

```
>>> map = {'key-1': 'value-1', 'key-2': 'value-2'}
>>> map
{'key-1': 'value-1', 'key-2': 'value-2'}
```

You can access the value associated with a key using square bracket syntax:

```
>>> map['key-1']
'value-1'
>>> map['key-2']
'value-2'
```

You can use the same syntax to set a value. If the key is not in the dict, it adds as a new entry. If it already exists, the value changes to the new value:

```
>>> map
{'key-1': 'value-1', 'key-2': 'value-2'}
>>> map['key-3'] = 'value-3'
>>> map
{'key-1': 'value-1', 'key-2': 'value-2', 'key-3': 'value-3'}
>>> map['key-1'] = 13
>>> map
{'key-1': 13, 'key-2': 'value-2', 'key-3': 'value-3'}
```

If you try to access a key that has not been defined in a dict, a `KeyError` exception will be thrown:

```
>>> map['key-4']
Traceback (most recent call last):
  File "<input>", line 1, in <module>
    map['key-4']
KeyError: 'key-4'
```

You can check if the exists in a *dict* using the *in* syntax we saw with *Sequences*. In the case of *dicts*, it checks for the existence of keys.

```
>>> if 'key-4' in map:
...         print(map['key-4'])
... else:
...         print('key-4 not there')
...
...
key-4 not there
```

A more intuitive solution is to use the `get()` method. If you have not defined a key in a dict, it returns a supplied default value. If you have not supplied a default value is supplied it returns `None`:

```
>>> map.get('key-4', 'default-value')
'default-value'
```

Use `del` to remove a key-value pair from a `dict`.

```
>>> del(map['key-1'])
>>> map
{'key-2': 'value-2', 'key-3': 'value-3'}
```

The `keys()` method returns an `dict_keys` object with the `dict`'s keys. The `values()` method returns an `dict_values` object and the `items()` method returns key-value pairs. This last method is useful for iterating through the contents of a `dict`.

```
>>> map.keys()
dict_keys(['key-1', 'key-2'])
>>> map.values()
dict_values(['value-1', 'value-2'])
>>> for key, value in map.items():
...         print(f"{key}: {value}")
...
...
key-1: value-1
key-2: value-2
```

Similar to list comprehensions, dict comprehensions are one line statements returning a dict by iterating through a sequence:

```
>>> letters = 'abcde'
>>> # mapping individual letters to their upper-case representations
>>> cap_map = {x: x.upper() for x in letters}
>>> cap_map['b']
'B'
```

Functions

You have seen some Python built-in functions already. Now move on to writing your own. Remember, a *function* is a mechanism for encapsulating a block of code. You can repeat the behavior of this block in multiple spots without having to duplicate the code. Your code will be better organized, more testable, maintainable, and easier to understand.

Anatomy of a Function

The first line of a function definition starts with the keyword `def`, followed by the function name, function parameters enclosed in parenthesis, and then `:`. The rest of the function is a code block and is indented:

```
def <FUNCTION NAME>(<PARAMETERS>):

    <CODE BLOCK>
```

If a string using multi-line syntax is provided first in the indented block, it acts as documentation. Use these to describe what your function does, how parameters work, and what it can be expected to return. You will find these docstrings are invaluable to communicate with future users of your code. Various programs and services also use them to create documentation.

Providing docstrings is considered a best practice and is highly recommended:

```
>>> def my_function():
...     '''This is a doc string.
...
...   It should describe what the function does,
...   what parameters work, and what the
...   function returns.
...   '''
```

Function arguments occur in the parenthesis following the function name. They can be either positional or keyword. Positional arguments use the order of the arguments to assign value:

```
>>> def positioned(first, second):
...       """Assignment based on order."""
...       print(f"first: {first}")
...       print(f"second: {second}")
...
...
>>> positioned(1, 2)
first: 1
second: 2
>>>
```

With keyword arguments, assign each argument a default value.

```
>>> def keywords(first=1, second=2):
...       '''Default values assigned'''
...       print(f"first: {first}")
...       print(f"second: {second}")
...
...
```

The default values are used when no values are passed during function invocation. The keyword parameters can be called by name during function invocation, in which case the order will not matter.

```
>>> keywords(0)
first: 0
second: 2
>>> keywords(3,4)
first: 3
second: 4
```

```
>>> keywords(second='one', first='two')
first: two
second: one
```

When using keyword parameters, all parameters defined after a keyword parameter must be keyword parameters as well. All functions return a value. The `return` keyword is used to set this value. If not set from a function definition, the function returns `None`.

```
>>> def no_return():
...         '''No return defined'''
...         pass
...
>>> result = no_return()
>>> print(result)
None
>>> def return_one():
...         '''Returns 1'''
...         return 1
...
>>> result = return_one()
>>> print(result)
1
```

Functions as Objects

Functions are objects. They can be passed around, or stored in data structures. You can define two functions, put them in a list, and then iterate through the list to invoke them:

```
>>> def double(input):
...         '''double input'''
...         return input*2
...
>>> double
<function double at 0x107d34ae8>
>>> type(double)
<class 'function'>
>>> def triple(input):
...         '''Triple input'''
...         return input*3
...
>>> functions = [double, triple]
>>> for function in functions:
...         print(function(3))
...
...
6
```

Anonymous Functions

When you need to create a very limited function, you can create an unnamed (anonymous) one using the `lambda` keyword. Generally, you should limit their use to situations where a function expects a small function as a argument. In this example, you take a `list` of `lists` and `sort` it. The default sorting mechanism compares based on the first item of each sub-list:

```
>>> items = [[0, 'a', 2], [5, 'b', 0], [2, 'c', 1]]
>>> sorted(items)
[[0, 'a', 2], [2, 'c', 1], [5, 'b', 0]]
```

To sort based on something other than the first entry, you can define a method which returns the item’s second entry and pass it into the sorting function’s `key` parameter.

```
>>> def second(item):
...     '''return second entry'''
...     return item[1]
...
>>> sorted(items, key=second)
[[0, 'a', 2], [5, 'b', 0], [2, 'c', 1]]
```

With the `lambda` keyword, you can do the same thing without the full function definition. Lambda’s work with the `lambda` keyword followed by a parameter name, then a colon and a return value

```
lambda <PARAM>: <RETURN EXPRESSION>
```

Sort using lambdas, first using the second entry, and then using the third.

```
>>> sorted(items, key=lambda item: item[1])
[[0, 'a', 2], [5, 'b', 0], [2, 'c', 1]]
```

```
>>> sorted(items, key=lambda item: item[2])
[[5, 'b', 0], [2, 'c', 1], [0, 'a', 2]]
```

Be cautious of using lambdas more generally, they can create code that is poorly documented and confusing to read if used in place of general functions.

Using Regular Expressions

The need to match patterns in strings comes up again and again. It could be looking for an identifier in a log file or checking user input for keywords, a myriad of other cases. You have already seen simple pattern matching using the `in` operation for sequences or the string `.endswith` and `.startswith` methods. To do more sophisticated matching, you need a more powerful tool. Regular expressions, often referred to as regex, are the answer. Regular expressions use a string of characters to define search patterns. The Python `re` package offers regular expression operations similar to those found Perl. The `re` module uses backslashes (\) to delineate special characters used in matching. To avoid confusion with regular string escape sequences, raw strings are recommended in defining regular expression patterns. Raw strings are prepended with an *r* before the first quotation mark.

NOTE

Python strings have several escape sequences. Among the most common are linefeed `\n` and tab `\t`

Searching

Let say you have a cc list from an email as a text and you want to understand more about who is in this list:

```
In [1]: cc_list = '''Ezra Koenig <ekoenig@vpwk.com>,
   ...: Rostam Batmanglij <rostam@vpwk.com>,
   ...: Chris Tomson <ctomson@vpwk.com,
   ...: Bobbi Baio <bbaio@vpwk.com'''
```

If you want to know whether a name is in this text, you could use the `in` sequence membership syntax:

```
In [2]: 'Rostam' in cc_list
Out[2]: True
```

To get similar behavior, you can use the `re.search` function, which returns a `re.Match` object only if there is a match:

```
In [3]: import re

In [4]: re.search(r'Rostam', cc_list)
Out[4]: <re.Match object; span=(32, 38), match='Rostam'>
```

You can use this as a conditional to test for membership:

```
>>> if re.search(r'Rostam', cc_list):
...         print('Found Rostam')
...
...
Found Rostam
```

Character sets

Okay, so far `re` hasn't given you anything you couldn't do with the `in` operator. However, what if you are looking for a person in a text, but you can't remember if their name is *Bobbi* or *Robby*?

With regular expressions, you can use groups of character, any one of which could appear in a spot. These are called character sets. The characters from which a match should be chosen are enclosed by square brackets in the regular expression definition. You can match on *B* or *R* followed by *obb* and either *i* or *y*:

```
In  [5]: re.search(r'[R,B]obb[i,y]', cc_list)
Out[5]: <re.Match object; span=(101, 106), match='Bobbi'>
```

You can put comma–separated individual characters in a character set, or use ranges. The range *A-Z* includes all the capitalized letters, *0–9* the digits from zero to nine:

```
In  [6]: re.search(r'Chr[a-z][a-z]', cc_list)
Out [6]: <re.Match object; span=(69, 74), match='Chris'>
```

The + after an item in a regular expression matches one or more of it. A number in brackets matches an exact number of characters.

```
In  [7]: re.search(r'[A-Za-z]+', cc_list)
Out [7]: <re.Match object; span=(0, 4), match='Ezra'>
In  [8]: re.search(r'[A-Za-z]{6}', cc_list)
Out [8]: <re.Match object; span=(5, 11), match='Koenig'>
```

We can construct a match using a combination of character sets and other characters to make a naive match of an email address. The . character has a special meaning. It is a wildcard and matches any character. To match against the actual . character, you must escape it using a backslash:

```
In  [9]: re.search(r'[A-Za-z]+@[a-z]+\.[a-z]+', cc_list)
Out[9]: <re.Match object; span=(13, 29), match='ekoenig@vpwk.com'>
```

This example is just a demonstration of character sets. It does not represent the full complexity of a production-ready regex for emails.

Character Classes

In addition to character sets, Python's `re` offers character classes. These are pre-made characters sets. Some commonly used ones are `\w`, which is equivalent to `[a-zA-Z0-9_]` and `\d` which is equivalent to `[0-9]`. You can use the + modifier to match for multiple characters:

```
>>> re.search(r'\w+', cc_list)
<re.Match object; span=(0, 4), match='Ezra'>
```

And you can replace our primative email matcher with `\w`:

```
>>> re.search(r'\w+\@\w+\.\w+', cc_list)
<re.Match object; span=(13, 29), match='ekoenig@vpwk.com'>
```

Groups

You can use parentheses to define groups in a match. These groups can be accessed from the match object. They are numbered in the order they appear, with the zero group being the full match.

```
>>> re.search(r'(\w+)\@(\w+)\.(\w+)', cc_list)
<re.Match object; span=(13, 29), match='ekoenig@vpwk.com'>
>>> matched = re.search(r'(\w+)\@(\w+)\.(\w+)', cc_list)
>>> matched.group(0)
'ekoenig@vpwk.com'
>>> matched.group(1)
'ekoenig'
>>> matched.group(2)
'vpwk'
>>> matched.group(3)
'com'
```

Named Groups

You can also supply names for the groups by adding `?P<NAME>` in the group definition. Then you can access the groups by name instead of number:

```
>>> matched = re.search(r'(?P<name>\w+)\@(?P<SLD>\w+)\.(?P<TLD>\w+)',
cc_list)
>>> matched.group('name')
```

```
'ekoenig'
>>> print(f'''name: {matched.group("name")}
... Secondary Level Domain: {matched.group("SLD")}
... Top Level Domain: {matched.group("TLD")}''')
name: ekoenig
Secondary Level Domain: vpwk
Top Level Domain: com
```

Find All

Up until now, we have demonstrated returning just the first match found. We can also use `findall` to return all of the matches as a list of strings:

```
>>> matched = re.findall(r'\w+\@\w+\.\w+', cc_list)
>>> matched
['ekoenig@vpwk.com', 'rostam@vpwk.com', 'ctomson@vpwk.com',
'cbaio@vpwk.com']
>>> matched = re.findall(r'(\w+)\@(\w+)\.(\w+)', cc_list)
>>> matched
[('ekoenig', 'vpwk', 'com'), ('rostam', 'vpwk', 'com'),
  ('ctomson', 'vpwk', 'com'), ('cbaio', 'vpwk', 'com')]
>>> names = [x[0] for x in matched]
>>> names
['ekoenig', 'rostam', 'ctomson', 'cbaio']
```

Find Iterator

When dealing with large texts, such as logs, it is useful to not process the text all at once. You can produce an *iterator* object using the `finditer` method. This object processes text until it finds a match, and then stop. Passing it to the `next` function returns the current match and continues processing until finding the next match. In this way, you can deal with each match individually without devoting resources to process all of the input at once.

```
>>> matched = re.finditer(r'\w+\@\w+\.\w+', cc_list)
>>> matched
<callable_iterator object at 0x108e68748>
>>> next(matched)
<re.Match object; span=(13, 29), match='ekoenig@vpwk.com'>
>>> next(matched)
<re.Match object; span=(51, 66), match='rostam@vpwk.com'>
```

```
>>> next(matched)
<re.Match object; span=(83, 99), match='ctomson@vpwk.com'>
```

The iterator object, `matched`, can be used in a `for` loop as well.

```
>>> matched = re.finditer("(?P<name>\w+)\@(?P<SLD>\w+)\.(?P<TLD>\w+)",
cc_list)
>>> for m in matched:
...       print(m.groupdict())
...
...
{'name': 'ekoenig', 'SLD': 'vpwk', 'TLD': 'com'}
{'name': 'rostam', 'SLD': 'vpwk', 'TLD': 'com'}
{'name': 'ctomson', 'SLD': 'vpwk', 'TLD': 'com'}
{'name': 'cbaio', 'SLD': 'vpwk', 'TLD': 'com'}
```

Substitution

Besides searching and matching, regexes can be used to substitute part or all of a string:

```
>>> re.sub("\d", "#", "The passcode you entered was  09876")
'The passcode you entered was  #####'
>>> users = re.sub("(?P<name>\w+)\@(?P<SLD>\w+)\.(?P<TLD>\w+)",
                        "\g<TLD>.\g<SLD>.\g<name>", cc_list)
>>> print(users)
Ezra Koenig <com.vpwk.ekoenig>,
Rostam Batmanglij <com.vpwk.rostam>,
Chris Tomson <com.vpwk.ctomson,
Chris Baio <com.vpwk.cbaio
```

Compiling

All of the examples so far have called methods on the `re` module directly. This is adequate for many cases, but if the same match is going to happen many times, performance gains can be had by compiling the regular expression into an object. This object can be re-used for matches without re-compiling:

```
>>> regex = re.compile(r'\w+\@\w+\.\w+')
>>> regex.search(cc_list)
<re.Match object; span=(13, 29), match='ekoenig@vpwk.com'>
```

Regular expressions offer many more features than we have dealt with here. Indeed many books have been written on their use, but you should now be prepared for most basic cases.

Lazy Evaluation

Lazy evaluation is the idea that, especially when dealing with large amounts of data, you do not want process all of the data before using the results. You have already seen this with the `range` type, where the memory footprint is the same, even for one representing a large group of numbers.

Generators

You can use generators in a similar way to `range` objects. They perform some operation on data in chunks as requested. They pause their state in between calls. This means that you can store variables that are needed to calculate output and they are accessed every time the generator is called.

To write a generator function, use the `yield` keyword rather than a return statement. Every time the generator is called, it returns the value specified by `yield` and then pauses it's state until it is next called. Let's write a generator that simply counts, return each subsequent number:

```
>>> def count():
...         n = 0
...         while True:
```

```
...             n += 1
...             yield n
...
...
>>> counter = count()
>>> counter
<generator object count at 0x10e8509a8>
>>> next(counter)
1
>>> next(counter)
2
>>> next(counter)
3
```

Note that the generator keeps track of its state, and hence the variable `n` in each call to the generator reflects the value previously set. Let's implement a Fibonacci generator:

```
>>> def fib():
...     first = 0
...     last = 1
...     while True:
...         first, last = last, first + last
...         yield first
...
>>> f = fib()
>>> next(f)
1
>>> next(f)
1
>>> next(f)
2
>>> next(f)
3
```

We can also iterate using the generator in a `for` loop.

```
>>> f = fib()
>>> for x in f:
...     print(x)
...     if x > 12:
...         break
...
1
1
2
3
5
8
13
```

Generator Comprehensions

We can use generator comprehensions to create one–line generators. They are created using a similar syntax to list comprehensions, but parentheses are used rather than square brackets.

```
>>> list_o_nums = [x for x in range(100)]
>>> gen_o_nums = (x for x in range(100))
>>> list_o_nums
[0, 1, 2, 3, ...  97, 98, 99]
>>> gen_o_nums
<generator object <genexpr> at 0x10ea14408>
```

Even with this small example, we can see the difference in memory used by using the `sys.getsizeof` method which returns the size of an object in bytes.

```
>>> import sys
>>> sys.getsizeof(list_o_nums)
912
>>> sys.getsizeof(gen_o_nums)
120
```

More IPython Features

You saw some of IPython’s features at the beginning of the chapter. Now let’s look at some more advanced features, running shell commands from within the IPython interpreter and using magic functions.

Using IPython to Run Unix Shell Commands

You can use IPython to run shell commands. This is one of the most compelling reasons to perform DevOps actions in the IPython shell. Let’s take a look at a very simple example where the `!` character, which IPython uses to identify shell commands, is put in front of the command `ls`:

```
In [3]: var_ls = !ls -l
In [4]: type(var_ls)
Out[4]: IPython.utils.text.SList
```

The output of the command is assigned to a Python variable `var_ls`. The `type` of this variable is `IPython.utils.text.SList`. The `SList` type converts a regular shell command into an object that has three main methods: `fields`, `grep`, `sort`. Here is an example in action using the Unix `df` command. The `sort` method can interpret the white space from this Unix command and then sort the third column by size.

```
In [6]: df = !df
In [7]: df.sort(3, nums = True)
```

Let's take a look at `SList` and `.grep` next. Here is an example that greps for what `kill` commands are installed in the */usr/bin* directory:

```
In [10]: ls = !ls -l /usr/bin
In [11]: ls.grep("kill")
Out[11]:
['-rwxr-xr-x   1 root   wheel      1621 Aug 20  2018 kill.d',
  '-rwxr-xr-x   1 root   wheel     23984 Mar 20 23:10 killall',
  '-rwxr-xr-x   1 root   wheel     30512 Mar 20 23:10 pkill']
```

The key take away here is this is that IPython a dream environment for hacking around with little shell scripts.

USING IPYTHON MAGIC COMMANDS

If you get in the habit of using IPython, you should also get in the habit of using built-in magic commands. They are essentially shortcuts that pack a big punch. Magic commands are indicated by prepending them with `%%`.

Here is an example of how to write inline bash inside of IPython. Note, this is just a small command, but it could be an entire bash script:

```
In [13]: %%bash
      ...: uname -a
      ...:
      ...:
Darwin nogibjj.local 18.5.0 Darwin Kernel Version 18.5.0: Mon Mar ...
```

The `%%writefile` is pretty tricky because you can write and test Python or Bash scripts on the fly, using IPython to execute them. Not a bad party trick at all:

```
In [16]: %%writefile print_time.py
      ...: #!/usr/bin/env python
      ...: import datetime
      ...: print(datetime.datetime.now().time())
      ...:
      ...:
      ...:
Writing print_time.py

In [17]: cat print_time.py
#!/usr/bin/env python
import datetime
print(datetime.datetime.now().time())

In [18]: !python print_time.py
19:06:00.594914
```

Another very useful command, `%who`, will show you what is loaded into memory. It comes in quite handy when you have been working in a terminal that has been running for a long time:

```
In [20]: %who
df        ls        var_ls
```

Exercises

- Write a Python function which takes a name as an argument and prints that name.

- Write a Python function which takes a string as an argument and prints whether it is upper or lower case.
- Write a list comprehension which results in a list of every letter in the word *smogtether* capitalized.
- Write a generator which alternates between returning *Even* and *Odd*.