Community Experience Distilled

What you need to know about Python

The absolute essentials you need to get Python up and running

Pierluigi Riti

What You Need to Know about Python

The absolute essentials you need to get Python up and running

Pierluigi Riti

[PACKT] PUBLISHING

BIRMINGHAM - MUMBAI

What You Need to Know about Python

First Published: August 2016

Production reference: 1120816

Published by Packt Publishing Ltd.
Livery Place
35 Livery Street
Birmingham B3 2PB, UK.

www.packtpub.com

About the Author

Pierluigi Riti is a cloud and DevOps engineer. He started development on a Commodore 64 when he was only 10 years old, and then continued to work with software and architecture. He moved on to DevOps and the cloud later. Pierluigi also specializes in continuous integration and continuous delivery, and is interested in all cloud technologies and architectures.

You can contact him at `cloudconsultating@gmail.com`.

I would like to thank my wife, Mara Ester, for giving me the best things in my life, our two kids, Nicole and Mattia. I would also like to thank the Packt Publishing team, especially Aishwarya and Larissa, without whose work this book would never have existed.

About the Reviewer

Sanjeev Jaiswal is a computer graduate with 5 years of industrial experience. He basically uses Perl and GNU/Linux for his day-to-day work. He also teaches Drupal and WordPress CMS to bloggers. He first developed an interest in web application penetration testing in 2013; he is currently working on projects involving penetration testing, source code review, and log analysis, where he provides analysis and defense against various kinds of web-based attacks.

Sanjeev loves teaching technical concepts to engineering students and IT professionals, and has been teaching for the last 6 years in his leisure time. He founded Alien Coders (`http://www.aliencoders.org`), based on the learning through sharing principle, for computer science students and IT professionals in 2010, which became a huge hit in India among engineering students.

He usually uploads technical videos to YouTube under the *Alien Coders* tag. He has got a huge fan base at his site because of his simple but effective way of teaching and his philanthropic nature toward students. You can follow him on Facebook at `http://www.facebook.com/aliencoders` and on Twitter at `@aliencoders`.

He wrote *Instant PageSpeed Optimization* for Packt Publishing, and looks forward to authoring or reviewing more books for Packt Publishing and other publishers.

Table of Contents

What you need to know about Python

This eGuide is designed to act as a brief, practical introduction to Python. It is full of practical examples which will get you up and running quickly with the core tasks of Python.

We assume that you know a bit about what Python is, what it does, and why you want to use it, so this eGuide won't give you a history lesson in the background of Python. What this eGuide will give you, however, is a greater understanding of the key basics of Python so that you have a good idea of how to advance after you've read the guide. We can then point you in the right direction of what to learn next after giving you the basic knowledge to do so.

What You Need to Know about Python will:

- Cover the fundamentals and the things you really need to know, rather than niche or specialized areas.
- Assume that you come from a fairly technical background and so understand what the technology is and what it broadly does.
- Focus on what things are and how they work.
- Include examples to get you up, running, and productive quickly.

Overview

When, 25 years ago, Guido Van Rossum invented the Python language, he probably didn't imagine how diffused the language would become.

Python is actually one of the most diffused programming languages in the world. The areas of interest of Python essentially cover all of computer programming.

Python is easy to learn and use, and because of that, it has become one of the most popular languages. Python makes it easy to create complex web applications. Because of the nature of the language, it's easy to create Unix/Linux scripts for maintaining systems, as well as being possible to operate with database and scientific calculations to create powerful big data analysis.

In this eGuide, we will cover different areas of the language; in particular, we will see the basics of different areas and how to use some of the major libraries. This eGuide is just an introduction, and its aim is to introduce the libraries and "make you hungry" to learn more about Python.

1
Python in 2016

When reading Internet discussions, I sometimes see people ask, "Why learn Python in 2016?", but for me the real question is, "Why not learn Python in 2016?"

Python is a general-purpose, high-level, and interpreted dynamic language. The language was first established in 1991 and designed by Guido van Rossum. Python is classified as a scripting language; this means that it doesn't need to be built to be executed. Python is often considered as a binding of Java and C/C++. Python has a clear readable syntax, like Java, but at the same time can be used for writing powerful programs, like C/C++.

Today it's possible to use Python in various different environments, and it's possible to find a different "dialect" of Python, such as IronPython, a .NET version of Python. With Python, it's possible to create different kinds of applications; for example, it's possible to use Python to manipulate data, to write powerful web applications, or for some DevOps automations.

Python is a high-level language, so developers don't need to manage memory. Python has an internal garbage collector, like Java, for freeing up memory when it is not needed. The main emphasis of Python lies in its readable code and potential for high productivity; with Python it's easy to write complex programs with just a few lines of code. Python can be used in a huge range of application domains; here is a list of some of the domains and frameworks available with Python:

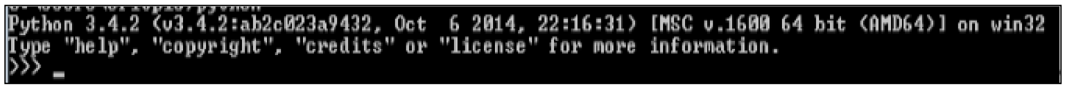
Web/cloud development	
Django	Pyramid
Flask	Bootle
Django CMS	Plone
AWS SDK	

Scientific/numeric	
SciPy	Pandas
iPython	
System administration/infrastructure as code	
Ansible	Salt
OpenStack	DevStack

Getting started with coding

The best way to learn a new language is to "get your hands dirty," so it's time to start with some code. The first step is to download the Python interpreter for your environment, which you can download directly from the Python website, `https://www.python.org/`. In this book, we're using version 3.4.4.

Follow the steps for installing Python, using the guide shown on the site. When Python is installed, open the command line and type `python`; this will open the Python REPL:



```
Python 3.4.2 (v3.4.2:ab2c023a9432, Oct  6 2014, 22:16:31) [MSC v.1600 64 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license" for more information.
>>> _
```

Python is a dynamic-typed language, so developers don't need to declare variable types before using it. The following few lines of code are an example of how to create and determine the type of variable:

```
>>> regards='hello'
>>> regards
'hello'
>>> type(regards)
<class 'str'>
>>>
```

It's possible to see the variable and its associated string in Python without any explicit declaration.

Storing and manipulating data in Python

In Python, there are four built-in data structures: **list**, **tuple**, **dictionary**, and **set**. These data structures are used to store and manipulate data. We can organize the data structure into four different families:

- **Ordered data structure**: For example, list and tuple

- **Unordered data structure**: For example, set and dictionary
- **Mutable**: For example, set, list, and dictionary
- **Immutable**: For example, tuple

A *mutable* data structure can change its size, whereas an *immutable* data structure will always maintain the same size.

We will now give you a short introduction to the different data structures and how to use them.

Lists

A list is a data structure that holds an ordered collection of items. A list in Python starts from 0, meaning the first element of the list is in position 0, the second in 1, and so on:

```
>>> test_list = ['test1','test2',3]
```

The syntax for creating a list is very simple: just declare a variable and create the list with square parentheses `[ ]`. Inside the parentheses, add the value of the list separated by a comma (`,`).

To access an element of the list, just use the position:

```
>>> test_list[1]
'test2'
```

We can read the list with a `for` loop:

```
>>> for item in test_list:
  print(item)
test1
test2
3
```

To remove an element from a list, use the word `del`, followed by the position of the element in the list:

```
>>> del test_list[1]
>>> test_list
['test1', 3]
```

Tuples

Tuples are similar to lists, only tuples are immutable. Tuples are defined by specifying items separated by commas within an optional pair of parentheses. Because tuples are immutable objects, they are usually used when the list of values doesn't change.

```
>>> test_tuple = 'element1','element2','element3';
>>> test_tuple
('element1', 'element2', 'element3')
```

Or we can define a tuple using parentheses:

```
>>> test_tuple_new = ('element1','element2','element3')
>>> test_tuple_new
('element1', 'element2', 'element3')
```

It's possible to access an element of a tuple in the same way we access an element of a list:

```
>>> test_tuple[0]
'element1'
```

Dictionary

A dictionary is a key-value structure, and it's possible to retrieve the value using the key. The key of the dictionary can be only created by using an immutable object, and the value can be either a mutable or immutable object.

Key-value pairs in a dictionary are created using the notation `d = {key : value, key : value}`, as follows:

```
>>> dictionary = {'ab' : 'abcd','cd' : 'efgh'}
```

To access a member of the dictionary, use the following syntax:

```
>>> dictionary['ab']
```

Because a dictionary is a mutable structure, it is possible to delete one element of the dictionary using the word `del`:

```
>>> del dictionary['ab']
>>> dictionary
{'cd': 'efgh'}
```

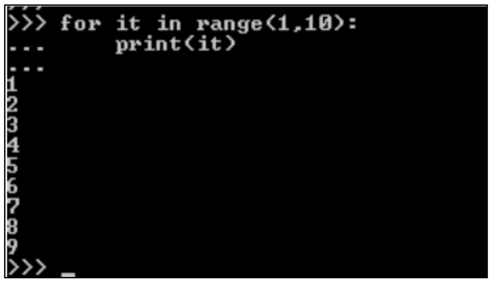
Sets

Sets are unordered collections of unique simple elements. A set is used when the collection is more important than the order of the elements or how many times they occur:

```
>>> set(['Jane','Mark','Stephen'])
{'Mark', 'Jane', 'Stephen'}
```

For loops

The `for…in` statement is a looping statement that iterates over a sequence of objects, for example, a tuple or a list:



```
>>> for it in range(1,10):
...     print(it)
...
1
2
3
4
5
6
7
8
9
>>> _
```

The preceding code prints out a set of numbers to generate the number used in the function range. The range returns a list of numbers starting from the first number. The `for` loop just iterates the range and prints out all numbers in the list.

Functions

Creating a function in Python is easy; the syntax is: `def <name of method>(<parameters>):`

```
#!/usr/bin/python
def first_program(time):
    if 6 <= time <= 12:
        print("Good Morning\n")
    elif 12 <= time <= 18:
        print("Good Afternoon\n")
    else:
        print("Good Evening\n")

if __name__ == "__main__":
    first_program(10)
```

```
        first_program(5)
        first_program(15)
```

This simple program shows some basic features of Python: we define a simple function with a parameter and then use an `if…then…else` to print out a value.

The line `def first_program(time):` defines a method in Python and indicates the parameter input for the method. Because Python doesn't need the type to define a variable, the parameter of the method doesn't need the type either. Another difference between Python and other languages is that Python doesn't need a return type for the method.

Using object-oriented programming in Python

Creating a class in Python is easy: just use the word `class` followed by the name. A basic class in Python looks as follows:

```
>>> class BasicClass:
  Pass
```

This creates a basic class without any method. To create an instance of the class, just create a variable and associate it with that class. Unlike in other languages, in Python we don't need the word `new` to create a class:

```
>>> basic = BasicClass()
>>> basic
<__main__.BasicClass object at 0x000000000347A6A0>
```

Class methods

A class method in Python is defined in the same way that we define a function; the only difference is with the parameter itself. This parameter should be defined before any other parameter. If the class doesn't have a parameter, it must only have `self`:

```
class SayHello:
  def hello(self)
    print('Hello World')
```

The method `hello` uses the parameter `self`. This parameter is the equivalent of `this` in other languages, such as Java.

Every class method needs this parameter for it to be correctly defined. To use this class, just instance it and call the method:

```
>> hello = SayHello()
>> hello.hello()
Hello World!
```

We have just seen a high-level introduction to the Python language, and we can now start to learn more about the language and explore some areas where Python is actually used.

2
Cloud and Web Development

In this chapter, we will see how to use Python for web and cloud development. Web development with Python is easy, and there are several frameworks available that can help our daily job.

With Python it's possible to use **Django**, an MVC framework that allows the developer to use the pure MVC pattern, or **Flask**, a micro-framework based on **WerKezeug**, a WSGI library, and **Jinja2**, an HTML templating language.

Another aspect we will look at is how to use Python for REST and cloud development. We will see how to use **AWS SDK**, the Amazon SDK for developing on the cloud, and how we can create REST web services in Python with Flask.

Web applications with Django

Django is a web application framework written in Python and designed to build complex web applications quickly without any hassle.

Installing Django is easy; just use this command on your command line:

```
pip install Django==1.9.6
```

This command will download and install Django version 1.9.6, which is the most recent version. To check if Django is correctly installed, we can use this simple command:

```
python -c "import django; print(django.get_version())"
```

The output of the command should be:

```
1.9.6
```

If Django is not installed, you'll get an error saying, `No module named Django`.

Your first project with Django

With Django correctly installed on your local machine, it's possible to start creating your first Django project.

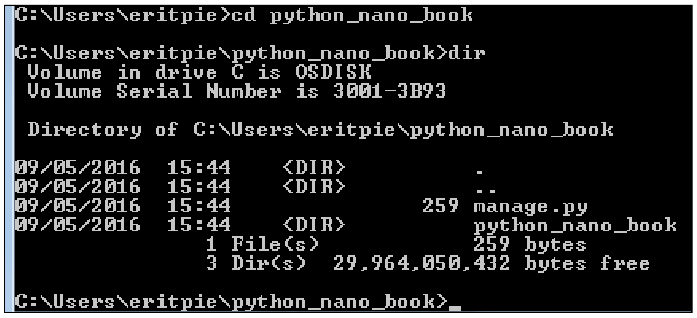
Creating a new project in Django is easy; from the command line, just use this command:

```
django-admin startproject <my new site>
```

Here is an example:

```
django-admin startproject python_nano_book
```

This creates a folder called `python_nano_book`; this folder will contain the basic folder/files for the basic Django project:



```
C:\Users\eritpie>cd python_nano_book

C:\Users\eritpie\python_nano_book>dir
 Volume in drive C is OSDISK
 Volume Serial Number is 3001-3B93

 Directory of C:\Users\eritpie\python_nano_book

09/05/2016  15:44    <DIR>          .
09/05/2016  15:44    <DIR>          ..
09/05/2016  15:44               259 manage.py
09/05/2016  15:44    <DIR>          python_nano_book
               1 File(s)            259 bytes
               3 Dir(s)  29,964,050,432 bytes free

C:\Users\eritpie\python_nano_book>_
```

The command `startproject` creates a basic folder structure necessary for beginning our Django site.

Let's take a look at the folder structure created by the `startproject` command:

- The `python_nano_book` folder is just a container for our server/app (Django doesn't use it), and we can rename it after creation without any problems. It is the Python package created for the project.

  Inside the `python_nano_book` folder, it's possible to find some other files for managing the Django application:

  - `__init__.py`: This file is empty, but when creating a package, it's necessary to tell Python it's a package
  - `setting.py`: This file contains the configuration for your Django project
  - `urls.py`: This file shows the route for your Django project; every new "view" for the project is described here

    - `wsgi.py`: This file contains the entry point for the WSGI-compatible web servers

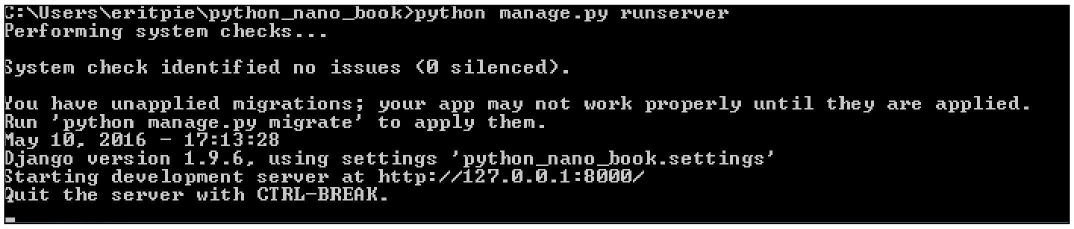
- The `manage.py` file is a command-line utility; with this file, it's possible to interact with Django to execute different operations.

The Django server

The first step in starting work with Django is to start the Django server. To start the server, use the following command:

```
python manage.py runserver
```

This command starts the Django server, the server used to contain our app. We will talk about this app later on in the book. The output of the command is as follows:

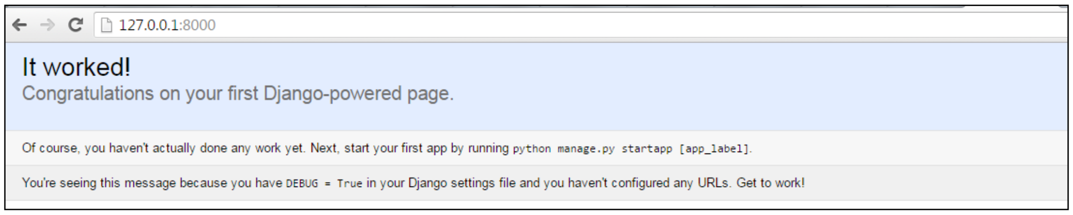


```
C:\Users\eritpie\python_nano_book>python manage.py runserver
Performing system checks...

System check identified no issues (0 silenced).

You have unapplied migrations; your app may not work properly until they are applied.
Run 'python manage.py migrate' to apply them.
May 10, 2016 - 17:13:28
Django version 1.9.6, using settings 'python_nano_book.settings'
Starting development server at http://127.0.0.1:8000/
Quit the server with CTRL-BREAK.
```

The server included in Django is a very lightweight server, written entirely in Python. With this server it's possible to create our Django app without using an Apache or NGINX server.

What we will do next is open a browser and type this address: `http://127.0.0.1:8000`.

The result will be a page that looks as follows:



127.0.0.1:8000

It worked!

Congratulations on your first Django-powered page.

Of course, you haven't actually done any work yet. Next, start your first app by running `python manage.py startapp [app_label]`.

You're seeing this message because you have `DEBUG = True` in your Django settings file and you haven't configured any URLs. Get to work!

This page advises us that we don't have an app installed, as at this time only the basic server is running. However, it does show our basic Django infrastructure at work.

For the next step, we need to write our first app and see how easily Django develops and builds a web application.

Our first Django app

Until now we have seen how we can create a basic Django project and run it, so now it's time to create an "application".

An application is a specific piece of functionality contained in a project; an app, for example, can be a to-do list. A project can contain multiple applications and an application can be present in multiple projects.

To create a new Python app, use this command:

```
python manage.py startapp python_book
```

The folder structure for a Python app is as follows:

```
python_book\
  admin.py
  apps.py
  migrations\
  models.py
  tests.py
  views.py
  __init__.py
```

The descriptions of the files/folders are as follows:

- `admin.py`: This file is used to describe all the models and should be shown in the Django admin page.
- `apps.py`: This file contains the classes for the basic configuration of our Python app.
- `Migrations/`: This folder contains all the migrations for our application; a migration is created by the difference between the actual database and the new model. We will talk about migrations later on in this section.
- `models.py`: This file contains the database model for our Django application; in our case, our application doesn't have a database so we have left the file blank. Every new model we add in this file will generate a new migration.
- `tests.py`: This file contains the tests necessary for our application.
- `views.py`: This file contains the interceptor for our HTTP request/response; essentially the logic of our app is written here.

Designing the app

Before we start designing, we need to know what we want for our apps. In our case, the app is a simple to-do list; this simple app shows us how to use CRUD operations using Django.

Models in Django

Django is an MVC framework, and this means we can define a "model", a physical representation of a table, and use it like a normal object.

To create a model in Django, we just need to create a class derived from `django.db.model.Models`, in which every attribute represents a database field. When we call a migration, Django creates a table for every table contained in the `models.py` file.

The next step is to design the model we want to use for our to-do list project; here we need to define a simple table to manage the task:

```
from django.db import models
from django.contrib.auth.models import User

class Task(models.Model):
    TASK_STATUS = (
        ('on_hold', 'On Hold'),
        ('complete', 'Conmplete'),
        ('in_progress', 'In Progress'),
        ('to_do', 'To Do'),
     )
    task = models.CharField(max_length=250)
    author = models.ForeignKey(User, related_name='ToDoList')
    body = models.TextField()
    created = models.DateTimeField(auto_now_add=True)
    updated = models.DateTimeField(auto_now=True)
    status = models.CharField(max_length=20, choices=TASK_STATUS,
default='to_do')

    class Meta:
        ordering = ('-created',)

    def __str__(self):
        return self.task
```

This code creates a model called `Task`. We can see that the creation of a model is simply a class creation with some added fields. For a better understanding of how the model is created, take a look at all the fields for the `Task` model:

- `task`: This field is the `Task` title. The field is a `CharField`; this is equivalent to a `VARCHAR` in an SQL model.
- `author`: This field is a `ForeignKey`. A `ForeignKey` is used to establish a connection between two tables; in this case we define a many-to-one relation. We tell Django that each task has a user and every user can have multiple tasks.
- `body`: This field is the body of the task; we use a `TEXT` field for the database.
- `created`: This field is a `datatime` field, is autopopulated by Django, and is used to save the time of the creation of the task.
- `updated`: This field is a `datetime` field, is autopopulated by Django, and is used to indicate when the task has updated.
- `status`: This field shows the status of a task; we use `choices`. This parameter allows the Django UI to make a choice from a specific value.

The model is ready; what we need to do now is activate the application and start the migration.

Configuring the application

In Django there is an easy way to configure the application on the server, allowing us to use it. To configure an application in Python, we need to modify the `setting.py` file in the `python_nano_book` folder:

```
INSTALLED_APPS = [
    'django.contrib.admin',
    'django.contrib.auth',
    'django.contrib.contenttypes',
    'django.contrib.sessions',
    'django.contrib.messages',
    'django.contrib.staticfiles',
    'python_book',
]
```

To add the app, just add a line in the `INSTALLED_APPS` settings. This tells the server there is a new app that needs to be launched in our system.

Migration

The application is now configured in Django; the next step is to run a migration to tell Django to create all the tables required to execute the application.

To create a migration, enter the root directory of our project and then execute the following command:

```
python manage.py makemigrations python_book
```

The output of the command is something like the following:

```
Migrations for 'python_book:
0001_initial.py:
Create model Task
```

The migration creates a file containing our migration. The file has a sequential number, in this case the file is `0001_initial.py`. If we take a look at the file, we can find all the code required to create the table in the database:

```
# -*- coding: utf-8 -*-
# Generated by Django 1.9.6 on 2016-05-16 16:07
from __future__ import unicode_literals

from django.conf import settings
from django.db import migrations, models
import django.db.models.deletion


class Migration(migrations.Migration):

    initial = True

    dependencies = [
        migrations.swappable_dependency(settings.AUTH_USER_MODEL),
    ]

    operations = [
        migrations.CreateModel(
            name='Task',
            fields=[
                ('id', models.AutoField(auto_created=True, primary_
key=True, serialize=False, verbose_name='ID')),
                ('task', models.CharField(max_length=250)),
                ('body', models.TextField()),
                ('created', models.DateTimeField(auto_now_add=True)),
                ('updated', models.DateTimeField(auto_now=True)),
                ('status', models.CharField(choices=[('on_hold', 'On
Hold'), ('complete', 'Conmplete'), ('in_progress', 'In Progress'),
('to_do', 'To Do')], default='to_do', max_length=20)),
                ('author', models.ForeignKey(on_delete=django.
db.models.deletion.CASCADE, related_name='ToDoList', to=settings.AUTH_
USER_MODEL)),
            ],
            options={
                'ordering': ('-created',),
            },
        ),
    ]
```

To execute the migration, we use the following command:

```
python manage.py sqlmigrate python_book 0001
```

This creates the SQL for execution on your system. After the creation of the SQL, we can launch the migration with the following command:

```
python manage.py migrate
```

The output of the migration is as follows:

```
Operations to perform:
  Apply all migrations: auth, admin, python_book, contenttypes, sessions
Running migrations:
  Rendering model states... DONE
  Applying contenttypes.0001_initial... OK
  Applying auth.0001_initial... OK
  Applying admin.0001_initial... OK
  Applying admin.0002_logentry_remove_auto_add... OK
  Applying contenttypes.0002_remove_content_type_name... OK
  Applying auth.0002_alter_permission_name_max_length... OK
  Applying auth.0003_alter_user_email_max_length... OK
  Applying auth.0004_alter_user_username_opts... OK
  Applying auth.0005_alter_user_last_login_null... OK
  Applying auth.0006_require_contenttypes_0002... OK
  Applying auth.0007_alter_validators_add_error_messages... OK
  Applying python_book.0001_initial... OK
  Applying sessions.0001_initial... OK
```

We have finally created the model for our site; what we need to do now is start to create the UI for our site.

Creating the first UI

Until now we have created the model for our site. What we need to do now is create the UI for our site. A UI is important because it's what our user will see—the HTML page displayed when a user tries to access our site. A UI is the first point of contact and iteration with the end user. A UI is a simple Python class used to intercept HTTP events, and this class is the `views.py` of our task application.

First of all, let's create a view that shows the full list of the `Task`:

```
from django.shortcuts import render
from .model import Task

def task_list(request):
  tasks = Task.published.all()
  return render(request,'blog/task/list.html',{'tasks': tasks})
```

This code simply creates an action to reply to a request HTTP action; in this case what the view will do is just render a list with all tasks. To show the page, we now need to add the URL patterns in the view, and to do so we need to add a URL pattern.

A URL pattern is composed of a regular expression, a view, and a name that allows the developer to name it in the project. To create the URL pattern, we need a file called `urls.py`. We can create this file in the `python_book` application folder:

```
from django.conf.urls import url
from . import views

urlpatterns = [
  # Task views
  url(r'^$', views.task_list, name='task_list'),
]
```

These simple lines of code show you how to create a URL; in this case we have a simple URL for finding the page, *Task List*. For every new page we add to the site, we need to add a URL line with the regular expression required to identify the page.

The next step is to add the address on the main server. To do so, update the `urls.py` file located in the main folder, in this case `python_nano_book`:

```
from django.conf.urls import url
from django.contrib import admin

urlpatterns = [
    url(r'^admin/', admin.site.urls),
  url(r'^task/', include('task.urls', namespace='task', app_
name='task')),
]
```

Django templates

The last step in showing tasks is creating an HTML template to display the tasks in a user-friendly way. To create a template, first create the folder structure for the template in our `python_book` application:

```
templates/
  task/
        base.html
        task/
                        list.html
```

The `base.html` file contains the basic definition of the site; it will look as follows:

```
{% load staticfiles %}
<!DOCTYPE html>
<html>
  <head>
    <title>{% block title %}{% endblock %}</title>
    <link href="{% static "css/task.css" %}" rel="stylesheet">
  </head>
  <body>
    <div id="content">
      {% block content %}
      {% endblock %}
    </div>
    <div id="sidebar">
      <h2>Task List</h2>
      <p>My Task List.</p>
    </div>
  </body>
</html>
```

Now we have the basic template, we can create the *Task List* page:

```
{% extends "blog/base.html" %}
  {% block title %}My Blog{% endblock %}
  {% block content %}
    <h1>My Blog</h1>
      {% for post in posts %}
      <h2>
      <a href="{{ post.get_absolute_url }}">
        {{ post.title }}
      </a>
      </h2>
      <p class="date">
        Published {{ post.publish }} by {{ post.author }}
      </p>
    {{ post.body|truncatewords:30|linebreaks }}
  {% endfor %}
{% endblock %}
```

We have now finished all the required configuration for our app. Now we just need to save and call our first Django app.

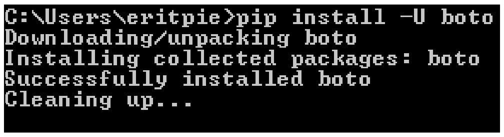
Using the AWS SDK

Another important area to consider is cloud development, and in Python we can interface our project with the cloud, in particular with AWS, using the library `boto` (`http://boto.cloudhackers.com/en/latest/`). This library allows us to interface our program with AWS and move our application to the cloud.

To install the `boto` library, we can simply use `pip`:

```
pip install -U boto
```

This will download and install the latest version of `boto` on your system:



```
C:\Users\eritpie>pip install -U boto
Downloading/unpacking boto
Installing collected packages: boto
Successfully installed boto
Cleaning up...
```

With the library installed, we can now start writing the code to interact with AWS. We need to configure the file to store the credentials necessary for access to the AWS server.

The next step is to create the file to store the credentials for access to the AWS on the system. In Linux/Unix systems, the path is `~/.boto`; in the case of Windows, it's necessary to indicate the path of the file in the `HOME` path.

The file must contain these lines:

```
[Credentials]
aws_access_key_id = {ACCESS KEY ID}
aws_secret_access_key = {SECRET ACCESS KEY}
```

With the configuration file created, we can now start writing the code to access AWS:

```
import boto
s3 = boto.connect_s3()
bucket = s3.create_bucket('your.bucket')
key = bucket.new_key('examples/python_book.csv')
key.set_contents_from_filename('/home/python_book.csv')
key.set_acl('public-read')
```

The previous code creates a new S3 bucket on your AWS domain. You can see that with just a few lines of code we can easily store data on AWS.

REST web services with Flask

Flask is a micro-framework for web development. Flask allows Python developers to write simple REST API web services in an easy way.

REST is an acronym for **REpresentational State Transfer**. In REST web services, developers intercept and use the HTTP states to manage the action of the API.

Because we want to design some APIs, we need to define the HTTP method and the associated URI.

HTTP method	URI
GET	`/python_book/v1.0/todo`

With the API defined, we can now start to write some code.

The first step is to install Flask. To do so we use the following command:

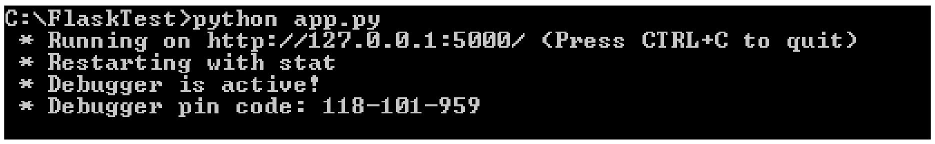
```
pip install flask
```

This will install Flask on our system. With Flask installed, we can now open our editor and start writing code:

```
from flask import Flask, jsonify

app = Flask(__name__)

todos = [
    {
        'id': 1,
        'title': 'Learn Python',
        'description': 'Learn Python',
        'done': False
    },
    {
        'id': 2,
        'title': 'Learn Flask',
        'description': 'Learn Flask',
        'done': False
    }
]

@app.route('/python_book/v1.0/todo', methods=['GET'])
def get_list():
    return jsonify({'todos': todos})

if __name__ == '__main__':
    app.run(debug=True)
```

Now we can see how a Flask program is a simple Python program. In this program, we use the annotation `@app.route('/python_book/v1.0/todo', methods=['GET'])` to define a new route. This route is the endpoint of our REST API.

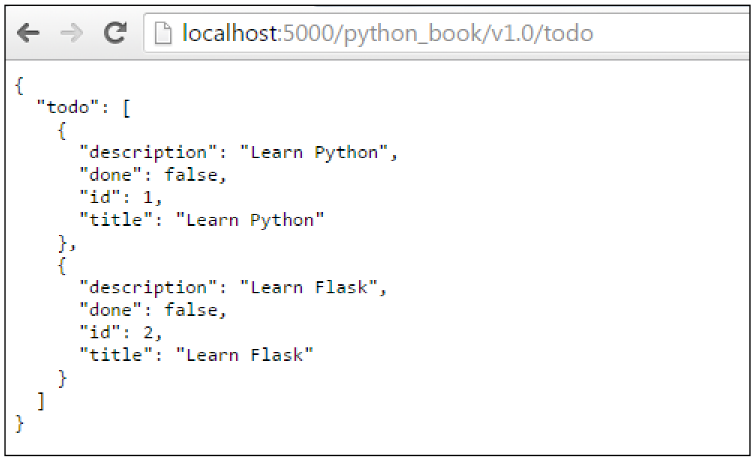
Save the file with the name `app.py` and launch it; this will start the server and allow Flask to receive the HTTP call:



```
C:\FlaskTest>python app.py
 * Running on http://127.0.0.1:5000/ (Press CTRL+C to quit)
 * Restarting with stat
 * Debugger is active!
 * Debugger pin code: 118-101-959
```

It's also possible to call the API via your browser; the address is: `http://localhost:5000/python_book/v1.0/todo`.

The result is shown in the following screenshot:



localhost:5000/python_book/v1.0/todo

```
{
  "todo": [
    {
      "description": "Learn Python",
      "done": false,
      "id": 1,
      "title": "Learn Python"
    },
    {
      "description": "Learn Flask",
      "done": false,
      "id": 2,
      "title": "Learn Flask"
    }
  ]
}
```

We have now learned how to start using Flask to design our REST web API. Flask is a very lightweight framework, but at the same time provides the power to create reusable REST APIs.

3
Data Visualization with Python

In this section, we will talk about the use of Python for data visualization. Python has a powerful project for just that: **Jupyter Notebook**. If you want try it out, visit the official site at `https://try.jupyter.org/`.

Data analysis is the process of inspecting, clearing, and transforming data with the goal of discovering more important information.

Data analysis covers many different areas, from business to human sciences, and has become more important every day when you consider the increase in jobs created for the purpose of data analysis.

Installing Jupyter

The first step in starting your data visualization project is installing the software. The simplest way to install the software is to use `pip`; the command is as follows:

```
pip3 install jupyter
```

In our case, we are using Python version 2.7, and the command is as follows:

```
pip install jupyter
```

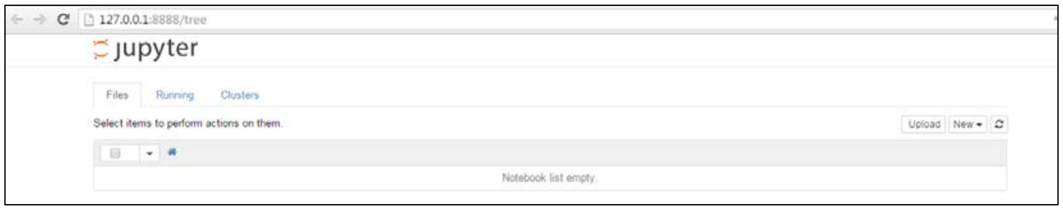
When the installation is complete, we can start the server using the following command:

```
jupyter notebook
```

When the server has started, we will see output like the following:

```
c:\python34\lib\site-packages\widgetsnbextension\__init__.py:30:
UserWarning: To
 use the jupyter-js-widgets nbextension, you'll need to update
    the Jupyter notebook to version 4.2 or later.
  the Jupyter notebook to version 4.2 or later.""")
[W 21:21:23.430 NotebookApp] Cannot bind to localhost, using 127.0.0.1 as
defaul
t ip
    [Errno 11004] getaddrinfo failed
[I 21:21:23.430 NotebookApp] Serving notebooks from local directory: C:\
Users\User
[I 21:21:23.446 NotebookApp] 0 active kernels
[I 21:21:23.446 NotebookApp] The Jupyter Notebook is running at:
http://127.0.0.
1:8888/
[I 21:21:23.446 NotebookApp] Use Control-C to stop this server and shut
down all
 kernels (twice to skip confirmation).
```

The Jupyter Notebook server is now running on our local address. This command will also open a browser page showing a tree of the actual folder where we will execute the command:



Congratulations, your server is now ready to use! First, before we move on and create your notebook, it's best that we describe what a notebook actually is and what the Jupyter architecture involves.

Description of Jupyter Notebook

Jupyter Notebook is an interactive computing environment that enables the user to create a "notebook".

Every notebook can include:

- Live code
- Interactive widgets
- Narrative text
- Equations
- Images
- Video

A notebook can be exported and shared in different formats; every notebook is a complete and self-contained record computation.

The official documentation for Jupyter Notebook is available at `http://jupyter.readthedocs.io/en/latest/index.html`. This documentation contains basic information about Jupyter and shows you how to create notebooks for use in different languages.

Jupyter Notebook combines three components:

- **Web application**: An interactive application used for writing and running the code for the notebook
- **Kernels**: The separate process started in order to run the users' code
- **Documents**: Self-contained documents contain a representation of all content visible in the notebook

matplotlib

matplotlib is a 2D Python library used for plotting 2D charts. To continue with our notebook, we first need to install this library.

This library is integrated with iPython and this enables us to create very powerful graphs for our notebook.

To install matplotlib, we must first satisfy some dependencies:

- setuptools
- NumPy 1.6 or later
- dateutil 1.1 or later
- pyparsing
- libpng 1.2 or later

- pytz
- FreeType 2.3 or later
- Cycler 0.9 or later

To install matplotlib, use the following command:

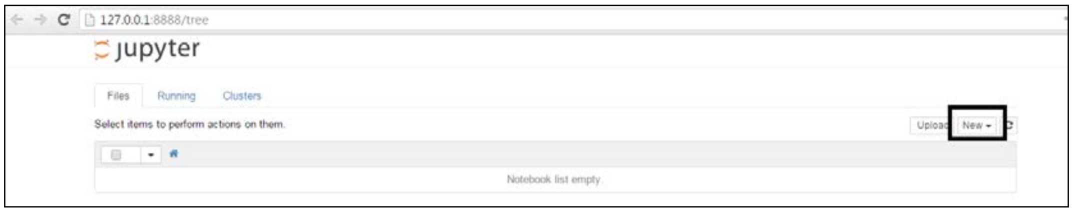
```
pip install matplotlib
```

This installs all the missing dependencies, if you didn't have them installed previously, and installs matplotlib. Now we have matplotlib installed, we can start to write our first notebook.

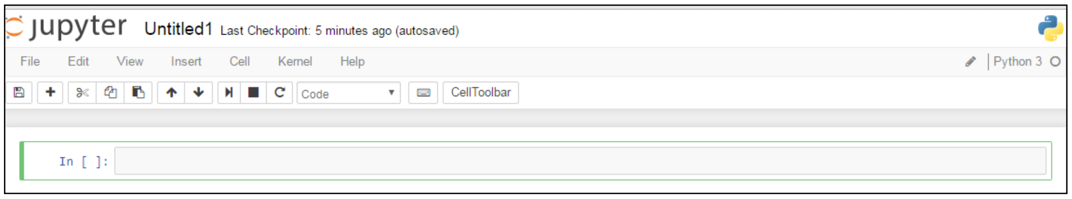
Creating our first notebook

A notebook is a set of cells used to define the instruction of the code we want to execute. Every cell contains Python commands, and these commands can be executed to make a specific program.

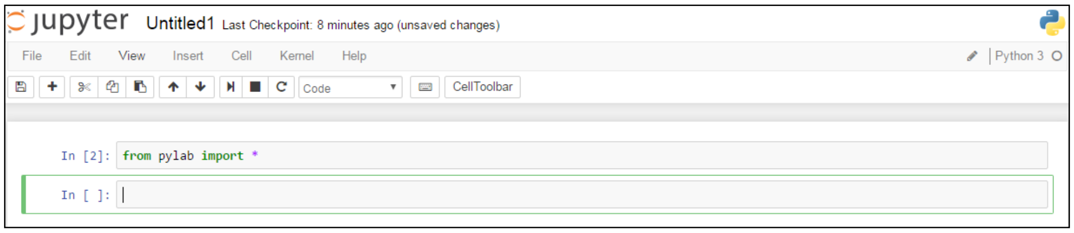
To create a new notebook, we click on the **New** menu button on the left-hand side of the Jupyter server webpage:



This creates a new notebook; a new notebook starts with a simple white cell:



Now we can start to create our first notebook, so first import the matplotlib library:


jupyter Untitled1 Last Checkpoint: 8 minutes ago (unsaved changes)
File Edit View Insert Cell Kernel Help
Python 3
Code
CellToolbar

```
In [2]: from pylab import *
In [ ]:
```

To insert and process the line, press *Alt* + *Enter* in the first line to import the matplotlib library. Now we can start to insert simple Python commands to plot our graph.

First, we need to define two variables, `x` and `y`:

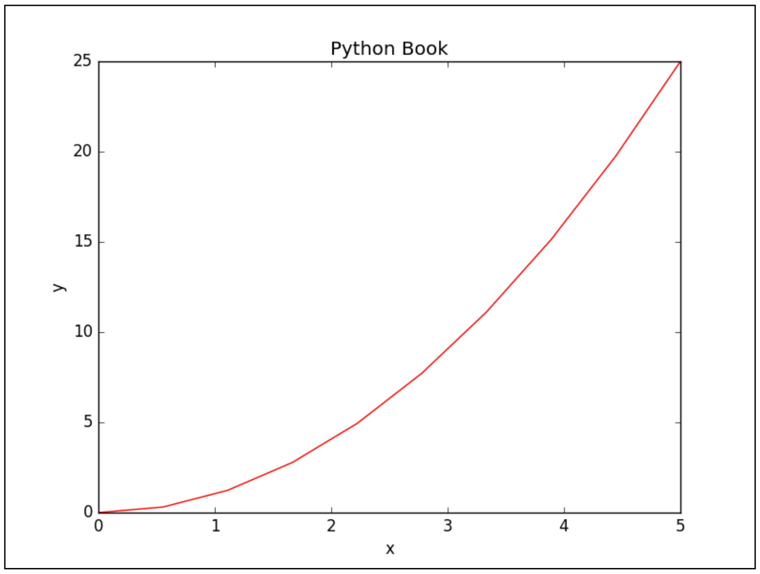
```
x = linspace(0, 10, 20)
y = x ** 2
```

This code defines a linear spaced vector; this library is not from Python but from **MathLab**—matplotlib defines a portion of MathLab in Python.

Press *Alt* + *Enter* to create a new cell and insert the code to generate the graph:

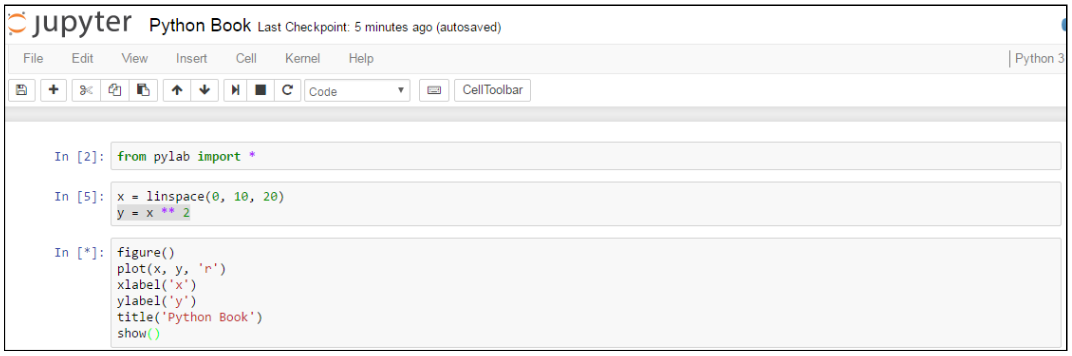
```
figure()
plot(x, y, 'r')
xlabel('x')
ylabel('y')
title('Python Book')
show()
```

Press *Alt* + *Enter* and execute the code; this will design a graph such as the one shown in the following figure:



Save the file with the name `graph.png` in the folder we executed the server from.

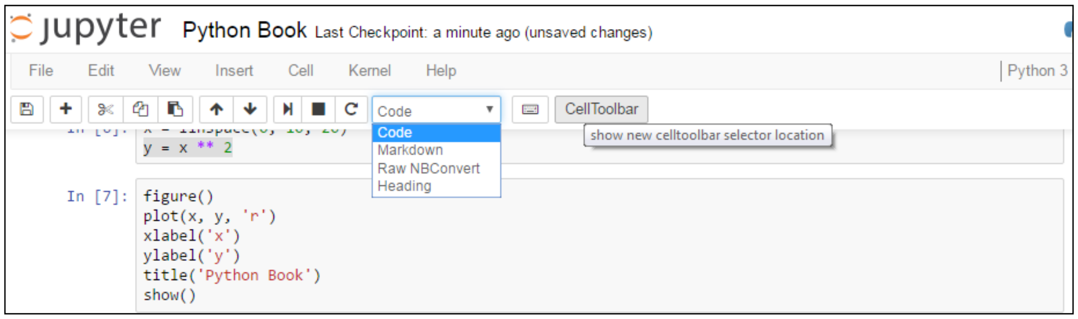
What we need to do now is save our notebook and give it a useful name; to do that, just click on the Save icon. This will save the notebook as `untitled`; to change the name of the notebook, just click on the title and rename it, in this case to `Python Book`. Our notebook should look as follows:



```
In [2]: from pylab import *

In [5]: x = linspace(0, 10, 20)
        y = x ** 2

In [*]: figure()
        plot(x, y, 'r')
        xlabel('x')
        ylabel('y')
        title('Python Book')
        show()
```

Other kinds of cell

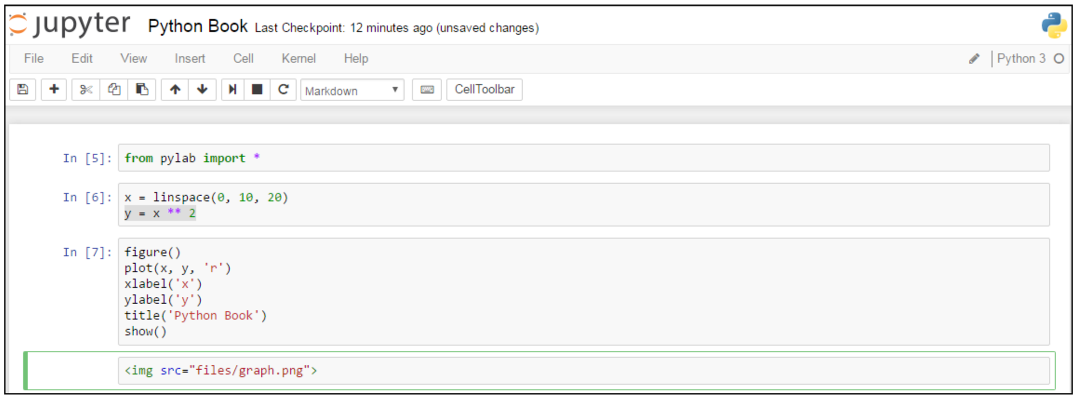
In Jupyter, a notebook is not only a set of cells with code, it's also possible to change the kind of cell. This is useful when we want to create a complete document and export the notebook. To change the cell, just click on the **Cell** drop-down tab and change the type of cell; we can use different cells to create a more complex notebook:


jupyter
Python Book Last Checkpoint: a minute ago (unsaved changes)
File
Edit
View
Insert
Cell
Kernel
Help
Python 3
Code
Code
Markdown
Raw NBConvert
Heading
CellToolbar
show new celltoolbar selector location

```
y = x ** 2
```

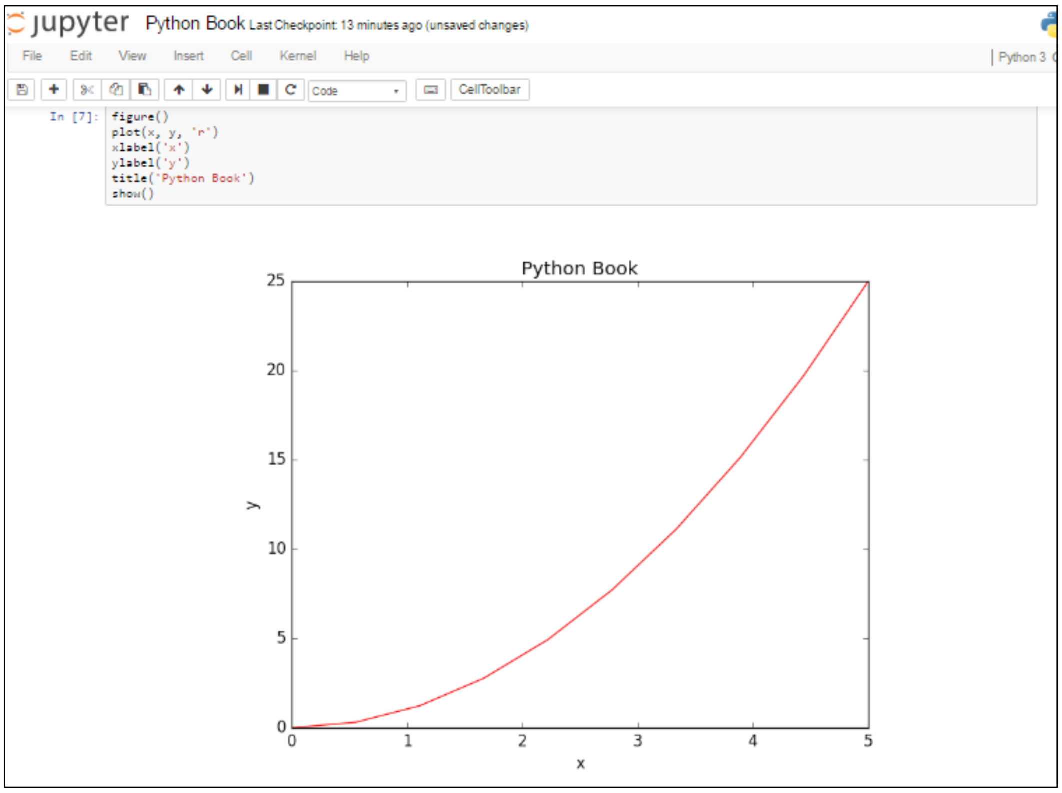
```
In [7]: figure()
        plot(x, y, 'r')
        xlabel('x')
        ylabel('y')
        title('Python Book')
        show()
```

The **Markdown** field is used to insert text. When we run this cell, the code is passed on the kernel and is automatically converted to HTML.



```
In [5]: from pylab import *

In [6]: x = linspace(0, 10, 20)
        y = x ** 2

In [7]: figure()
        plot(x, y, 'r')
        xlabel('x')
        ylabel('y')
        title('Python Book')
        show()

        <img src="files/graph.png">
```

For example, if we want to include the graph created previously, just create a markdown cell and insert the code `<img src=files/graph.png>`. Now, execute the cell and see what happens:



```
In [7]: figure()
        plot(x, y, 'r')
        xlabel('x')
        ylabel('y')
        title('Python Book')
        show()
```

Congratulations! You have finally created your first notebook! This is just the start of a fantastic journey in data analysis. We can look at a notebook as essentially a mix of code, sentences, and graphs and all this information is used to translate data into a "human language." Normally, data analysis is always connected with big data, and together they give a complete picture of the data.

4
Introduction to DevOps

DevOps is a new discipline in IT, and its aim is to create a bridge from the developer to the operations team.

DevOps is a cultural change for a company, and every company adopts DevOps in a different way, but essentially all companies eventually embrace the cultural change from DevOps. In DevOps, both areas of a company work together to eliminate the disconnect between operations and development.

When DevOps is used in the right manner, it removes the bottlenecks that normally slow down the process of releasing software with maximal quality. New features and updates are released quickly, and in every sprint we can have more than one software release; if DevOps works properly, it is possible to have more than one release per day. To remove bottlenecks, the developer must work closely with the operations team; this way it's possible to manage the problem as soon as it appears and speed up the process for releasing software in production.

The challenge of DevOps

DevOps is a cultural change for a company, as it covers different areas of the company, and the final goal of DevOps is to reduce the time it takes to market a new feature with a high-quality release. To reach this goal, DevOps adopts a set of tools and practices:

- **Metrics**: To check the number of errors in the software or to respond faster to a new error.
- **Monitoring**: To constantly check the functionality of the software in production.
- **Continuous integration**: Every time a new feature is added to the repository branch, a complete set of tests is executed; in this way, it is possible to reduce the number of errors in every release.

- **Continuous delivery**: This releases small parts of the software to reduce the possibility of introducing a new bug in the system.
- **IaaS Code**: Infrastructure as Code is a practice used to automate the release and the installation of a new server in the architecture. In particular, with the diffusion of cloud architecture, it's important to release a new server quickly. To reach this goal in DevOps, we can use tools for configuration management, such as Chef.

All these practices are not only technical changes, but managerial and procedural changes as well; for example to first release the software to the entire world, a company uses DevOps to release the software to a restricted number of users or servers. This practice is called a "canary server;" in this way a restricted number of users add another layer of testing on the software and their feedback can be used to improve the quality of the software.

Working with DevOps

In this introduction, we have said that DevOps is essentially a change in company culture; this change embraces all areas of the company and promotes the notion of communication and transparency between every single area of the company. To do that, the team responsible for the DevOps transformation must start to create their own tools and use other tools to manage and monitor the production software.

Another area where DevOps is important is automation—with DevOps, it's possible to write software to create some automatic procedure necessary for fixing issues around different servers.

In Python, there is a library we can use to execute commands around a server; it's called **Fabric**. But to show the benefits of DevOps, we need to touch on an area more important to every company: **continuous release**. We have tried to create a chain for continuous release by using a mix of Fabric + SaltStack + Git to create our own tool. Of course, this is just an example and for a complete description of that we would probably need another book, but it is a good example to show how DevOps works.

What is Fabric?

Fabric is a library and set of command line tools that execute commands on a remote server. Fabric uses an SSH connection to connect to the server and then executes commands against it.

A typical use of Fabric is composing using Python modules and executing by the Fabric command line. This module executes commands against the SO and helps to make some operations automatic.

Installing Fabric

The best way to install Fabric is to use `pip`:

```
pip install fabric
```

If you are installing it on an Ubuntu server, you can use an alternative command:

```
sudo apt-get install fabric
```

When Fabric is installed, we can start to write our **fabfiles**. A fabfile is a file that can be used by the Fabric command line to execute an operation on the system.

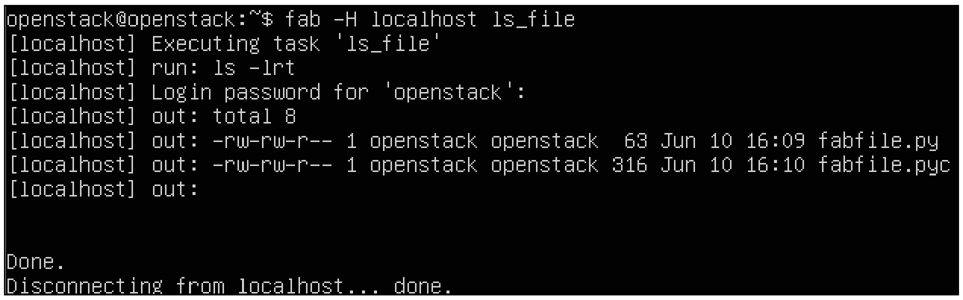
For our first fabric file, let's start with a simple command to show all directories on our system:

```
from fabric.api import run
def ls_file():
    run('ls -lrt')
```

The file must be called `fabfile.py`; to execute the script, use this command:

```
fab -H localhost ls_file
```

This command produces the following output:



```
openstack@openstack:~$ fab -H localhost ls_file
[localhost] Executing task 'ls_file'
[localhost] run: ls -lrt
[localhost] Login password for 'openstack':
[localhost] out: total 8
[localhost] out: -rw-rw-r-- 1 openstack openstack  63 Jun 10 16:09 fabfile.py
[localhost] out: -rw-rw-r-- 1 openstack openstack 316 Jun 10 16:10 fabfile.pyc
[localhost] out:


Done.
Disconnecting from localhost... done.
```

Fabric opens an SSH connection with the server indicated; in this case it's `localhost`, and executes the method `ls_file`. The method executes the API `run` for calling the `ls -lrt` command locally. The parameter `-H` tells Fabric we still define the host, and with this parameter we can send a comma separated list to the host.

The Fabric script

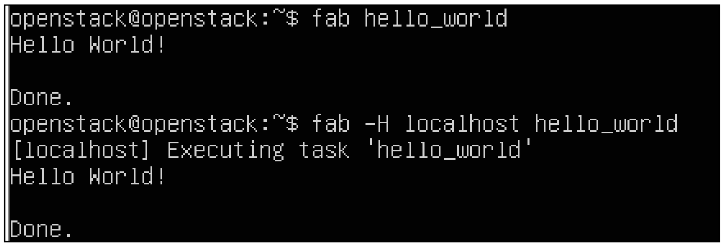
Firstly, in order to write more complex scripts, we need to better understand how to write a Fabric script. So, let's start to write a simple Hello World program:

```
def hello_world():
    print("Hello World!")
```

We can execute the script with this command:

```
fab -H localhost hello_world
```

The output of the program is as follows:



```
openstack@openstack:~$ fab hello_world
Hello World!

Done.
openstack@openstack:~$ fab -H localhost hello_world
[localhost] Executing task 'hello_world'
Hello World!

Done.
```

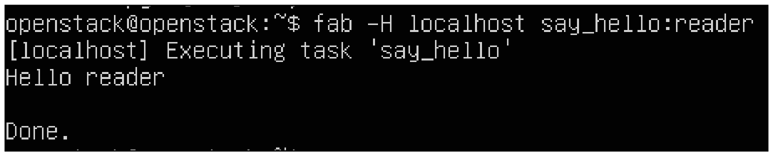
Some scripts require parameters to work; to create a method with a parameter, follow the same rules as used in Python:

```
def say_hello(name):
    print("Hello %s" % name)
```

To send the parameter, the command syntax is as follows:

```
fab -H localhost say_hello:reader
```

As you can see, after the method we added `:` and after that the value of the parameter. The output is as follows:



```
openstack@openstack:~$ fab -H localhost say_hello:reader
[localhost] Executing task 'say_hello'
Hello reader

Done.
```

We can now see how to send a parameter to a Fabric script, and the next step is to describe the use of Fabric for DevOps.

Using Fabric for DevOps

As we mentioned at the start of this section, the aim of DevOps is to help to reduce the gap between developer and operations. In order to do so, DevOps needs to establish some common practices to automate common and repetitive procedures, and to help encourage continuous development.

The Fabric API

Fabric offers a lot of features for SSH and sysadmin operations, and in Fabric we have different levels of API:

- **The core API**: This is the building block of Fabric; this set of APIs is the main functionality of Fabric
- **The contrib API**: This API is an extension build to improve the functionality of Fabric

The core API

The core API is all the APIs that constitute the building blocks of Fabric; in this API we can find the API for `run` and `sudo`. We can split the API into different areas:

- Color output functions
- Context managers
- Decorators
- Network
- Operations

Due to the scope of this book, we have only described the most commonly used APIs in Fabric.

Color output functions

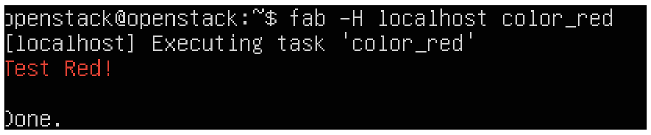
This API is used to color the output of a function, such as text, and this is helpful, for example, when we need to highlight an error or a warning during the execution of a script:

```
from fabric.colors import red


def color_red():
    print(red("Test red!"))
```

The call for a function is always the same; the result is red text:

```
fab -H localhost color_red
```

The output is as follows:



```
openstack@openstack:~$ fab -H localhost color_red
[localhost] Executing task 'color_red'
Test Red!

Done.
```

Context managers

The context managers API is used for the `with` statement, for example:

```
with cd('/var/www')
    run('ls -lrt')
```

It is possible to identify some context manager APIs:

- `cd`: Used to open a remote directory; in this case, if we want to open a local directory we can use `lcd`
- `char_buffered`: Only available on Unix systems
- `hide`: Hides the output passed with parameters on the function
- `lcd`: Used to move into a local directory
- `path`: Adds the specified path to the actual path of the system

Decorators

Often a decorator is used in a file. A decorator is similar to an annotation and in Fabric there are some defined decorators:

- `hosts`: Defines the host to execute the function
- `parallel`: Forces the function to be executed in parallel and not sequentially
- `roles`: Defines the roles to be used to execute the function
- `runs_once`: Tells the command line to run the function just once; if we use the `parallel` decorator, `runs_once` does not work
- `serial`: Forces the function to run only sequentially, never in parallel

Networks

This set of APIs defines the core to work with the network, and these are probably the most important APIs to know. This set of APIs defines a large number of functions that should be used for working with the network key:

- `disconnect_all`: Closes all network connections
- `HostConnectionCache`: A dictionary subclass that allows caching of the host connections/client
- `connect`: Opens a connection with the specific server
- `denormalize`: Strips out the default value for the host string
- `get_gateway`: Creates and returns a gateway socket if it is necessary
- `join_host_strings`: Turns user, host, and port into a string like `user@host:port`
- `key_filenames`: Returns the list of all key files present on `env.host_string`

We have presented just the most commonly used APIs for networking, and will now show you another set of APIs necessary for general use.

Operations

This set of APIs is used to execute operations on the remote/local system, for example, to download a file or open a remote shell.

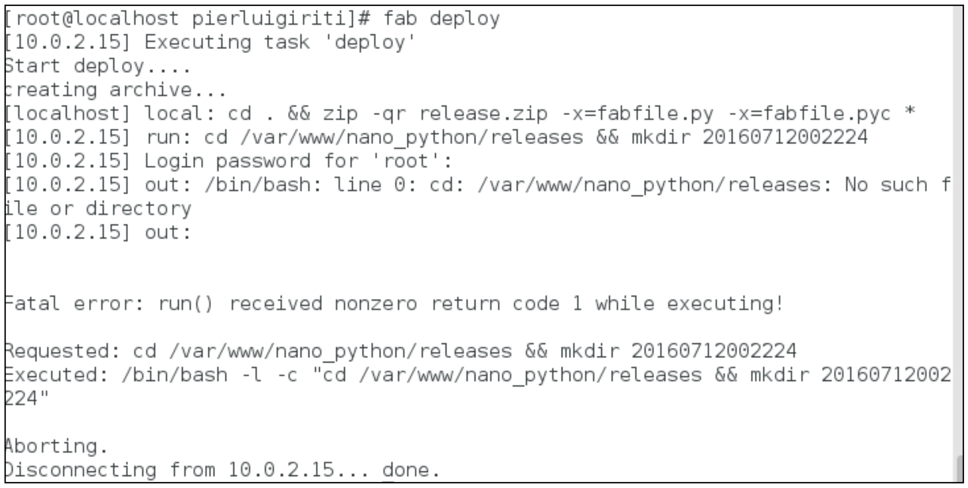
- `get`: Is used to download files from the remote host
- `local`: Runs command in the local system
- `open_shell`: Opens a shell on the remote system
- `prompt`: Returns a text and returns the input
- `put`: Uploads one or more files on the remote system
- `reboot`: Is used to reboot the remote system

Mixing it all together

Until now, we have just shown you the Fabric API, so for a better understanding of how Fabric works, we now need to write code for deploying software on the system:

```
from fabric.api import local, run, env, put
import os, time

env.hosts = ['10.0.2.15']

env.archive_source = '.'

env.archive_name = 'release'

env.deploy_project_root = '/var/www/nano_python/'

env.deploy_release_dir = 'releases'

env.deploy_current_dir = 'current'

def upload_archive():
  print('creating archive...')
  local('cd %s && zip -qr %s.zip -x=fabfile.py -x=fabfile.pyc *' \
    % (env.archive_source, env.archive_name))

  deploy_timestring = time.strftime("%Y%m%d%H%M%S")
  run('cd %s && mkdir %s' % (env.deploy_project_root + \
    env.deploy_release_dir, deploy_timestring))

  env.deploy_full_path = env.deploy_project_root + \
    env.deploy_release_dir + '/' + deploy_timestring
  put(env.archive_name+'.zip', env.deploy_full_path)
  run('cd %s && unzip -q %s.zip -d . && rm %s.zip' \
    % (env.deploy_full_path, env.archive_name, env.archive_name))

def create_symlink():
  print('creating symlink to uploaded code...')
  run('rm -f %s' % env.deploy_project_root + env.deploy_current_dir)
  run('ln -s %s %s' % (env.deploy_full_path, env.deploy_project_root + \
    env.deploy_current_dir))


def cleanup():
  print('cleanup...')
  local('rm -rf %s.zip' % env.archive_name)


def deploy():
    print('Start deploy....')
  upload_archive()
  create_symlink()
  cleanup()
  print('deploy complete!')
```

The output will be as follows:



```
[root@localhost pierluigiriti]# fab deploy
[10.0.2.15] Executing task 'deploy'
Start deploy....
creating archive...
[localhost] local: cd . && zip -qr release.zip -x=fabfile.py -x=fabfile.pyc *
[10.0.2.15] run: cd /var/www/nano_python/releases && mkdir 20160712002224
[10.0.2.15] Login password for 'root':
[10.0.2.15] out: /bin/bash: line 0: cd: /var/www/nano_python/releases: No such f
ile or directory
[10.0.2.15] out:


Fatal error: run() received nonzero return code 1 while executing!

Requested: cd /var/www/nano_python/releases && mkdir 20160712002224
Executed: /bin/bash -l -c "cd /var/www/nano_python/releases && mkdir 20160712002
224"

Aborting.
Disconnecting from 10.0.2.15... done.
```

This simple script manages the installation in a server for a web application. The script defines some variables using the `env` API of Fabric, and after that defines some methods for uploading the file (`upload_archive`), creating the symlink, and cleanup. This method uses the Fabric APIs for interaction with the operating system.

We have talked about Fabric as a library used for automating more operations on the host system, but with Fabric it is also possible to download the code directly to the repo source, such as Git, and compile the code.

SaltStack

SaltStack is a configuration management tool, like Chef or Ansible, and allows the user to create what we will define as "Infrastructure as Code." By this definition, we mean all software that can be used to create the software used for defining our infrastructure. This tool is very important in DevOps because it reduces the time-to-market and the complexity connected with the release of a new feature.

Because it reduces the time to create and configure a new server, since the software does all the jobs, we also reduce the potential for "human error" when working with the configuration of the system, libraries, or installation. All this effort translates into better stability throughout the entire system.

Installing SaltStack

To install SaltStack, I would suggest that you use Linux, in my case Fedora, on a virtual machine. In a Fedora environment, the fastest way to install SaltStack is to use `yum`:

```
yum install salt-master
```

This command installs `salt-master`, the server. To work with Salt, we need a client. Salt uses `salt-minion`, the client used to connect to the server.

The command for installing it is as follows:

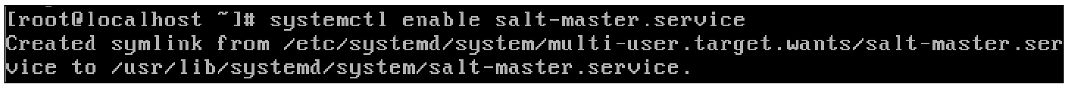
```
yum install salt-minion
```

When the client and the server are installed, the next step is to allow the service on the system and start it.

To have a master directly at boot time, we use the following command:

```
systemctl enable salt-master.service
```

The following screenshot represents this:



```
[root@localhost ~]# systemctl enable salt-master.service
Created symlink from /etc/systemd/system/multi-user.target.wants/salt-master.ser
vice to /usr/lib/systemd/system/salt-master.service.
```

When the service is configured to start at boot time, we can start it without restarting the machine. To do so the command is as follows:

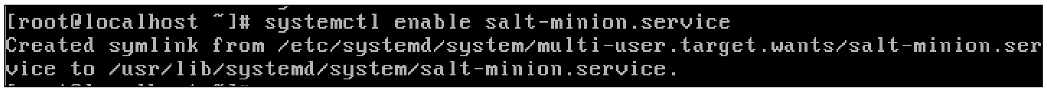
```
systemctl start salt-master.service
```

This starts the Salt service on the machine.

We can now use the same steps for activating the service on `salt-minion`; the first command is used to start the client at boot:

```
systemctl enable salt-minion.service
```

The following screenshot shows this:



```
[root@localhost ~]# systemctl enable salt-minion.service
Created symlink from /etc/systemd/system/multi-user.target.wants/salt-minion.ser
vice to /usr/lib/systemd/system/salt-minion.service.
```

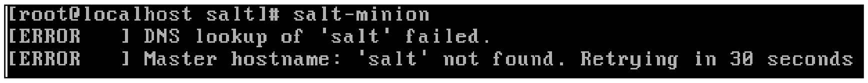
And finally, we can start the client on the system:

```
systemctl start salt-minion.service
```

Now that the Salt client and server are installed on our system, the next step is to configure Salt and play with it.

Configuring the client

If we haven't configured the client and server and we try to use the command to get the configuration from the server, we will see an error as follows:

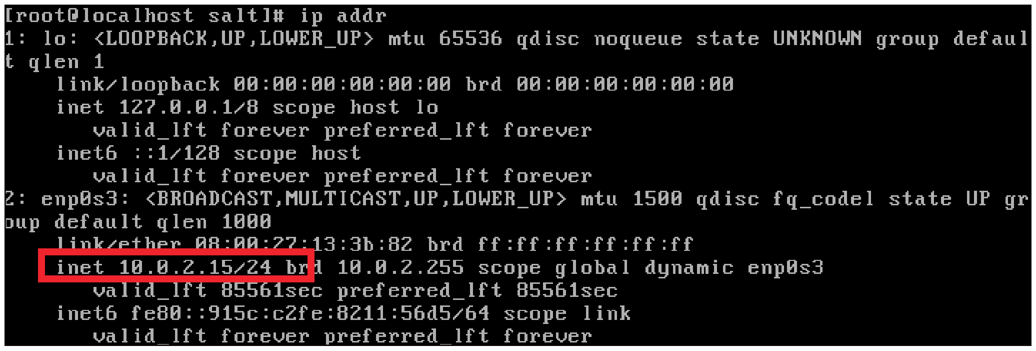


```
[root@localhost salt]# salt-minion
[ERROR   ] DNS lookup of 'salt' failed.
[ERROR   ] Master hostname: 'salt' not found. Retrying in 30 seconds
```

This error identifies the wrong configuration of the client for communicating with the server. We need to configure the server and the client, and to do that we need to first identify the IP address of the machine. In this case, because we have Fedora, we use the following command:

```
ip addr
```

This shows the output of the network:



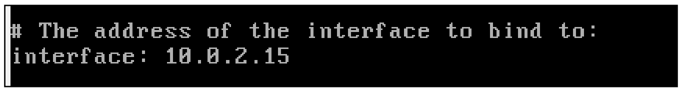
```
[root@localhost salt]# ip addr
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN group defaul
t qlen 1
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
    inet 127.0.0.1/8 scope host lo
       valid_lft forever preferred_lft forever
    inet6 ::1/128 scope host
       valid_lft forever preferred_lft forever
2: enp0s3: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc fq_codel state UP gr
oup default qlen 1000
    link/ether 08:00:27:13:3b:82 brd ff:ff:ff:ff:ff:ff
    inet 10.0.2.15/24 brd 10.0.2.255 scope global dynamic enp0s3
       valid_lft 85561sec preferred_lft 85561sec
    inet6 fe80::915c:c2fe:8211:56d5/64 scope link
       valid_lft forever preferred_lft forever
```

We can see that the address actually used by the machine in this case is `10.0.2.15`.

The next step is to configure the Salt server and minion to work together. To configure Salt, we need to move it to the folder `/etc/salt`, and in this folder we need to change the configuration file for both the server and minion, starting with the server.

Inside this folder, we have two files, `minion` and `master`; these files are used to configure the server and the minion in Salt.

The first file we go to modify is the `master` file, so edit the file with the command `vi` and identify the `interface` line. Replace the actual line with the line with the IP address of the machine. By default, Salt opens all IPs with the value `0.0.0.0`; we need to change the line to the value of our IP address, in this case `10.0.2.15`:

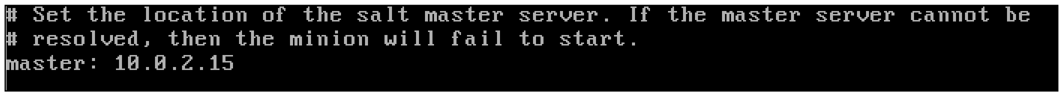


```
# The address of the interface to bind to:
interface: 10.0.2.15
```

Save the file and restart the server with the following command:

```
systemctl restart salt-master.service
```

With the server now configured, we can now configure the client by configuring the `minion` file, so open the `minion` file and configure the `master` line:



```
# Set the location of the salt master server. If the master server cannot be
# resolved, then the minion will fail to start.
master: 10.0.2.15
```

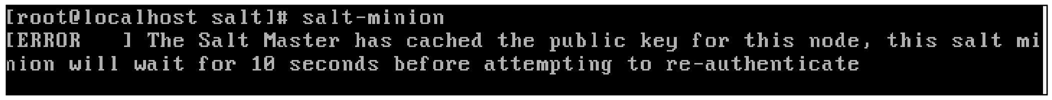
The IP address we add is the IP address of the server; after the configuration, simply restart the client using the following command:

```
systemctl restart salt-minion.service
```

Now we have configured the server to talk with the client, we can try the configuration using the following command:

```
salt-minion
```

This command shows the following error:

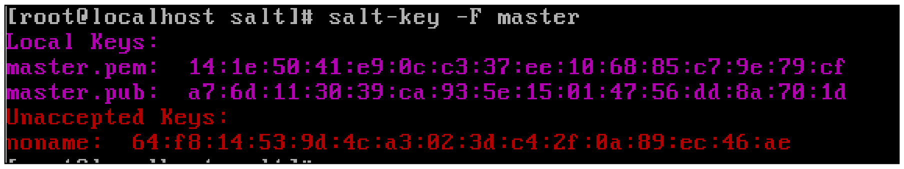


```
[root@localhost salt]# salt-minion
[ERROR   ] The Salt Master has cached the public key for this node, this salt mi
nion will wait for 10 seconds before attempting to re-authenticate
```

This error has occurred because the configuration key for the master and the client has not been configured. To configure the key, use the following command:

```
salt-key -F master
```

This shows the key we need to configure on the minion side:



```
[root@localhost salt]# salt-key -F master
Local Keys:
master.pem:  14:1e:50:41:e9:0c:c3:37:ee:10:68:85:c7:9e:79:cf
master.pub:  a7:6d:11:30:39:ca:93:5e:15:01:47:56:dd:8a:70:1d
Unaccepted Keys:
noname:  64:f8:14:53:9d:4c:a3:02:3d:c4:2f:0a:89:ec:46:ae
```

To configure the key, we need to add the key in the `master_finger` section of our `minion` configuration file. After we add the key, we can restart the minion client:

```
systemctl restart salt-minion.service
```

When all these steps are complete, we need to now check the fingerprint to see if the system is correctly configured.

To check that, we can use the following command:

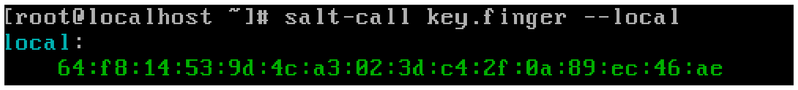
```
salt-key -L
```

This shows all the keys actually accepted on the system, but because we have one single node configuration, our local key has probably not been accepted.

The step for checking our local key is simple: first show all keys using the following command:

```
salt-call key.finger --local
```

This command shows the local key on the system:

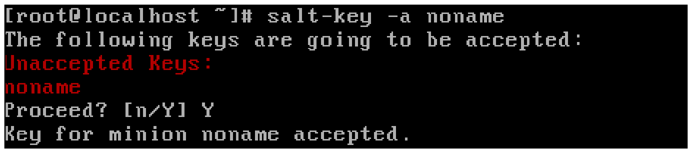


```
[root@localhost ~]# salt-call key.finger --local
local:
    64:f8:14:53:9d:4c:a3:02:3d:c4:2f:0a:89:ec:46:ae
```

If the key we have matches the unaccepted key on the system, we can accept this key using the following command:

```
salt-key -A <domain name>
```

This is also shown in the following screenshot:

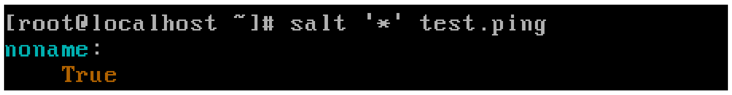


```
[root@localhost ~]# salt-key -a noname
The following keys are going to be accepted:
Unaccepted Keys:
noname
Proceed? [n/Y] Y
Key for minion noname accepted.
```

Now the system is ready to work, we can see if the master will talk with the minion using the following command:

```
salt '*' test.ping
```

This command executes a ping to all minions registered on the master and returns the result of the ping:



```
[root@localhost ~]# salt '*' test.ping
noname:
    True
```

Congratulations! You have just configured Salt on your system! Now we have a basic infrastructure to put in place a system for continuous delivery.

Salt state

Before we move on to the next step, it is important to understand how Salt configures the system. In Salt, a "state" is how Salt manages the configuration. We can write our "state" to manage and configure the system.

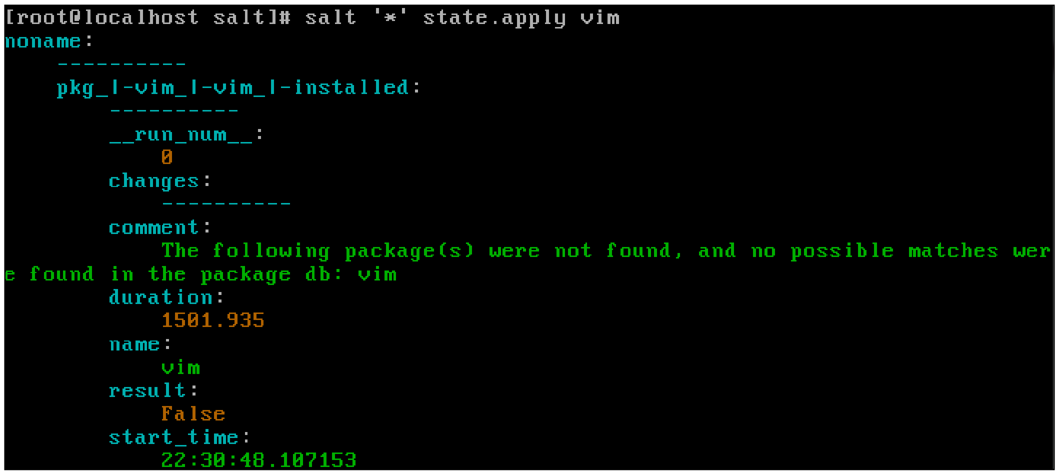
A state is written in the path `/srv/salt/` in the master node and is written in YAML with the extension `sls`. This is easy and simple to read; for example we can write a simple state to install `vim` on the system as follows:

```
vim:
      pkg.installed
```

This simple code creates a state for installing `vim` on the minion, so save the file with the name `vim.sls`. To execute the script, we need a simple call:

```
salt '*' state.apply vim
```

The result is shown in the following screenshot:



```
[root@localhost salt]# salt '*' state.apply vim
noname:
    ----------
    pkg_|-vim_|-vim_|-installed:
        ----------
        __run_num__:
            0
        changes:
            ----------
        comment:
            The following package(s) were not found, and no possible matches wer
e found in the package db: vim
        duration:
            1501.935
        name:
            vim
        result:
            False
        start_time:
            22:30:48.107153
```

Okay, so we now have a complete idea of how Salt works and how this software can help create the basic infrastructure of our code. This is just the start. If you want to continue learning more about Salt, I would recommend going through the official site at `https://docs.saltstack.com/en/latest/`.

We have now seen two powerful tools for deploying and configuration management, and this is only the start of our journey in DevOps. DevOps is more than just a technical skill, it is a set of practices that will drive the entire company process.

5 Cheat Sheet

In this section, we will show you some of the most common reserved words and syntax used in Python and in some of the tools we've used, such as Django, Salt, and Fabric.

The reader can use this section as a reference for quickly finding a command or a direction where more information about commands can be found.

Python

Let's get started with the reserved words.

Reserved words

The following words can be used in Python. These are reserved for the internal use of the language, and all Python keywords are lowercase only. The following table shows a list of reserved words:

`and`	`exec`	`not`
`assert`	`Finally`	`or`
`Break`	`for`	`Pass`
`Class`	`From`	`Print`
`Continue`	`global`	`raise`
`def`	`if`	`return`
`del`	`Import`	`Try`
`elif`	`In`	`While`
`else`	`Is`	`With`
`except`	`lambda`	`yield`

Basic types

`int`	`float`	`bool`
`str`	`bytes`	

Ordered sequences

These kinds of sequence have fast index access and repeatable values:

`[ ]` list	`( )` tuple

Immutable sequences

`()` tuple	`str` immutable sequence of `char`	`bytes` immutable sequence of `byte`

Key containers

`dict` dictionary	`set/frozenset`

Identifiers

A Python identifier is a name used to identify a variable, function, class, module, or other object. An identifier starts with a letter from A to Z, a to z, or an underscore (_) followed by zero or more letters, underscores, and digits (0 to 9). Python is a "case-sensitive language;" this means that `Test` and `test` are two different identifiers in Python.

An identifier in Python is used to define:

- Class
- Variable
- Functions
- Modules
- Names

Variables

In Python, a variable can consist of an identifier on the left-hand side, followed by the equals symbol (=) and a value on the right-hand side.

For example:

```
test = 1
```

List methods

`append(value)`	`pop(value)`	`count(value)`
`remove`	`extend(list)`	`reverse()`
`index(value)`	`sort()`	`insert(position, value)`

String methods

`capitalize()`	`lstrip()`	`center(width)`
`partition(separator)`	`count(substring, start, stop)`	`replace(old, new)`
`decode()`	`rfind(substring, start, stop)`	`rindex(substring, start, stop)`
`endswith(substring)`	`rjust(width)`	`expandtabs()`
`rpartition(substring)`	`find(substring,start, end)`	`rsplit(separator)`
`index(substring, start, stop)`	`rstrip()`	`isalnum()`
`split(separator)`	`isalpha()`	`splitlines()`
`isdigit()`	`startswith(substring)`	`islower()`
`strip()`	`isspace()`	`swapcase()`
`istitle()`	`title()`	`isupper()`
`join()`	`upper()`	`ljust(width)`
`zfill(width)`	`lower()`	

Comments

Comments in Python are created with the character `#` ; all text written after this sign will be considered as a comment:

```
# this is a comment
```

If you need to comment multiple lines, every line must start with the `#` character:

```
# This is
# a multi-line
# comment
```

Conditional statements

Conditional statements in Python are used to make a choice using a value.

if

The syntax for `if` is:

```
if expression:
    code block
```

if…else

The syntax for `if…else` is:

```
if expression:
    code block
else:
    code block
```

if…elif…else

The syntax for `if…elif…else` is:

```
if expression:
    code block
elif expression2:
    code block
else:
    code block
```

Loop statements

Loop statements execute an action until it breaks, and in Python we have two ways to implement a loop statement.

for

The syntax of the `for` loop is as follows:

```
for iterating_var in sequence:
    code block
```

while

The syntax of the `while` loop is as follows:

```
while expression:
    code block
```

Django

Let's get into the details.

Starting a project

To create a new project in Django, we must use the Django command line; the syntax is as follows:

```
django-admin startproject <name of the project>
```

Creating a new application

To create a new application, the syntax is as follows:

```
python manage.py startapp <name of the application>
```

Running the server

To run the Django server, the syntax is as follows:

```
python manage.py runserver
```

SaltStack

Let's have a look at SaltStack in detail.

Key management

Listing all key registration requests can be done as follows:

```
salt-key -L
```

Accepting a key can be done as follows:

```
salt-key -A
```

Accepting a key using the `minion_id` can be done as follows:

```
salt-key -a minion_id
```

Removing a key using the `minion_id` can be done as follows:

```
salt-key -d minion_id
```

Checking connectivity

Pinging all connected minions can be done as follows:

```
salt "*" test.ping
```

Showing all packages in the minion

The command for this is as follows:

```
salt "*" pkg.list_upgrades
```

Executing a command

The command for this is as follows:

```
salt '*' cmd.run 'ls -l /etc'
```

Installing a package

The command for this is as follows:

```
salt '*' pkg.install vim
```

Checking the network interface

The command for this is as follows:

```
salt '*' network.interfaces
```

Applying a state

In Salt, every state should be placed in the folder `/srv/salt/` on the Salt master:

```
salt '*' state.apply <name of the state>
```

Summary

What a journey! We have finally finished our journey with Python. We started with a simple introduction to the Python language, where we introduced the basic syntax of the language and presented its basic data structure. This was useful for designing our algorithms and speeding up our development.

In the next section, we presented some basic frameworks for web development, such as Flask, Django, and Amazon SDK; these are useful for new web/cloud development. Django is the most famous MVC framework for web development in Python, but we also showed you Flask, a micro-framework used for designing REST web services, which can also be used to design a microservice architecture. Memorably, we showed you how to interact with the cloud, and we also learned the basics of AWS SDK together.

We also learned how to use iPython and create notebooks, how to install a Jupyter server, and how to use simple notebooks to learn the basic principles for analyzing data.

We talked about DevOps—one of the areas with big improvements—summarizing the area of DevOps and showing you how to use certain tools to speed up your release/delivery process. We also talked about Fabric and SaltStack, and we saw how to create a fabfile and how to configure SaltStack to manage our infrastructure.

Of course, each individual area could easily be covered in separate books, and we have not exhausted our knowledge in any topic.

If you want to improve your knowledge in any area, the following are some links to help you improve and continue your journey with Python:

- SaltStack (`https://saltstack.com/`)
- iPython (`https://ipython.org/`)
- Flask (`http://flask.pocoo.org/`)
- Django (`https://www.djangoproject.com/`)
- AWS SDK (`https://aws.amazon.com/sdk-for-python/`)

What to do next?

Broaden your horizons with Packt