

Flask-CORS

build passing pypi v3.0.10 python 2.7 | 3.4 | 3.5 | 3.6 | 3.7 license mit

A Flask extension for handling Cross Origin Resource Sharing (CORS), making cross-origin AJAX possible.

This package has a simple philosophy: when you want to enable CORS, you wish to enable it for all use cases on a domain. This means no mucking around with different allowed headers, methods, etc.

By default, submission of cookies across domains is disabled due to the security implications. Please see the documentation for how to enable credential'd requests, and please make sure you add some sort of [CSRF](#) protection before doing so!

Installation

Install the extension with using pip, or easy_install.

```
$ pip install -U flask-cors
```

Usage

This package exposes a Flask extension which by default enables CORS support on all routes, for all origins and methods. It allows parameterization of all CORS headers on a per-resource level. The package also contains a decorator, for those who prefer this approach.

Simple Usage

In the simplest case, initialize the Flask-Cors extension with default arguments in order to allow CORS for all domains on all routes. See the full list of options in the [documentation](#).

```
from flask import Flask
from flask_cors import CORS

app = Flask(__name__)
CORS(app)

@app.route("/")
def helloWorld():
    return "Hello, cross-origin-world!"
```

Resource specific CORS

Alternatively, you can specify CORS options on a resource and origin level of granularity by passing a dictionary as the `resources` option, mapping paths to a set of options. See the full list of options in the [documentation](#).

```
app = Flask(__name__)
cors = CORS(app, resources={"/api/*": {"origins": "*"}})

@app.route("/api/v1/users")
def list_users():
    return "user example"
```

Route specific CORS via decorator

This extension also exposes a simple decorator to decorate flask routes with. Simply add `@cross_origin()` below a call to Flask's `@app.route(..)` to allow CORS on a given route. See the full list of options in the [decorator documentation](#).

```
@app.route("/")
@cross_origin()
def helloWorld():
    return "Hello, cross-origin-world!"
```

Documentation

For a full list of options, please see the full [documentation](#)

Troubleshooting

If things aren't working as you expect, enable logging to help understand what is going on under the hood, and why.

```
logging.getLogger('flask_cors').level = logging.DEBUG
```

Tests

A simple set of tests is included in `test/`. To run, install nose, and simply invoke `nosetests` or `python setup.py test` to exercise the tests.

Contributing

Questions, comments or improvements? Please create an issue on [Github](#), tweet at [@corydolphin](#) or send me an email. I do my best to include every contribution proposed in any way that I can.

Credits

This Flask extension is based upon the [Decorator for the HTTP Access Control](#) written by Armin Ronacher.

Installation

Python Version

We recommend using the latest version of Python. Flask supports Python 3.7 and newer.

Dependencies

These distributions will be installed automatically when installing Flask.

- [Werkzeug](#) implements WSGI, the standard Python interface between applications and servers.
- [Jinja](#) is a template language that renders the pages your application serves.
- [MarkupSafe](#) comes with Jinja. It escapes untrusted input when rendering templates to avoid injection attacks.
- [ItsDangerous](#) securely signs data to ensure its integrity. This is used to protect Flask's session cookie.
- [Click](#) is a framework for writing command line applications. It provides the `flask` command and allows adding custom management commands.

Optional dependencies

These distributions will not be installed automatically. Flask will detect and use them if you install them.

- [Blinker](#) provides support for [Signals](#).
- [python-dotenv](#) enables support for [Environment Variables From dotenv](#) when running `flask` commands.
- [Watchdog](#) provides a faster, more efficient reloader for the development server.

greenlet

You may choose to use gevent or eventlet with your application. In this case, `greenlet>=1.0` is required. When using PyPy, `PyPy>=7.3.7` is required.

These are not minimum supported versions, they only indicate the first versions that added necessary features. You should use the latest versions of each.

Virtual environments

 v: 2.2.x ▾

Use a virtual environment to manage the dependencies for your project, both in development and in production.

What problem does a virtual environment solve? The more Python projects you have, the more likely it is that you need to work with different versions of Python libraries, or even Python itself. Newer versions of libraries for one project can break compatibility in another project.

Virtual environments are independent groups of Python libraries, one for each project. Packages installed for one project will not affect other projects or the operating system's packages.

Python comes bundled with the `venv` module to create virtual environments.

Create an environment

Create a project folder and a `venv` folder within:

macOS/Linux

Windows

```
> mkdir myproject  
> cd myproject  
> py -3 -m venv venv
```

Activate the environment

Before you work on your project, activate the corresponding environment:

macOS/Linux

Windows

```
> venv\Scripts\activate
```

Your shell prompt will change to show the name of the activated environment.

Install Flask

 v: 2.2.x ▾

Within the activated environment, use the following command to install Flask:

```
$ pip install Flask
```

Flask is now installed. Check out the [Quickstart](#) or go to the [Documentation Overview](#).

 v: 2.2.x ▾

[Get unlimited access](#)[Open in app](#)Víctor Romário [Follow](#)

Apr 1, 2019 · 4 min read

...

[Save](#)

Python 3 | Configurando Variáveis de Ambiente no Windows 10!



Olá pessoal, este aqui é um guia passo a passo de como configurar as variáveis de ambiente no windows 10!

Por que configurar ?

Porque quando rodamos o nosso código de extensão .py, junto com a palavra *python*, pelo Prompt de Comando, este precisa saber o caminho do interpretador Python, para chamá-lo com o objetivo de interpretar o nosso código. Se não, vai aparecer uma mensagem informando que *python* não é um comando definido.



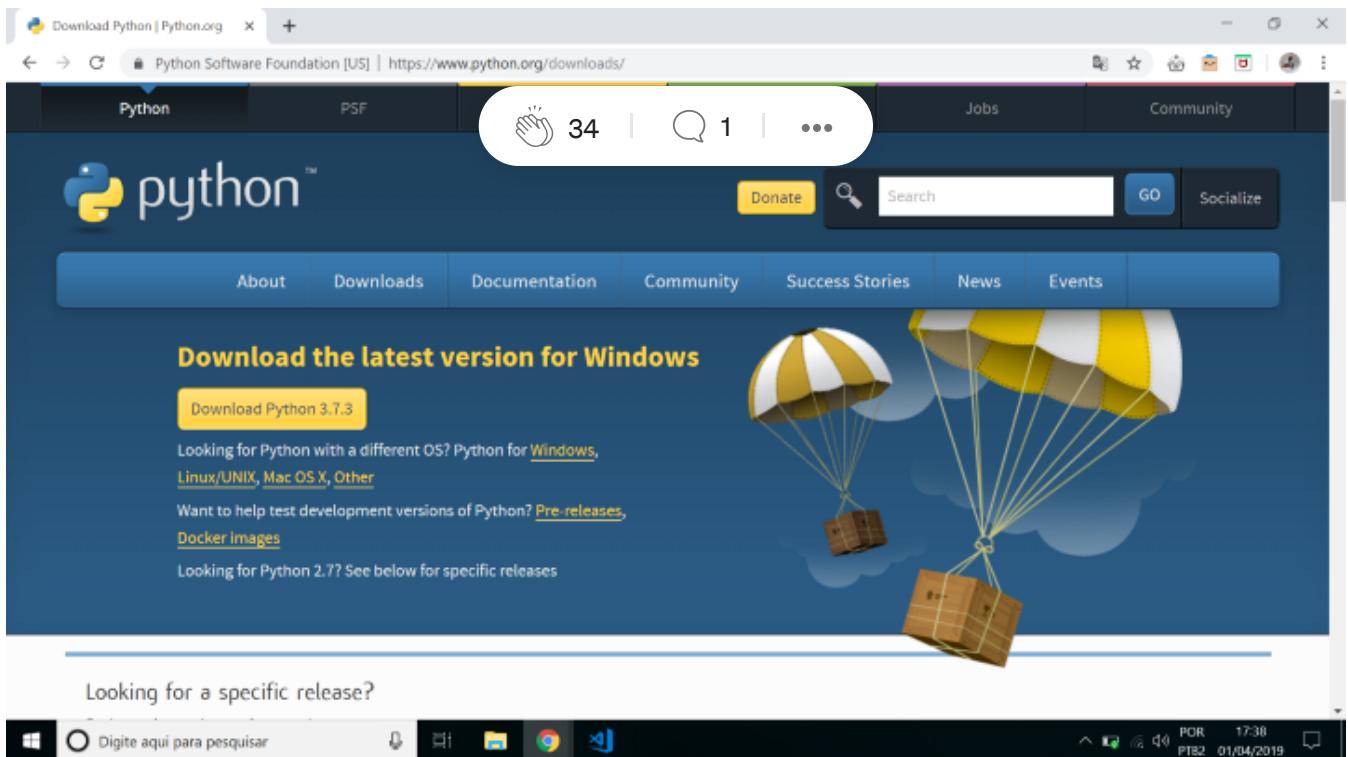
[Get unlimited access](#)[Open in app](#)

Chamando o python para interpretar o arquivo cambio.py

OBS.: Recomendável executar o Prompt de Comando como administrador.

1º passo: Instalar o Python na sua máquina

Acesse o site oficial do Python: <https://www.python.org/downloads/> e clique em download!

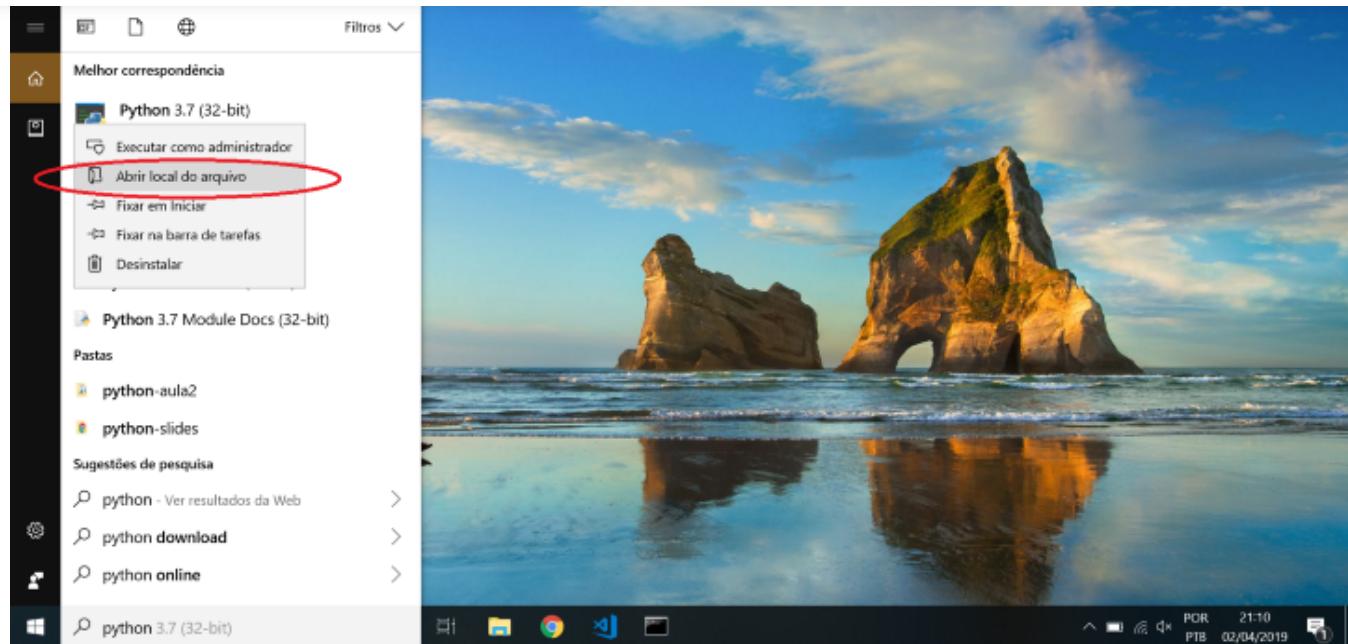


Site oficial do Python

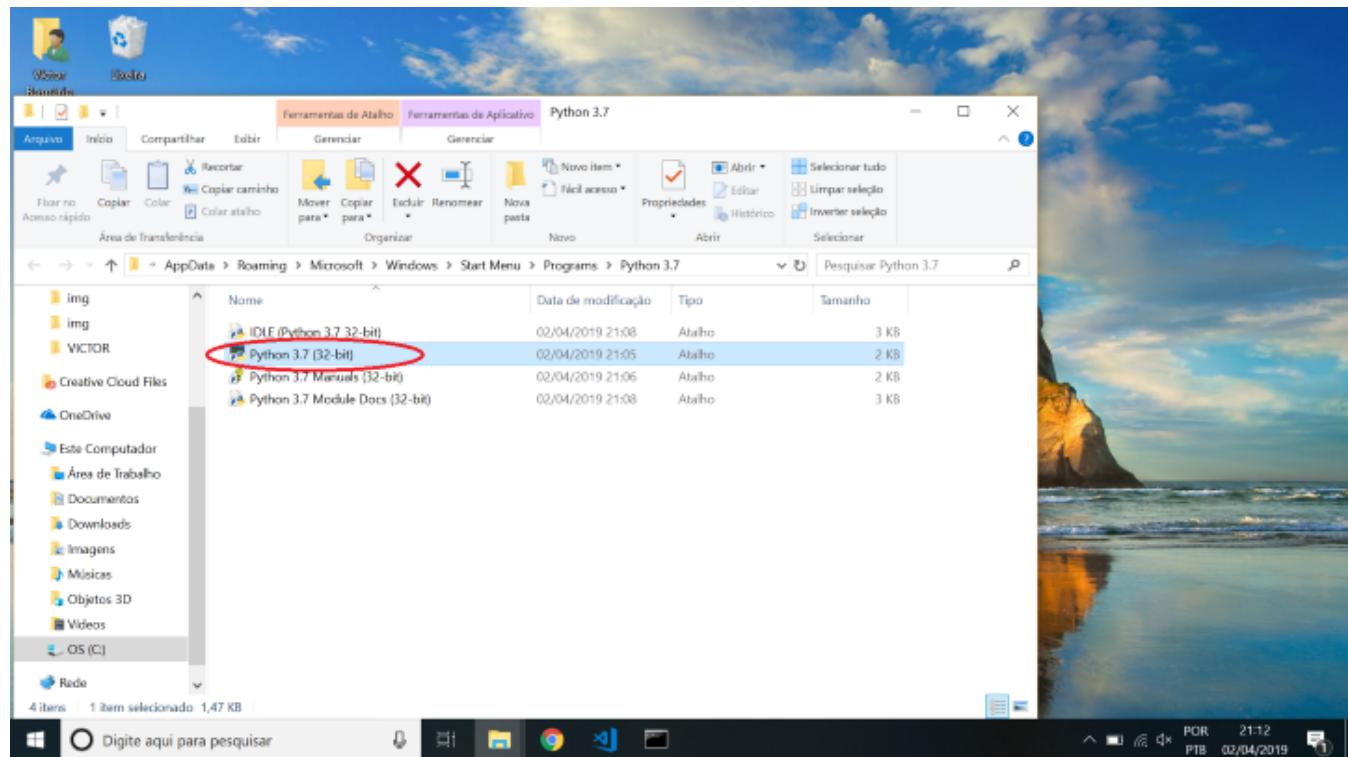
A instalação é basicamente avançar, avançar e concluir.

Depois digite python na barra de busca. E com o botão direito do mouse, clique no executável e escolha a opção de *Abrir o local do arquivo*.



[Get unlimited access](#)[Open in app](#)

Mas perceba que o local que vai abrir, os ícones vão estar em formato de atalho. Se não forem atalho você está no local certo, mas o meu abriu como formato de atalho, como mostra a imagem abaixo.



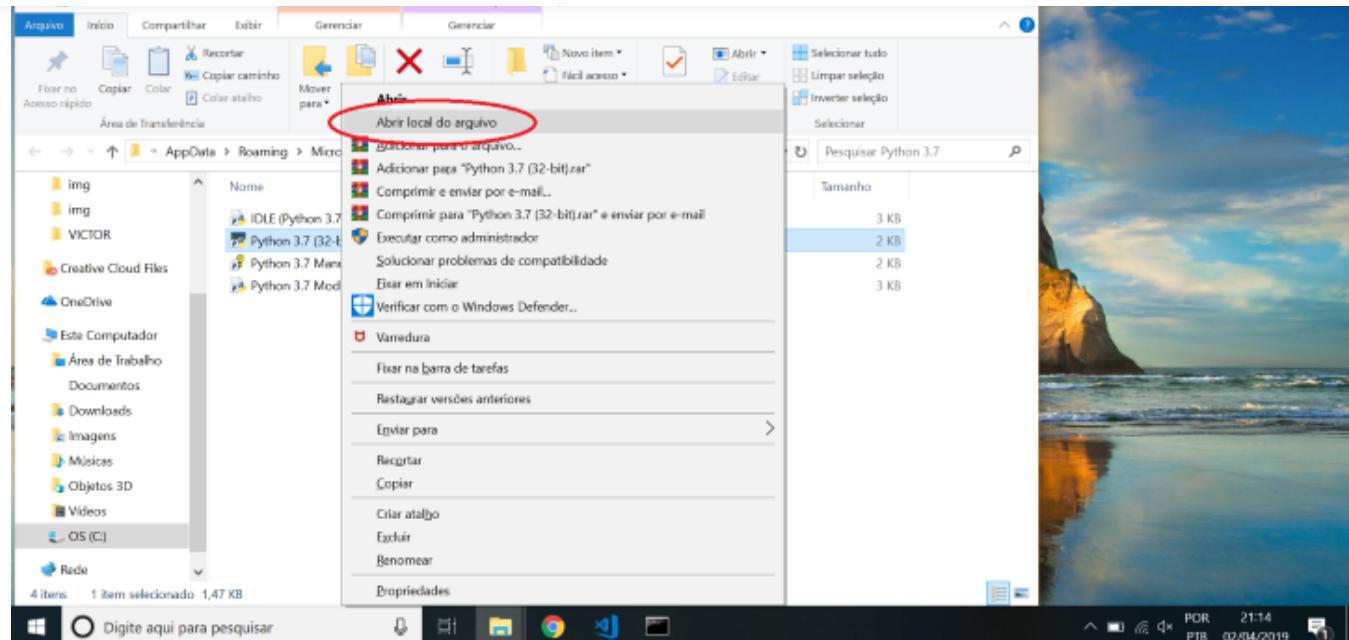
Então, com o botão direito do mouse, clique no atalho do executável do python e clique na opção de *Abrir o local do arquivo*.



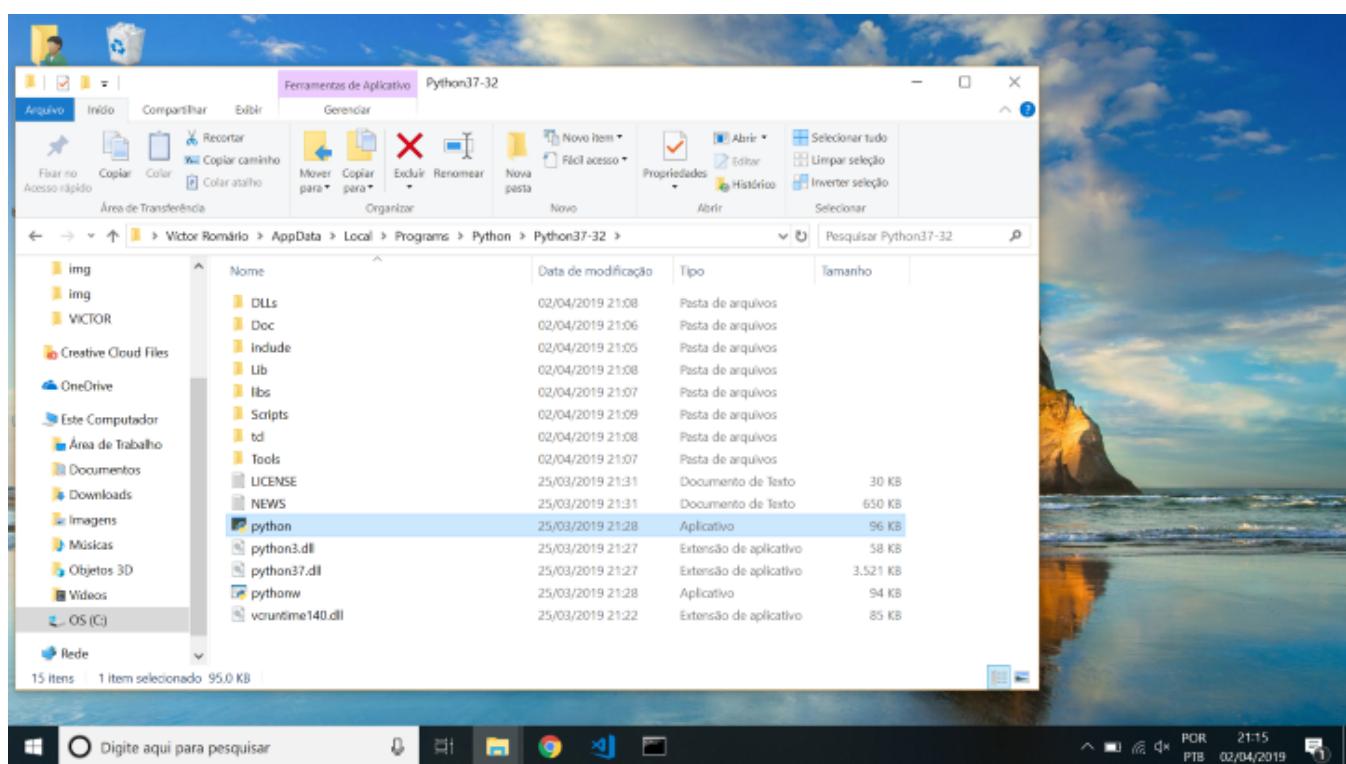


Get unlimited access

Open in app



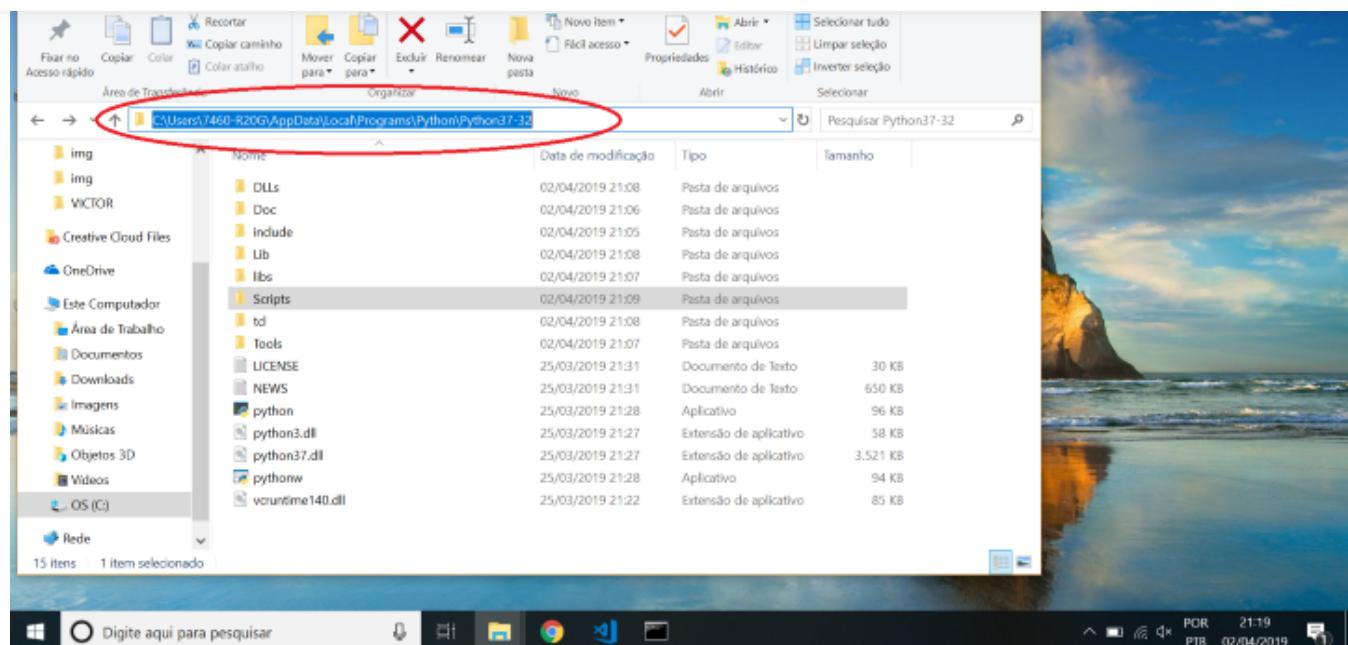
E você vai ver aparecer esta janela abaixo:



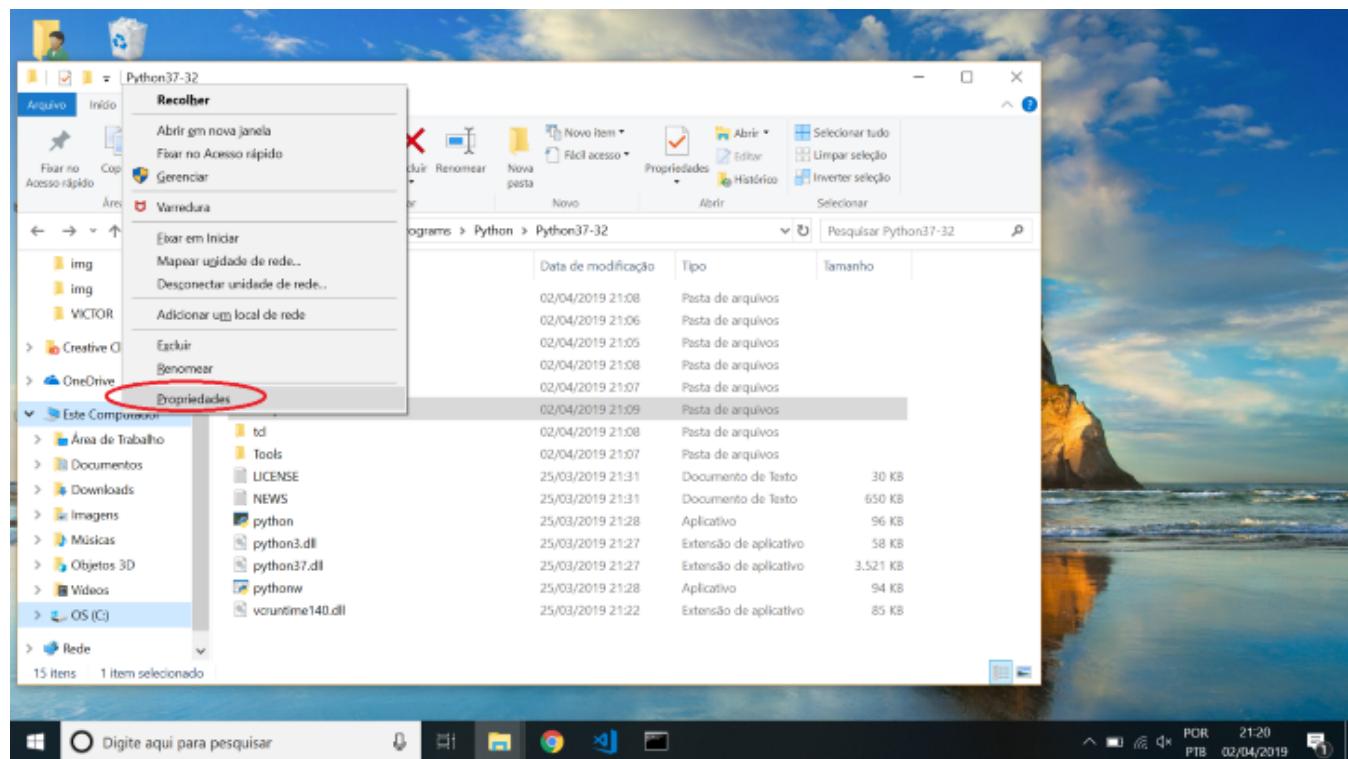
2º passo: adicionar os caminhos do interpretador e dos scripts do python nas variáveis de ambiente

Vamos agora copiar o caminho para a pasta do python



[Get unlimited access](#)[Open in app](#)

Agora, com o botão direito do mouse, clique em *Este Computador* e clique em *Propriedades*, que você verá abrir a área de sistemas, do painel de controle:



Agora clique em *Configurações avançadas do sistema*:



Get unlimited access

Open in app

Edição do Windows
Windows 10 Home Single Language
© 2018 Microsoft Corporation. Todos os direitos reservados.

Sistema

Processador: Intel(R) Core(TM) i7-7500U CPU @ 2.70GHz 2.90 GHz
Memória Instalada (RAM): 8.00 GB
Tipo de sistema: Sistema Operacional de 64 bits, processador com base em x64
Caneta e Toque: Nenhuma Entrada à Caneta ou por Toque está disponível para este vídeo

Nome do computador, domínio e configurações de grupo de trabalho

Nome do computador: DESKTOP-00VKA81
Nome completo do computador: DESKTOP-00VKA81
Descrição do computador:
Grupo de trabalho: WORKGROUP

Ativação do Windows

Windows ativo: Ler os Termos de Licença para Software Microsoft
ID do Produto (Product ID): 00342-41300-00000-ADEM

[Alterar configurações](#)

Consulte também: Segurança e Manutenção

Digitte aqui para pesquisar

POR PTB 02/04/2019 21:22

Clique em *Variáveis de Ambiente*:

Sistema

Início do Painel de Controle Exibir informações básicas sobre o computador

Propriedades do Sistema

Nome do Computador: Hardware: Avançado Proteção do Sistema: Remoto

Para trair o máximo proveito destas alterações, é preciso ter feito logon como administrador.

Desempenho

Efeitos visuais; agendamento de processador; uso de memória e memória virtual

Configurações...

Perfis de Usuário

Configurações da área de trabalho relativas à entrada

Configurações...

Inicialização e Recuperação

Informações sobre inicialização do sistema, falha do sistema e depuração

Configurações...

[Variáveis de Ambiente...](#)

OK Cancelar Aplicar

Software Microsoft
ADEM

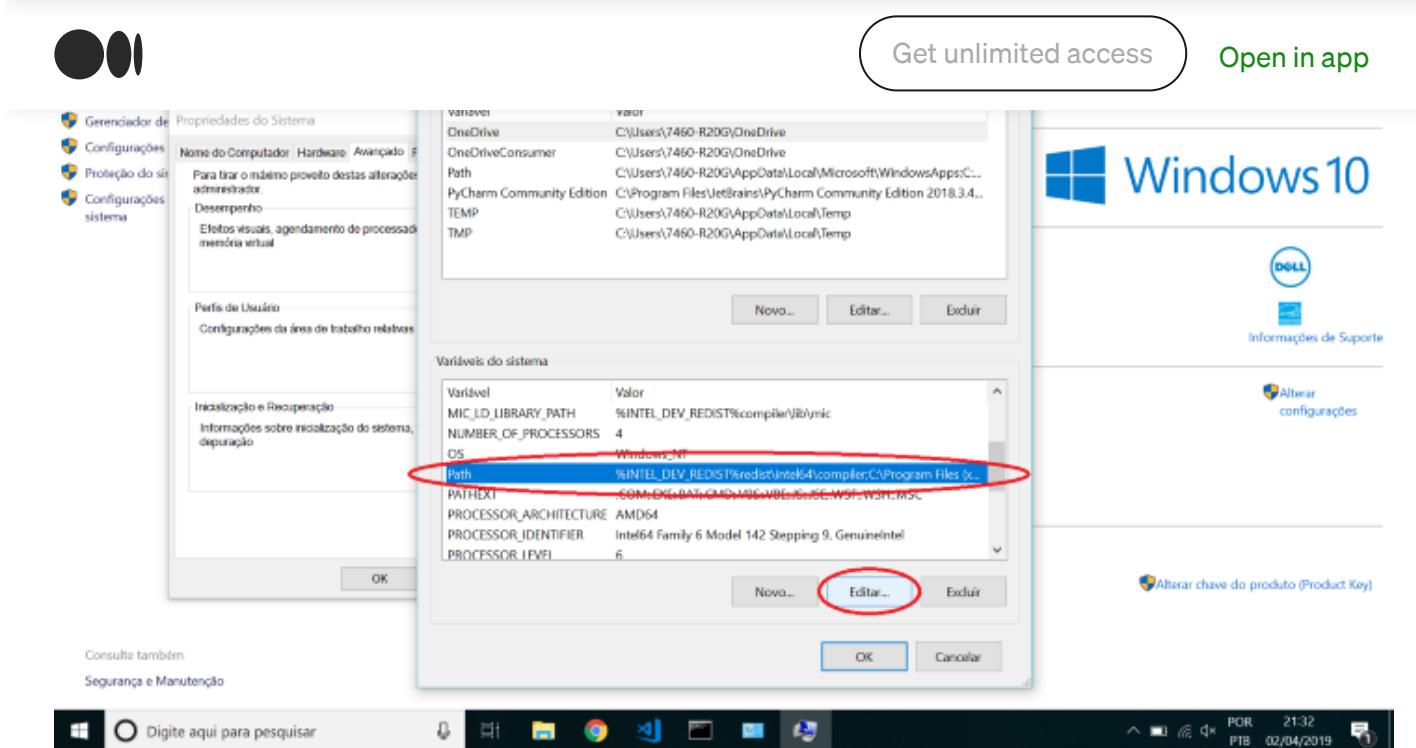
[Alterar configurações](#)

Consulte também: Segurança e Manutenção

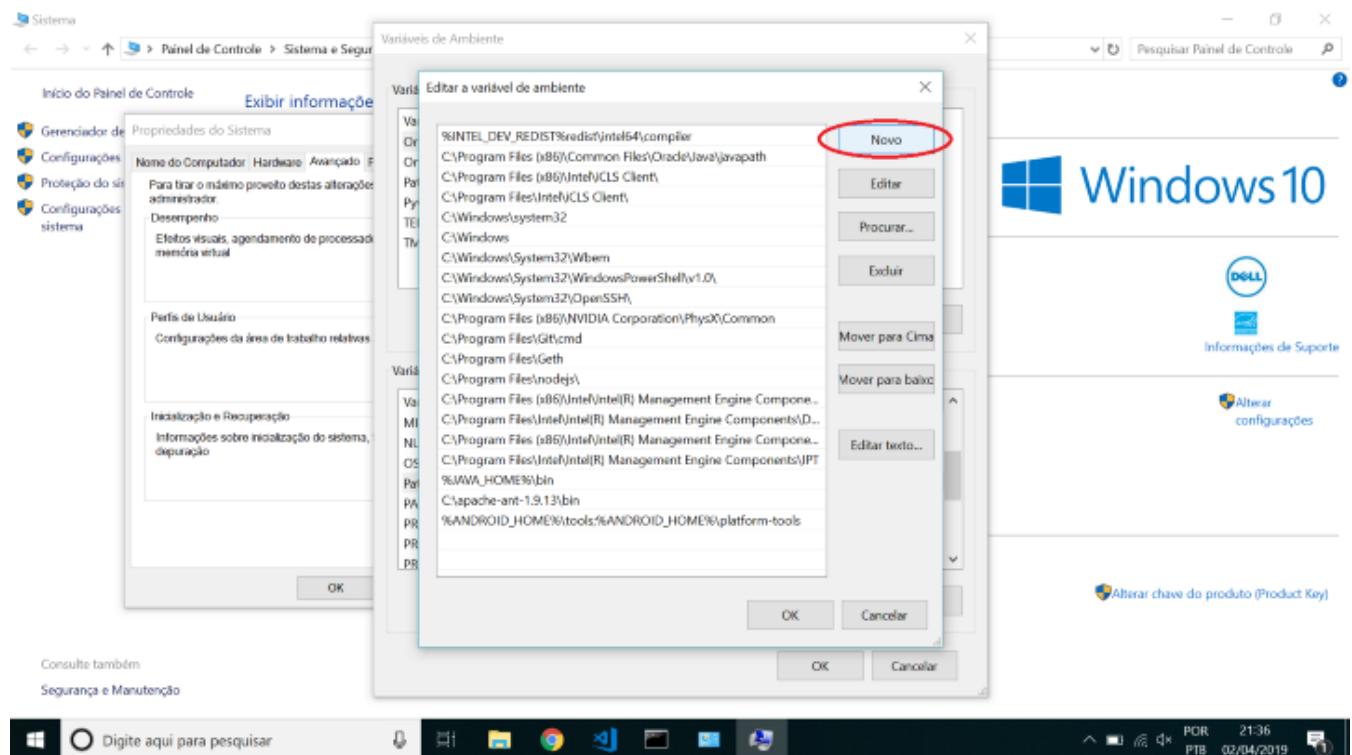
Digitte aqui para pesquisar

POR PTB 02/04/2019 21:30

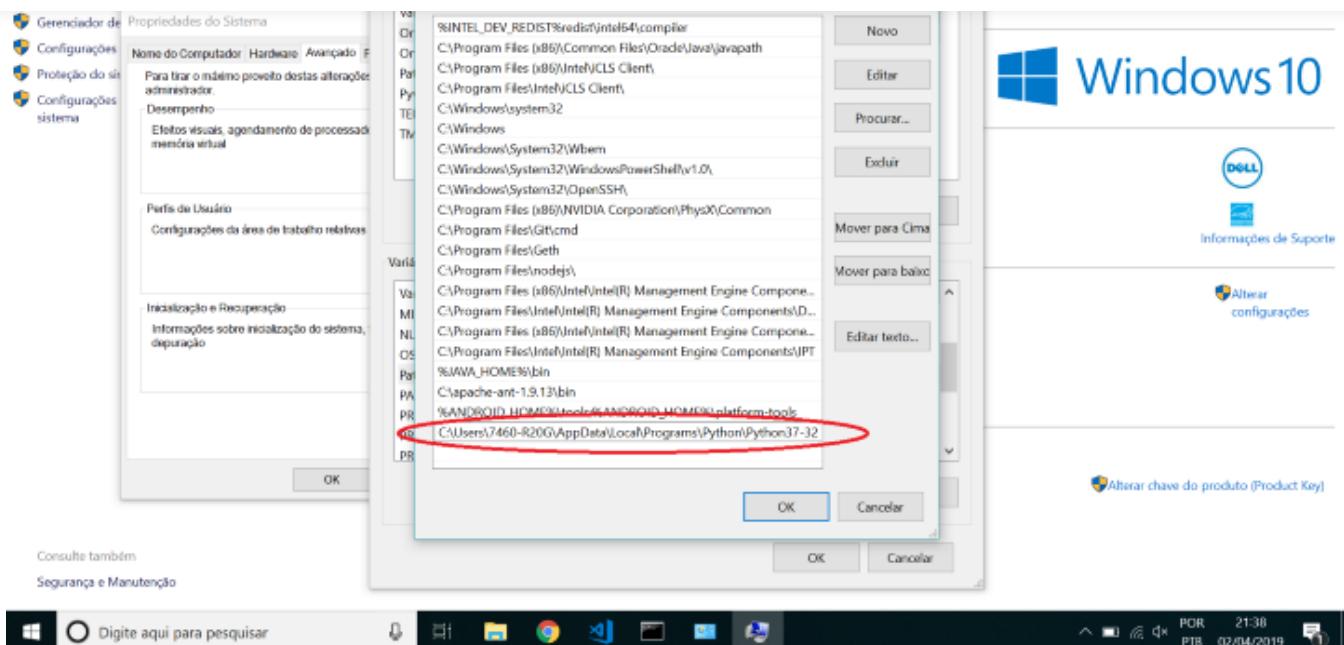
Procure por *Path* e clique em *Editar*:



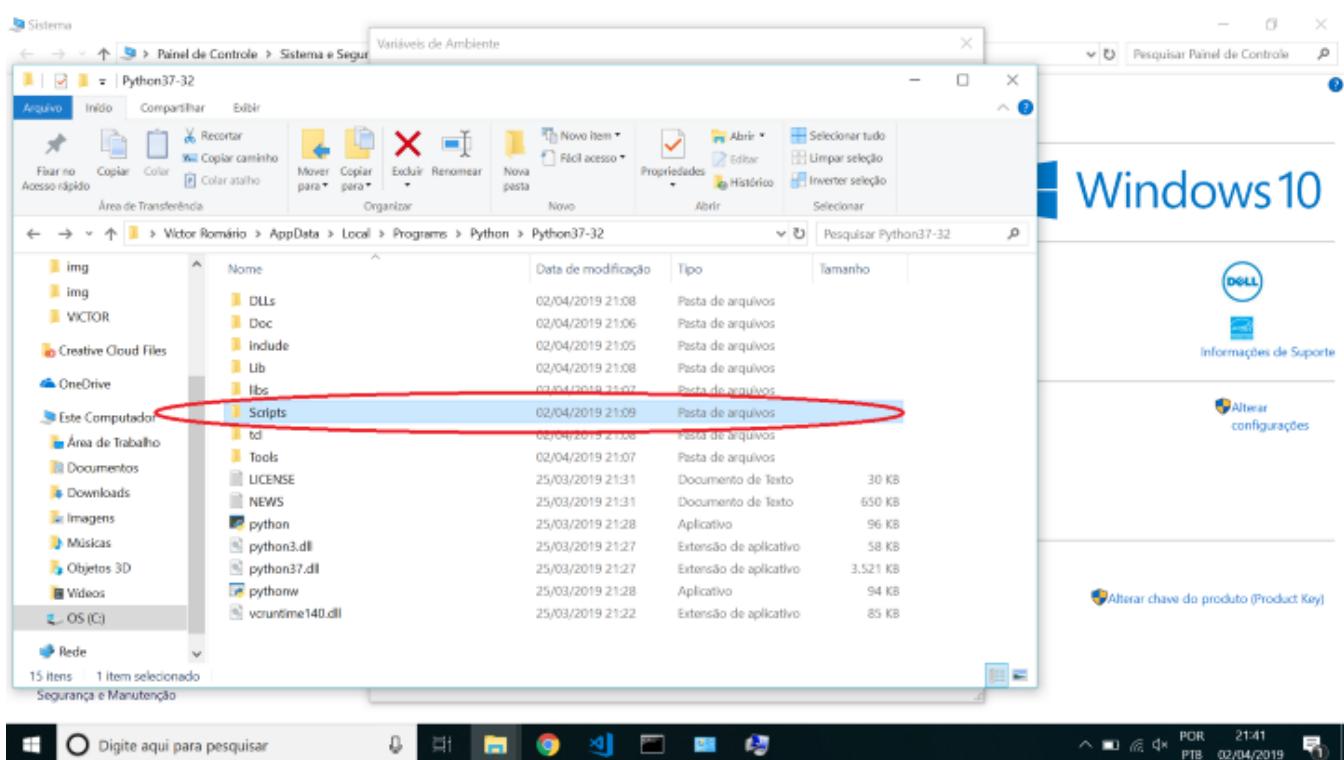
Na tela que aparecer, clique em *Novo* e você verá um campo em branco para você editar:



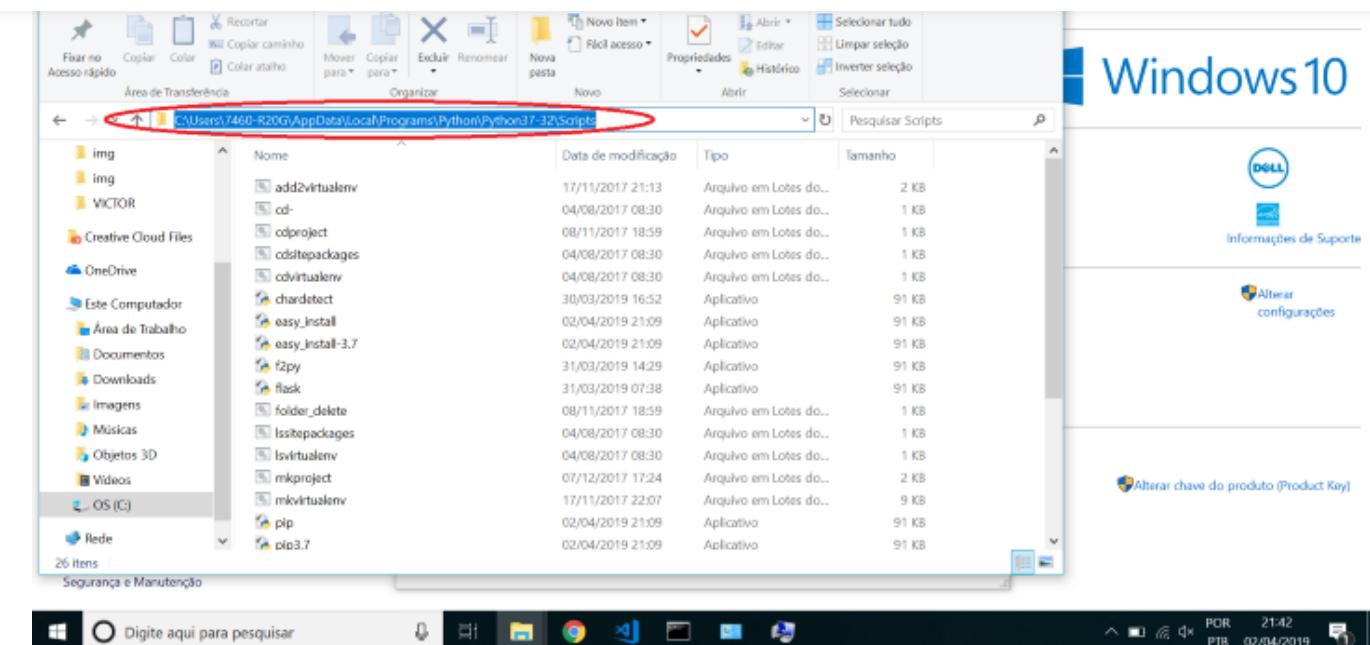
Neste novo campo, você vai colar o caminho da pasta do python:

[Get unlimited access](#)[Open in app](#)

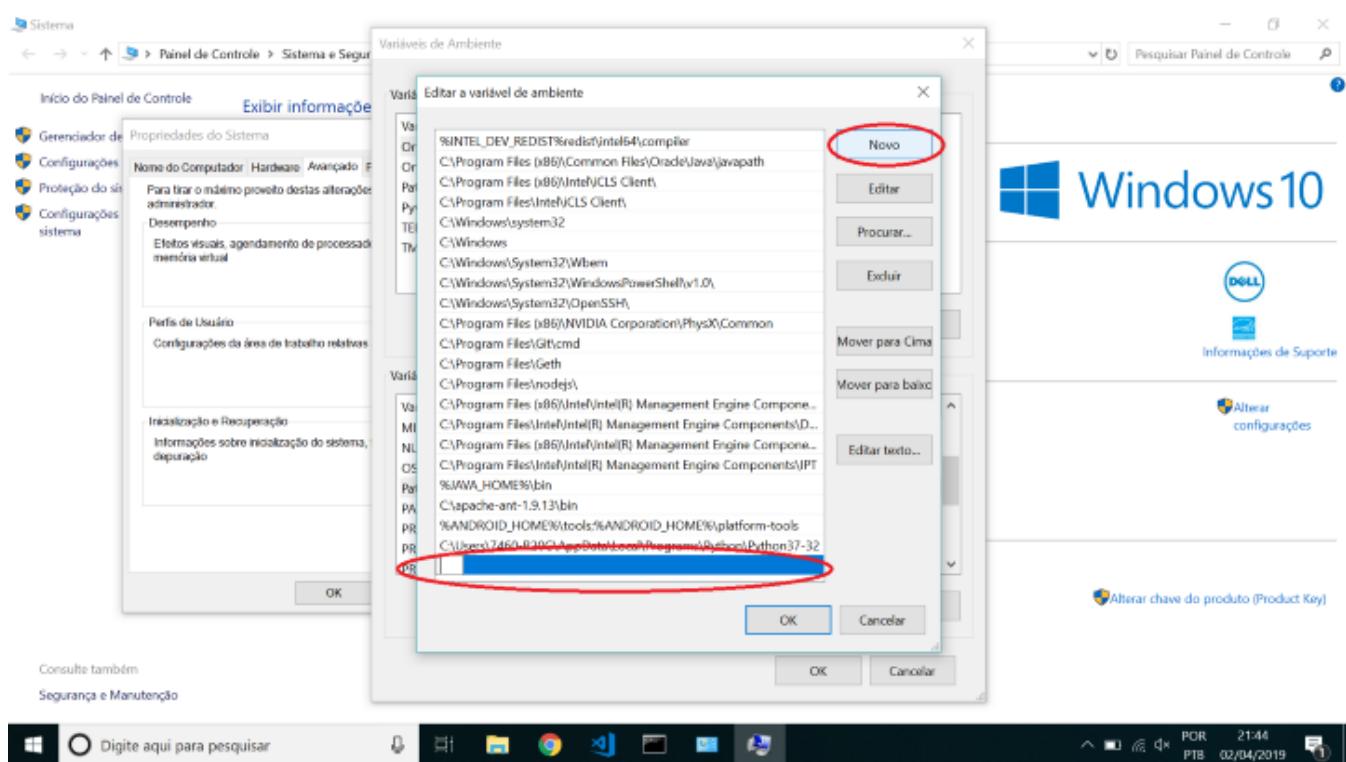
Agora vamos voltar até a pasta do python e clicar na pasta *Scripts*:



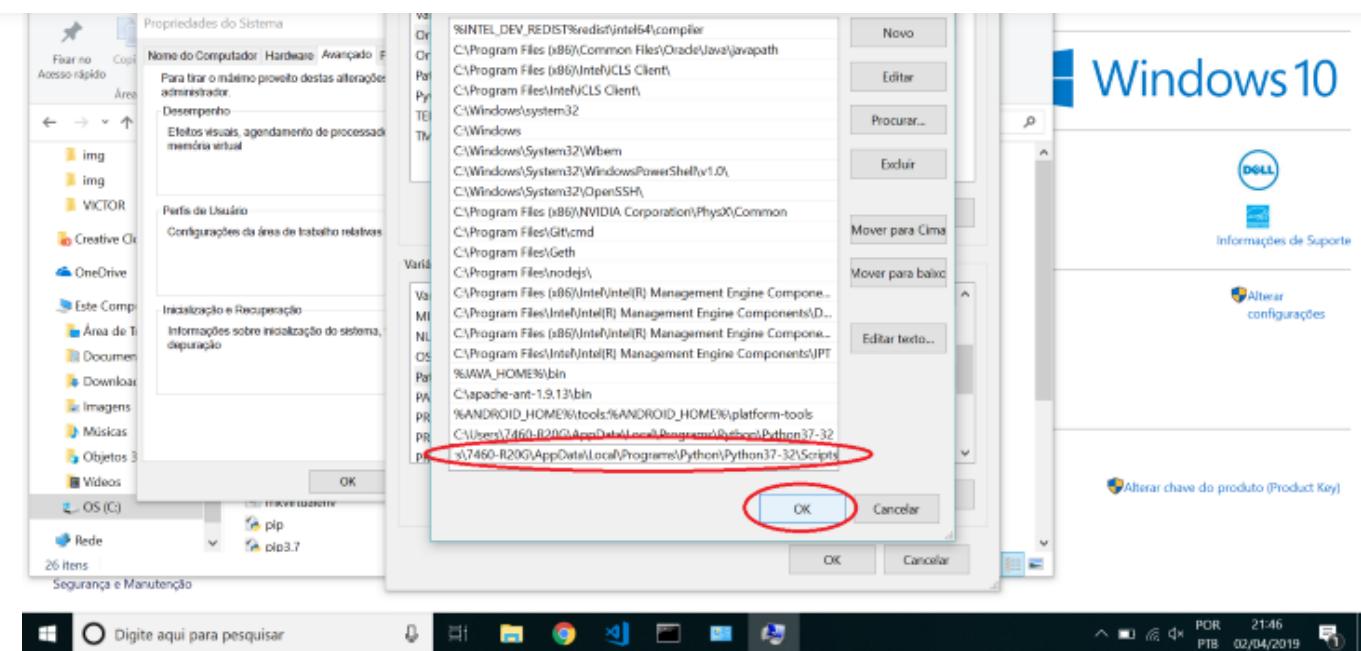
E copiar o endereço da pasta *Scripts*:

[Get unlimited access](#)[Open in app](#)

Nas variáveis de ambiente vamos clicar em *Novo* e você verá um novo campo acender:



Cole neste novo campo, o endereço da pasta *Scripts*, é nesta pasta que ficarão os módulos do *python* que iremos instalar via *Prompt de Comando*:

[Get unlimited access](#)[Open in app](#)

Depois é só clicar em ok, ok, ok e sair!



Quickstart

Eager to get started? This page gives a good introduction to Flask. Follow [Installation](#) to set up a project and install Flask first.

A Minimal Application

A minimal Flask application looks something like this:

```
from flask import Flask

app = Flask(__name__)

@app.route("/")
def hello_world():
    return "<p>Hello, World!</p>"
```

So what did that code do?

1. First we imported the `Flask` class. An instance of this class will be our WSGI application.
2. Next we create an instance of this class. The first argument is the name of the application's module or package. `__name__` is a convenient shortcut for this that is appropriate for most cases. This is needed so that Flask knows where to look for resources such as templates and static files.
3. We then use the `route()` decorator to tell Flask what URL should trigger our function.
4. The function returns the message we want to display in the user's browser. The default content type is HTML, so HTML in the string will be rendered by the browser.

Save it as `hello.py` or something similar. Make sure to not call your application `flask.py` because this would conflict with Flask itself.

To run the application, use the `flask` command or `python -m flask`. You need to tell the Flask where your application is with the `--app` option.

```
$ flask --app hello run
 * Serving Flask app 'hello'
 * Running on http://127.0.0.1:5000 (Press CTRL+C to quit)
```

Application Discovery Behavior:

As a shortcut, if the file is named `app.py` or `wsgi.py`, you don't have to use `--app`. See [Command Line Interface](#) for more details.

 v: 2.2.x ▾

This launches a very simple builtin server, which is good enough for testing but probably not what you want to use in production. For deployment options see [Deploying to Production](#).

Now head over to <http://127.0.0.1:5000/>, and you should see your hello world greeting.

If another program is already using port 5000, you'll see `OSError: [Errno 98] or OSError: [WinError 10013]` when the server tries to start. See [Address already in use](#) for how to handle that.

Externally Visible Server:

If you run the server you will notice that the server is only accessible from your own computer, not from any other in the network. This is the default because in debugging mode a user of the application can execute arbitrary Python code on your computer.

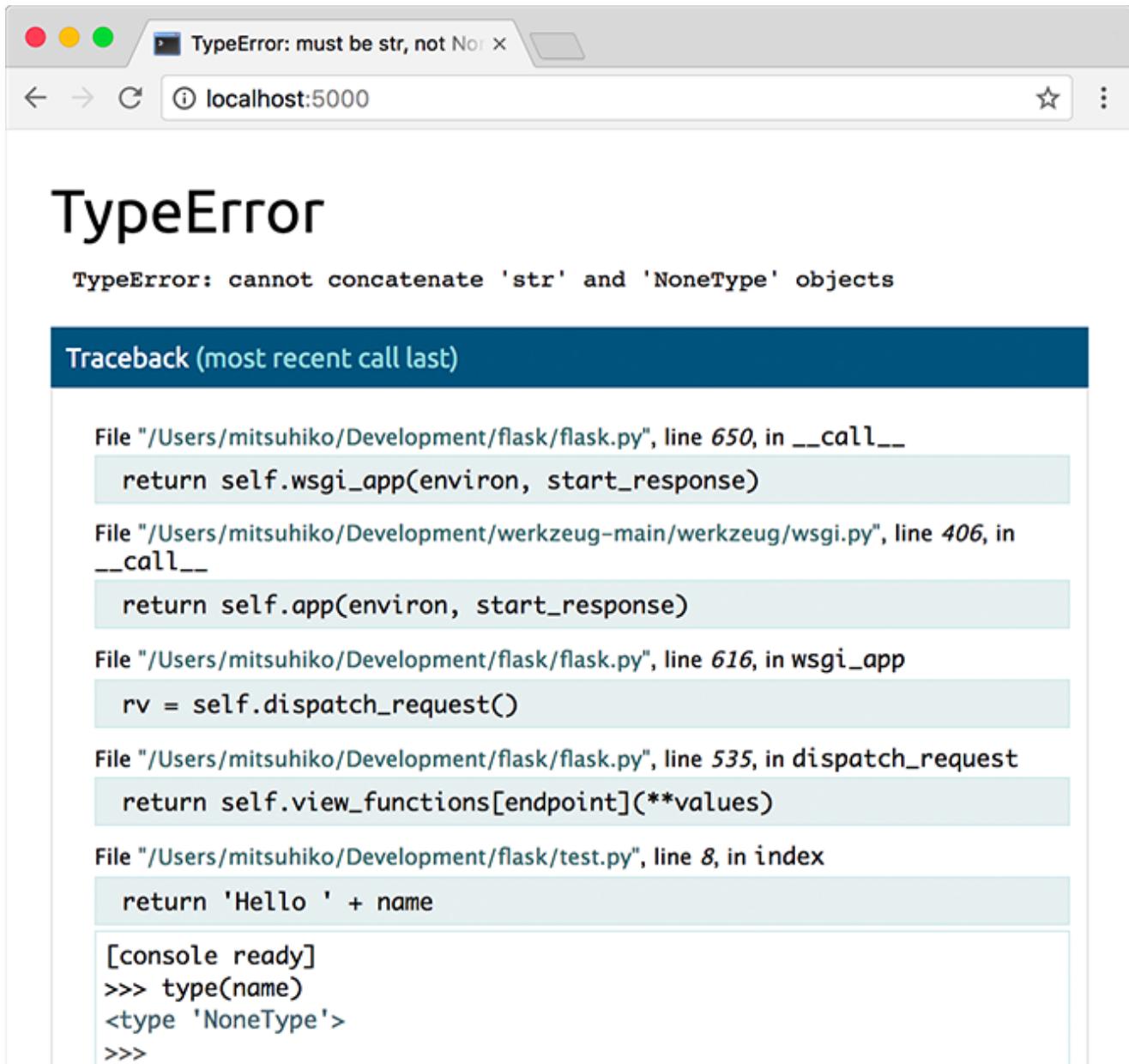
If you have the debugger disabled or trust the users on your network, you can make the server publicly available simply by adding `--host=0.0.0.0` to the command line:

```
$ flask run --host=0.0.0.0
```

This tells your operating system to listen on all public IPs.

Debug Mode ¶

The `flask run` command can do more than just start the development server. By enabling debug mode, the server will automatically reload if code changes, and will show an interactive debugger in the browser if an error occurs during a request.



The screenshot shows a browser window with the title "TypeError: must be str, not NoneType" and the URL "localhost:5000". The main content is a large error message:

```
TypeError: cannot concatenate 'str' and 'NoneType' objects
```

Below the error message is a "Traceback (most recent call last)" section:

```
File "/Users/mitsuhiko/Development/flask/flask.py", line 650, in __call__
    return self.wsgi_app(environ, start_response)

File "/Users/mitsuhiko/Development/werkzeug-main/werkzeug/wsgi.py", line 406, in
__call__
    return self.app(environ, start_response)

File "/Users/mitsuhiko/Development/flask/flask.py", line 616, in wsgi_app
    rv = self.dispatch_request()

File "/Users/mitsuhiko/Development/flask/flask.py", line 535, in dispatch_request
    return self.view_functions[endpoint](**values)

File "/Users/mitsuhiko/Development/flask/test.py", line 8, in index
    return 'Hello ' + name

[console ready]
>>> type(name)
<type 'NoneType'>
>>>
```

Warning:

The debugger allows executing arbitrary Python code from the browser. It is protected by a pin, but still represents a major security risk. Do not run the development server or debugger in a production environment.

To enable debug mode, use the `--debug` option.

```
$ flask --app hello --debug run
 * Serving Flask app 'hello'
 * Debug mode: on
 * Running on http://127.0.0.1:5000 (Press CTRL+C to quit)
 * Restarting with stat
 * Debugger is active!
 * Debugger PIN: nnn-nnn-nnn
```

 v: 2.2.x ▾

See also:

- [Development Server](#) and [Command Line Interface](#) for information about running in debug mode.
- [Debugging Application Errors](#) for information about using the built-in debugger and other debuggers.
- [Logging](#) and [Handling Application Errors](#) to log errors and display nice error pages.

HTML Escaping

When returning HTML (the default response type in Flask), any user-provided values rendered in the output must be escaped to protect from injection attacks. HTML templates rendered with Jinja, introduced later, will do this automatically.

`escape()`, shown here, can be used manually. It is omitted in most examples for brevity, but you should always be aware of how you're using untrusted data.

```
from markupsafe import escape

@app.route("/<name>")
def hello(name):
    return f"Hello, {escape(name)}!"
```

If a user managed to submit the name `<script>alert("bad")</script>`, escaping causes it to be rendered as text, rather than running the script in the user's browser.

`<name>` in the route captures a value from the URL and passes it to the view function. These variable rules are explained below.

Routing

Modern web applications use meaningful URLs to help users. Users are more likely to like a page and come back if the page uses a meaningful URL they can remember and use to directly visit a page.

Use the [route\(\)](#) decorator to bind a function to a URL.

```
@app.route('/')
def index():
    return 'Index Page'

@app.route('/hello')
def hello():
    return 'Hello, World'
```

 v: 2.2.x ▾

You can do more! You can make parts of the URL dynamic and attach multiple rules to a function.

Variable Rules

You can add variable sections to a URL by marking sections with `<variable_name>`. Your function then receives the `<variable_name>` as a keyword argument. Optionally, you can use a converter to specify the type of the argument like `<converter:variable_name>`.

```
from markupsafe import escape

@app.route('/user/<username>')
def show_user_profile(username):
    # show the user profile for that user
    return f'User {escape(username)}'

@app.route('/post/<int:post_id>')
def show_post(post_id):
    # show the post with the given id, the id is an integer
    return f'Post {post_id}'

@app.route('/path/<path:subpath>')
def show_subpath(subpath):
    # show the subpath after /path/
    return f'Subpath {escape(subpath)}'
```

Converter types:

<code>string</code>	(default) accepts any text without a slash
<code>int</code>	accepts positive integers
<code>float</code>	accepts positive floating point values
<code>path</code>	like <code>string</code> but also accepts slashes
<code>uuid</code>	accepts UUID strings

Unique URLs / Redirection Behavior

The following two rules differ in their use of a trailing slash.

```
@app.route('/projects/')
def projects():
    return 'The project page'

@app.route('/about')
def about():
    return 'The about page'
```

v: 2.2.x ▾

The canonical URL for the `projects` endpoint has a trailing slash. It's similar to a folder in a file system. If you access the URL without a trailing slash (`/projects`), Flask redirects you to the canonical URL with the trailing slash (`/projects/`).

The canonical URL for the `about` endpoint does not have a trailing slash. It's similar to the pathname of a file. Accessing the URL with a trailing slash (`/about/`) produces a 404 “Not Found” error. This helps keep URLs unique for these resources, which helps search engines avoid indexing the same page twice.

URL Building

To build a URL to a specific function, use the `url_for()` function. It accepts the name of the function as its first argument and any number of keyword arguments, each corresponding to a variable part of the URL rule. Unknown variable parts are appended to the URL as query parameters.

Why would you want to build URLs using the URL reversing function `url_for()` instead of hard-coding them into your templates?

1. Reversing is often more descriptive than hard-coding the URLs.
2. You can change your URLs in one go instead of needing to remember to manually change hard-coded URLs.
3. URL building handles escaping of special characters transparently.
4. The generated paths are always absolute, avoiding unexpected behavior of relative paths in browsers.
5. If your application is placed outside the URL root, for example, in `/myapplication` instead of `/`, `url_for()` properly handles that for you.

For example, here we use the `test_request_context()` method to try out `url_for()`. `test_request_context()` tells Flask to behave as though it's handling a request even while we use a Python shell. See [Context Locals](#).

```
from flask import url_for

@app.route('/')
def index():
    return 'index'

@app.route('/login')
def login():
    return 'login'

@app.route('/user/<username>')
def profile(username):
    return f'{username}\'s profile'

with app.test_request_context():
```

v: 2.2.x ▾

```

print(url_for('index'))
print(url_for('login'))
print(url_for('login', next='/'))
print(url_for('profile', username='John Doe'))

/
/login
/login?next=/
/user/John%20Doe

```

HTTP Methods

Web applications use different HTTP methods when accessing URLs. You should familiarize yourself with the HTTP methods as you work with Flask. By default, a route only answers to GET requests. You can use the `methods` argument of the `route()` decorator to handle different HTTP methods.

```

from flask import request

@app.route('/login', methods=['GET', 'POST'])
def login():
    if request.method == 'POST':
        return do_the_login()
    else:
        return show_the_login_form()

```

The example above keeps all methods for the route within one function, which can be useful if each part uses some common data.

You can also separate views for different methods into different functions. Flask provides a shortcut for decorating such routes with `get()`, `post()`, etc. for each common HTTP method.

```

@app.get('/login')
def login_get():
    return show_the_login_form()

@app.post('/login')
def login_post():
    return do_the_login()

```

If GET is present, Flask automatically adds support for the HEAD method and handles HEAD requests according to the [HTTP RFC](#). Likewise, OPTIONS is automatically implemented for you.

 v: 2.2.x ▾

Static Files

Dynamic web applications also need static files. That's usually where the CSS and JavaScript files are coming from. Ideally your web server is configured to serve them for you, but during development Flask can do that as well. Just create a folder called `static` in your package or next to your module and it will be available at `/static` on the application.

To generate URLs for static files, use the special '`static`' endpoint name:

```
url_for('static', filename='style.css')
```

The file has to be stored on the filesystem as `static/style.css`.

Rendering Templates

Generating HTML from within Python is not fun, and actually pretty cumbersome because you have to do the HTML escaping on your own to keep the application secure. Because of that Flask configures the [Jinja2](#) template engine for you automatically.

Templates can be used to generate any type of text file. For web applications, you'll primarily be generating HTML pages, but you can also generate markdown, plain text for emails, any anything else.

For a reference to HTML, CSS, and other web APIs, use the [MDN Web Docs](#).

To render a template you can use the `render_template()` method. All you have to do is provide the name of the template and the variables you want to pass to the template engine as keyword arguments. Here's a simple example of how to render a template:

```
from flask import render_template

@app.route('/hello/')
@app.route('/hello/<name>')
def hello(name=None):
    return render_template('hello.html', name=name)
```

Flask will look for templates in the `templates` folder. So if your application is a module, this folder is next to that module, if it's a package it's actually inside your package:

Case 1: a module:

```
/application.py
/templates
    /hello.html
```

 v: 2.2.x ▾

Case 2: a package:

```
/application
  /__init__.py
  /templates
    /hello.html
```

For templates you can use the full power of Jinja2 templates. Head over to the official [Jinja2 Template Documentation](#) for more information.

Here is an example template:

```
<!doctype html>
<title>Hello from Flask</title>
{% if name %}
  <h1>Hello {{ name }}!</h1>
{% else %}
  <h1>Hello, World!</h1>
{% endif %}
```

Inside templates you also have access to the `config`, `request`, `session` and `g`^[1] objects as well as the `url_for()` and `get_flashed_messages()` functions.

Templates are especially useful if inheritance is used. If you want to know how that works, see [Template Inheritance](#). Basically template inheritance makes it possible to keep certain elements on each page (like header, navigation and footer).

Automatic escaping is enabled, so if `name` contains HTML it will be escaped automatically. If you can trust a variable and you know that it will be safe HTML (for example because it came from a module that converts wiki markup to HTML) you can mark it as safe by using the `Markup` class or by using the `|safe` filter in the template. Head over to the Jinja 2 documentation for more examples.

Here is a basic introduction to how the `Markup` class works:

```
>>> from markupsafe import Markup
>>> Markup('<strong>Hello %s!</strong>') % '<blink>hacker</blink>'
Markup('<strong>Hello &lt;blink&gt;hacker&lt;/blink&gt;!</strong>')
>>> Markup.escape('<blink>hacker</blink>')
Markup('&lt;blink&gt;hacker&lt;/blink&gt;')
>>> Markup('<em>Marked up</em> &raquo; HTML').striptags()
'Marked up » HTML'
```

► Changelog

 v: 2.2.x ▾

[1] Unsure what that `g` object is? It's something in which you can store information for your own needs. See the documentation for `flask.g` and [Using SQLite 3 with Flask](#).

Accessing Request Data

For web applications it's crucial to react to the data a client sends to the server. In Flask this information is provided by the global `request` object. If you have some experience with Python you might be wondering how that object can be global and how Flask manages to still be threadsafe. The answer is context locals:

Context Locals

Insider Information:

If you want to understand how that works and how you can implement tests with context locals, read this section, otherwise just skip it.

Certain objects in Flask are global objects, but not of the usual kind. These objects are actually proxies to objects that are local to a specific context. What a mouthful. But that is actually quite easy to understand.

Imagine the context being the handling thread. A request comes in and the web server decides to spawn a new thread (or something else, the underlying object is capable of dealing with concurrency systems other than threads). When Flask starts its internal request handling it figures out that the current thread is the active context and binds the current application and the WSGI environments to that context (thread). It does that in an intelligent way so that one application can invoke another application without breaking.

So what does this mean to you? Basically you can completely ignore that this is the case unless you are doing something like unit testing. You will notice that code which depends on a request object will suddenly break because there is no request object. The solution is creating a request object yourself and binding it to the context. The easiest solution for unit testing is to use the `test_request_context()` context manager. In combination with the `with` statement it will bind a test request so that you can interact with it. Here is an example:

```
from flask import request

with app.test_request_context('/hello', method='POST'):
    # now you can do something with the request until the
    # end of the with block, such as basic assertions:
    assert request.path == '/hello'
    assert request.method == 'POST'
```

The other possibility is passing a whole WSGI environment to the `request_context` method:

```
with app.request_context(environ):
    assert request.method == 'POST'
```

The Request Object

The request object is documented in the API section and we will not cover it here in detail (see [Request](#)). Here is a broad overview of some of the most common operations. First of all you have to import it from the `flask` module:

```
from flask import request
```

The current request method is available by using the `method` attribute. To access form data (data transmitted in a POST or PUT request) you can use the `form` attribute. Here is a full example of the two attributes mentioned above:

```
@app.route('/login', methods=['POST', 'GET'])
def login():
    error = None
    if request.method == 'POST':
        if valid_login(request.form['username'],
                       request.form['password']):
            return log_the_user_in(request.form['username'])
        else:
            error = 'Invalid username/password'
    # the code below is executed if the request method
    # was GET or the credentials were invalid
    return render_template('login.html', error=error)
```

What happens if the key does not exist in the `form` attribute? In that case a special [KeyError](#) is raised. You can catch it like a standard [KeyError](#) but if you don't do that, a HTTP 400 Bad Request error page is shown instead. So for many situations you don't have to deal with that problem.

To access parameters submitted in the URL (`?key=value`) you can use the `args` attribute:

```
searchword = request.args.get('key', '')
```

We recommend accessing URL parameters with `get` or by catching the [KeyError](#) because users might change the URL and presenting them a 400 bad request page in that case is not user friendly.

For a full list of methods and attributes of the request object, head over to the [documentation](#).

File Uploads

You can handle uploaded files with Flask easily. Just make sure not to forget to set the `enctype="multipart/form-data"` attribute on your HTML form, otherwise the browser will not transmit your files at all.

Uploaded files are stored in memory or at a temporary location on the filesystem. You can access those files by looking at the `files` attribute on the request object. Each uploaded file is stored in that dictionary. It behaves just like a standard Python `file` object, but it also has a `save()` method that allows you to store that file on the filesystem of the server. Here is a simple example showing how that works:

```
from flask import request

@app.route('/upload', methods=['GET', 'POST'])
def upload_file():
    if request.method == 'POST':
        f = request.files['the_file']
        f.save('/var/www/uploads/uploaded_file.txt')
    ...

```

If you want to know how the file was named on the client before it was uploaded to your application, you can access the `filename` attribute. However please keep in mind that this value can be forged so never ever trust that value. If you want to use the filename of the client to store the file on the server, pass it through the `secure_filename()` function that Werkzeug provides for you:

```
from werkzeug.utils import secure_filename

@app.route('/upload', methods=['GET', 'POST'])
def upload_file():
    if request.method == 'POST':
        file = request.files['the_file']
        file.save(f"/var/www/uploads/{secure_filename(file.filename)}")
    ...

```

For some better examples, see [Uploading Files](#).

Cookies

To access cookies you can use the `cookies` attribute. To set cookies you can use the `set_cookie` method of response objects. The `cookies` attribute of request objects is a dictionary with all the cookies the client transmits. If you want to use sessions, do not use the cookies directly but instead use the [Sessions](#) in Flask that add some  v: 2.2.x top of cookies for you.

Reading cookies:

```
from flask import request

@app.route('/')
def index():
    username = request.cookies.get('username')
    # use cookies.get(key) instead of cookies[key] to not get a
    # KeyError if the cookie is missing.
```

Storing cookies:

```
from flask import make_response

@app.route('/')
def index():
    resp = make_response(render_template(...))
    resp.set_cookie('username', 'the username')
    return resp
```

Note that cookies are set on response objects. Since you normally just return strings from the view functions Flask will convert them into response objects for you. If you explicitly want to do that you can use the [make_response\(\)](#) function and then modify it.

Sometimes you might want to set a cookie at a point where the response object does not exist yet. This is possible by utilizing the [Deferred Request Callbacks](#) pattern.

For this also see [About Responses](#).

Redirects and Errors

To redirect a user to another endpoint, use the [redirect\(\)](#) function; to abort a request early with an error code, use the [abort\(\)](#) function:

```
from flask import abort, redirect, url_for

@app.route('/')
def index():
    return redirect(url_for('login'))

@app.route('/login')
def login():
    abort(401)
    this_is_never_executed()
```

 v: 2.2.x ▾

This is a rather pointless example because a user will be redirected from the index to a page they cannot access (401 means access denied) but it shows how that works.

By default a black and white error page is shown for each error code. If you want to customize the error page, you can use the [errorhandler\(\)](#) decorator:

```
from flask import render_template

@app.errorhandler(404)
def page_not_found(error):
    return render_template('page_not_found.html'), 404
```

Note the `404` after the [render_template\(\)](#) call. This tells Flask that the status code of that page should be `404` which means not found. By default `200` is assumed which translates to: all went well.

See [Handling Application Errors](#) for more details.

About Responses

The return value from a view function is automatically converted into a response object for you. If the return value is a string it's converted into a response object with the string as response body, a `200 OK` status code and a `text/html` mimetype. If the return value is a dict or list, [jsonify\(\)](#) is called to produce a response. The logic that Flask applies to converting return values into response objects is as follows:

1. If a response object of the correct type is returned it's directly returned from the view.
2. If it's a string, a response object is created with that data and the default parameters.
3. If it's an iterator or generator returning strings or bytes, it is treated as a streaming response.
4. If it's a dict or list, a response object is created using [jsonify\(\)](#).
5. If a tuple is returned the items in the tuple can provide extra information. Such tuples have to be in the form `(response, status)`, `(response, headers)`, or `(response, status, headers)`. The `status` value will override the status code and `headers` can be a list or dictionary of additional header values.
6. If none of that works, Flask will assume the return value is a valid WSGI application and convert that into a response object.

If you want to get hold of the resulting response object inside the view you can use the [make_response\(\)](#) function.

Imagine you have a view like this:

```
from flask import render_template

@app.errorhandler(404)
```

 v: 2.2.x ▾

```
def not_found(error):
    return render_template('error.html'), 404
```

You just need to wrap the return expression with `make_response()` and get the response object to modify it, then return it:

```
from flask import make_response

@app.errorhandler(404)
def not_found(error):
    resp = make_response(render_template('error.html'), 404)
    resp.headers['X-Something'] = 'A value'
    return resp
```

APIs with JSON

A common response format when writing an API is JSON. It's easy to get started writing such an API with Flask. If you return a `dict` or `list` from a view, it will be converted to a JSON response.

```
@app.route("/me")
def me_api():
    user = get_current_user()
    return {
        "username": user.username,
        "theme": user.theme,
        "image": url_for("user_image", filename=user.image),
    }

@app.route("/users")
def users_api():
    users = get_all_users()
    return [user.to_json() for user in users]
```

This is a shortcut to passing the data to the `jsonify()` function, which will serialize any supported JSON data type. That means that all the data in the dict or list must be JSON serializable.

For complex types such as database models, you'll want to use a serialization library to convert the data to valid JSON types first. There are many serialization libraries and Flask API extensions maintained by the community that support more complex applications.

Sessions

 v: 2.2.x ▾

In addition to the request object there is also a second object called `session` which allows you to store information specific to a user from one request to the next. This is implemented on top of cookies for you and signs the cookies cryptographically. What this means is that the user could look at the contents of your cookie but not modify it, unless they know the secret key used for signing.

In order to use sessions you have to set a secret key. Here is how sessions work:

```
from flask import session

# Set the secret key to some random bytes. Keep this really secret!
app.secret_key = b'_5#y2L"F4Q8z\n\xec]/'


@app.route('/')
def index():
    if 'username' in session:
        return f'Logged in as {session["username"]}'
    return 'You are not logged in'

@app.route('/login', methods=['GET', 'POST'])
def login():
    if request.method == 'POST':
        session['username'] = request.form['username']
        return redirect(url_for('index'))
    return '''
        <form method="post">
            <p><input type=text name=username></p>
            <p><input type=submit value=Login></p>
        </form>
    '''


@app.route('/logout')
def logout():
    # remove the username from the session if it's there
    session.pop('username', None)
    return redirect(url_for('index'))
```

How to generate good secret keys:

A secret key should be as random as possible. Your operating system has ways to generate pretty random data based on a cryptographic random generator. Use the following command to quickly generate a value for `Flask.secret_key` (or `SECRET_KEY`):

```
$ python -c 'import secrets; print(secrets.token_hex())'
'192b9bdd22ab9ed4d12e236c78afcb9a393ec15f71bbf5dc987d54727823bcbf'
```

A note on cookie-based sessions: Flask will take the values you put into the session object and serialize them into a cookie. If you are finding some values do not persist across requests, cookies are indeed enabled, and you are not getting a clear error message, check the size of the cookie in your page responses compared to the size supported by web browsers.

Besides the default client-side based sessions, if you want to handle sessions on the server-side instead, there are several Flask extensions that support this.

Message Flashing

Good applications and user interfaces are all about feedback. If the user does not get enough feedback they will probably end up hating the application. Flask provides a really simple way to give feedback to a user with the flashing system. The flashing system basically makes it possible to record a message at the end of a request and access it on the next (and only the next) request. This is usually combined with a layout template to expose the message.

To flash a message use the `flash()` method, to get hold of the messages you can use `get_flashed_messages()` which is also available in the templates. See [Message Flashing](#) for a full example.

Logging

► *Changelog*

Sometimes you might be in a situation where you deal with data that should be correct, but actually is not. For example you may have some client-side code that sends an HTTP request to the server but it's obviously malformed. This might be caused by a user tampering with the data, or the client code failing. Most of the time it's okay to reply with `400 Bad Request` in that situation, but sometimes that won't do and the code has to continue working.

You may still want to log that something fishy happened. This is where loggers come in handy. As of Flask 0.3 a logger is preconfigured for you to use.

Here are some example log calls:

```
app.logger.debug('A value for debugging')
app.logger.warning('A warning occurred (%d apples)', 42)
app.logger.error('An error occurred')
```

The attached `logger` is a standard logging [Logger](#), so head over to the official [docs](#) for more information.  v: 2.2.x 

See [Handling Application Errors](#).

Hooking in WSGI Middleware

To add WSGI middleware to your Flask application, wrap the application's `wsgi_app` attribute. For example, to apply Werkzeug's [ProxyFix](#) middleware for running behind Nginx:

```
from werkzeug.middleware.proxy_fix import ProxyFix
app.wsgi_app = ProxyFix(app.wsgi_app)
```

Wrapping `app.wsgi_app` instead of `app` means that `app` still points at your Flask application, not at the middleware, so you can continue to use and configure `app` directly.

Using Flask Extensions

Extensions are packages that help you accomplish common tasks. For example, Flask-SQLAlchemy provides SQLAlchemy support that makes it simple and easy to use with Flask.

For more on Flask extensions, see [Extensions](#).

Deploying to a Web Server

Ready to deploy your new Flask app? See [Deploying to Production](#).



Published in assert(QA)

Leonardo Galani [Follow](#)

Jul 27, 2020 · 7 min read

[Save](#)

Faça login em Medium com o Google



sasa claudino

edilsonclaudinosilva@gmail.com

[Continuar como sasa](#)

Tutorial de pytest para iniciantes

<https://pixnio.com/free-images/2018/11/19/2018-11-19-17-59-10.jpg>

TLDR: Se você manja um pouco de inglês ou gostaria de saber todas as possibilidades, você pode simplesmente ir para documentação oficial do pytest <https://docs.pytest.org/en/stable/index.html> e usar o tradutor do google.





algumas convenções e melhores prá

Existe uma possibilidade grande des posts sobre pytest, mas isso vai depender continuar esses posts comigo :D



Faça login em Medium com o Google



sasa claudino

edilsonclaudinosilva@gmail.com

Continuar como sasa

Mas o que é o pytest?

O *pytest* é uma framework de teste para **python** que provê soluções para executar testes e fazer validações diversas, com a possibilidade de estender com plugins e até rodar testes do próprio *unittest* do python.

É o queridinho da comunidade por sua flexibilidade, pela forma que usa *fixtures* e pela facilidade de estender suas funcionalidades.

Por mais que seja super popular, é curioso que não tenha muitos posts em português falando sobre essa belezinha, então vamos começar a fazer uma série de posts de como tirar proveito desta obra de arte.

Para instalar é tão simples quanto um

```
pip install pytest
```

Entendendo como pytest funciona

Antes de sair mudando toda sua code base para usar o brinquedo novo, vamos entender como o pytest funciona.

Quando você executa o comando `pytest` dentro do seu ambiente virtual python, ele vai fazer um scan nos diretórios e subdiretórios do seu repositório procurando por arquivos que respeitem o formato de nomenclatura `test *.py` ou `* test.py`.





com o comando `python -m pytest`), é padrão `__init__.py` em cada diretório para evitar colisão de nome nos arquivo de teste.

Também se aconselha o uso dos pads de teste estejam no mesmo repositório do código testado. A documentação oficial do `pytest` dá algumas sugestões de como organizar seus módulos de teste e módulos da aplicação:



Faça login em Medium com o Google



sasa claudino

edilsonclaudinosilva@gmail.com

Continuar como sasa

```
setup.py
src/
    mypkg/
        __init__.py
        app.py
        view.py
tests/
    __init__.py
foo/
    __init__.py
    test_view.py
bar/
    __init__.py
    test_view.py
```

retirado de
<https://docs.pytest.org/en/stable/goodpractices.html#test-discovery>

Dentro do arquivo que contém o seu teste, também é preciso respeitar a nomenclatura das classes e dos métodos de teste para que o `pytest` reconheça o que é preciso executar.

Exemplo 1:

```
def test_foo_bar():
    assert True

def not_foo_bar():
    pass
```





Faça login em Medium com o Google



s

sasa claudino

edilsonclaudinosilva@gmail.com

Continuar como sasa

```
class TestFOOBAR():
    def bar_nao_executado():
        pass

    def test_another_foo_bar():
        assert True

class SeiLa():
    def test_foo_bar_return():
        assert True
```

Neste segundo exemplo, a classe `SeiLa` tem um método que está nos padrões de nomenclatura `test_foo_bar_return` porém, não segue o padrão de nomenclatura de classe que o pytest procura. Se você quer agrupar seus testes em uma classe, use o padrão `Test*` — como é possível ver no exemplo `TestFOOBAR` em conjunto com o padrão de nomenclatura para métodos de teste.

Agrupando e Executando testes

Existem diversas formas agrupar, porém, como esse é um tutorial básico e introdutório, vamos ficar com os exemplos mais simples.

Você já sabe que se executar o comando `pytest` ele irá fazer aquela busca nos diretórios e executar os métodos e classes que atendem ao padrão do `pytest`, contudo, existem outras formas de chamar seus testes.

Vou separar em exemplos para poder explicar um pouco melhor.

Exemplo 1:

Leve em consideração a seguinte estrutura de um projeto de teste em python:





```
user/  
    __init__.py  
    test_new_user.py  
store/  
    __init__.py  
    test_new_store.py  
    test_delete_store.py  
    some_helper.py
```



Suponhamos que você precise rodar somente os testes dentro do diretório `store`. Para isso, podemos executar o pytest da seguinte forma:

```
pytest tests/store/
```

Neste caso, os 2 arquivos com nome `test` serão executados, deixando de lado todos os outros.

Seguindo a mesma ideia, para rodar somente um arquivo, você pode executar da seguinte forma:

```
pytest tests/store/test_new_store.py
```

Agora imagine que você está debugando um teste e não quer executar todos métodos de teste de um arquivo, para isso você pode executar o seguinte comando:

```
pytest tests/store/test_new_store.py::test_metodo_de_teste
```

Repare que o argumento `::test_metodo_de_teste` foi adicionado no comando, onde `test_metodo_de_teste` é o nome do método específico dentro do arquivo `test_new_store.py`.

Exemplo 2 :





Faça login em Medium com o Google



s

sasa claudino

edilsonclaudinosilva@gmail.com

Continuar como sasa

Para realizar esse tipo de agrupamento é possível usar a `pytest` chamada `marks` que é, basicamente, uma classe ou um método.

Com `pytest.marks` é possível marcar um teste para não ser executado (`skip`), definir parâmetro de execução data-driven (`parametrize`) e/ou definir um metadado "customizado".

Esse metadado customizado é o que iremos utilizar para fazer a *tag* dos métodos e *tags* de teste.

Exemplo 1:

```
import pytest

@pytest.mark.smoke
def test_send_http():
    assert True

# Exemplo baseado na documentação
https://docs.pytest.org/en/stable/example/markers.html#mark-examples
```

Neste exemplo estamos dizendo que o método de teste `test_send_http` está "marcado" como `smoke`, porém, se você tentar executar sem registrar esse `mark` customizado, ele vai reclamar que não sabe o que '`smoke`' significa.

Para isso você precisa criar um arquivo chamado `pytest.ini` na raiz do seu repositório de teste.

O conteúdo pode ser algo assim:

```
[pytest]
addopts = --strict-markers
markers =
    smoke
```





explícito neste documento, evitando

Exemplo 2:

Imagine que você tem uma classe no mesmo grupo de execução. Você não precisa colocar uma anotação em cada método, você pode, simplesmente, colocar a anotação na classe ou como um atributo da classe:



Faça login em Medium com o Google



sasa claudino

edilsonclaudinosilva@gmail.com

Continuar como sasa

```
import pytest

class TestNewUser:
    pytestmark = pytest.mark.slow

    def test_new_user_with_something(self):
        assert True

@pytest.mark.smoke
class TestNewStore:

    def test_new_store_with_something(self):
        assert True
```

Para a classe `TestNewUser` uma constante chamada `pytestmark` foi definida com `pytest.mark.slow` e com essa informação o `pytest` irá aplicar a marcação para todos os métodos da classe.

O mesmo aconteceu para classe `TestNewStore` porém, ao invés de usar uma constante específica do `pytest`, a anotação é feita a nível da classe e não dos métodos.

Rodando os testes agrupados:

Para executar seus testes com *marks* específicos, você pode rodar da seguinte forma:





Faça login em Medium com o Google



sasa claudino

edilsonclaudinosilva@gmail.com

Continuar como sasa

Pytest Fixtures

Uma das grandes barreiras para pessoas entender como executar processos a

Quem vem do `rspec` (ou similares) se

`afterMethod` mas a abordagem do pytest para fazer *setup* e *teardown* (antes e depois do teste) é diferente.

Vamos dar uma olhada em alguns exemplos.

Exemplo 1:

```
import pytest

@pytest.fixture
def smtp_connection():
    import smtplib

    return smtplib.SMTP("smtp.gmail.com", 587, timeout=5)
```

```
@pytest.fixture
def create_user():
    return {'name': 'Novo Usuário'}
```



```
def test_ehlo(smtp_connection):
    response, msg = smtp_connection.ehlo()
    assert response == 250
```

#Exemplo baseado da documentação oficial
<https://docs.pytest.org/en/stable/fixture.html>

Neste exemplo temos um teste chamado `test_ehlo` que recebe um argumento chamado `smtp_connection` e esse argumento é o nome de um método anotado como uma fixture .

Isso quer dizer que o método `smtp_connection` será executado antes do método de teste `test_ehlo` e é possível utilizar o objeto retornado pela fixture





Faça login em Medium com o Google



sasa claudino

edilsonclaudinosilva@gmail.com

Continuar como sasa

```
import pytest

@pytest.fixture
def define_target():
    return {'url': 'smtp.gmail.com', 'port': 587}

@pytest.fixture
def smtp_connection(define_target):
    import smtplib
    url = define_target['url']
    port = define_target['port']

    return smtplib.SMTP(url, port, timeout=5)

def test_ehlo(smtp_connection):
    response, msg = smtp_connection.ehlo()
    assert response == 250
```

#Exemplo baseado da documentação oficial
<https://docs.pytest.org/en/stable/fixture.html>

Outra capacidade das *fixtures* do *pytest* é possibilidade de "empilhar" *fixtures* e fazer o teste chamar somente a última.

Note que a *fixture* `smtp_connection` recebe a *fixture* `define_target` como argumento e usa ele para definir os parametros da chamada `smtplib.SMTP`.

Também leve em consideração que as *fixtures* estão no mesmo arquivo de teste e isso pode ser um problema quando existe uma grande quantidade de teste cases e suites.

Esse problema aumenta quando *fixtures* são empilhadas e o *pytest* obriga você a fazer o *import* de todas se elas estiverem em outro arquivo. Esse problema é resolvido em partes quando utilizado o plugin* `conftest` (vem junto com o *pytest* mas é considerado plugin) que será abordado mais a frente.

Exemplo 3:





```
def create_user():
    return {'user': 'foobar'}
```

```
def delete_user(user):
    ...
```

```
@pytest.fixture
```

```
def user_setup():
    user = create_user()
    yield user
    delete_user(user)
```

```
def test_new_user(user_setup):
    assert user_setup['user'] == 'foobar'
```



Faça login em Medium com o Google



sasa claudino

edilsonclaudinosilva@gmail.com

Continuar como sasa

No exemplo acima é utilizado a função `yield` que pode ser aplicada para realizar *setups e teardowns*.

Note que o objeto `user` não está no `return` da fixture, mas sim como um parametro da função `yield`. Isso quer dizer que a *fixture* será executada, o metodo de teste será chamado com o argumento `user` e, depois que o teste se finaliza, a *fixture* continua sua execução e invoca o método `delete_user`.

Exemplo 4 (conftest):

Para evitar o "*import hell*" que pode acontecer ao se ter muitas *fixtures* que são empilhadas, é uma boa prática utilizar o plugin `conftest` como mencionado anteriormente.

Não é preciso fazer nenhuma instalação adicional pois é um plugin que já vem com o `pytest` por padrão.

Para entender melhor como ele funciona, vamos levar em consideração a estrutura abaixo:

```
tests/
    __init__.py
    conftest.py
    user/
        __init__.py
        +-- no.py
```





Faça login em Medium com o Google



sasa claudino

edilsonclaudinosilva@gmail.com

Continuar como sasa

Veja que o arquivo `conftest` está na raiz do projeto. Isso significa que não há nenhuma fixture que esteja nesse arquivo. Se você quiser usar uma fixture de algum jeito, é só passar o nome da fixture como argumento do seu teste e pronto!

As *fixtures* deste arquivo serão disponibilizadas para todos os subdiretórios, e você pode ter múltiplos arquivos `conftest` em lugares estratégicos do seu repositório, respeitando a regra de disponibilização apenas para diretórios filhos e não para diretórios pais.

Bonus Trick:

Quando eu comecei a programar eu era um daqueles que colocava `print` de debug por todo lado, gerando saída aleatórios para fazer tracking do meu código e se desse algum problema, printar as coisas antes do `exception`. Depois eu conheci os debuggers e minha vida mudou. Agora imagina a minha felicidade quando descobri o `pytest` por padrão já vem com o `pdb` hookado por linha de comando caso algum teste seu falhe! Para isso basta executar o `pytest` com o argumento `--pdb` :

```
pytest -m slow --pdb
```

Críticas e sugestões são bem vindas!

Se esse post não *loopar* a gente faz uma série mais completa!!

Aguardamos seu feedback!

Thanks to Samanta Cicilia





Faça login em Medium com o Google X

sasa claudino
edilsonclaudinosilva@gmail.com

Continuar como sasa

About Help Terms Privacy

Get the Medium app



[Get unlimited access](#)[Open in app](#)

S

Welcome back. You are signed in as edilsonclaudinosilva@gmail.com. Not you?



Published in assert(QA)

The note you're looking for was deleted



Leonardo Galani [Follow](#)

...

Jul 27, 2020 · 7 min read

 [Save](#)

Tutorial de pytest para iniciantes



[Get unlimited access](#)[Open in app](#)

<https://pixnio.com/free-images/2018/11/19/2018-11-19-17-59-10.jpg>

TLDR: Se você manja um pouco de inglês ou gostaria de saber todas as possibilidades, você pode simplesmente ir para documentação oficial do pytest <https://docs.pytest.org/en/stable/index.html> e usar o tradutor do google.

Se você está começando com python e ainda não conhece essa maravilhosa ferramenta de teste, nesse tutorial inicial vou abordar o uso básico e deixar claro algumas convenções e melhores práticas para você começar do jeito certo.

Existe uma possibilidade grande desse post ser a primeira parte de uma série de posts sobre pytest, mas isso vai depender do público e se Maria Clara vai topar continuar esses posts comigo :D

Mas o que é o pytest?



[Get unlimited access](#)[Open in app](#)

E o queridinho da comunidade por sua flexibilidade, pela forma que usa *fixtures* e pela facilidade de estender suas funcionalidades.

Por mais que seja super popular, é curioso que não tenha muitos posts em português falando sobre essa beleza, então vamos começar a fazer uma série de posts de como tirar proveito desta obra de arte.

Para instalar é tão simples quanto um

```
pip install pytest
```

Entendendo como pytest funciona

Antes de sair mudando toda sua base code para usar o brinquedo novo, vamos entender como o pytest funciona.

Quando você executa o comando `pytest` dentro do seu ambiente virtual python, ele vai fazer um scan nos diretórios e subdiretórios do seu repositório procurando por arquivos que respeitem o formato de nomenclatura `test_*.py` ou `*_test.py`.

Como o processo de descoberta é amplo e existe mais de uma forma de chamar o `pytest` para rodar (tanto comando direto `pytest` mencionado anteriormente, quanto com o comando `python -m pytest`), é altamente recomendado usar o padrão `__init__.py` em cada diretório para o `pytest` reconhecer os módulos e evitar colisão de nome nos arquivo de teste.

Também se aconselha o uso dos padrões de organização de código-fonte caso seus testes estejam no mesmo repositório do código testado. A documentação oficial do `pytest` dá algumas sugestões de como organizar seus módulos de teste e módulos da aplicação:



[Get unlimited access](#)[Open in app](#)

```
tests/
    __init__.py
foo/
    __init__.py
    test_view.py
bar/
    __init__.py
    test_view.py

# retirado de
https://docs.pytest.org/en/stable/goodpractices.html#test-discovery
```

Dentro do arquivo que contém o seu teste, também é preciso respeitar a nomenclatura das classes e dos métodos de teste para que o *pytest* reconheça o que é preciso executar.

Exemplo 1:

```
def test_foo_bar():
    assert True

def not_foo_bar():
    pass
```

Levando em consideração o exemplo acima, o método `test_foo_bar` será executado, porém, o método `not_foo_bar` será deixado de lado.

Exemplo 2:

```
class TestFOOBAR():
    def bar_nao_executado():
        pass

    def test_another_foo_bar():
        assert True

class SeiLa():
```



[Get unlimited access](#)[Open in app](#)

nomenclatura `test_foo_bar_return` porém, não segue o padrão de nomenclatura de classe que o pytest procura. Se você quer agrupar seus testes em uma classe, use o padrão `Test*` — como é possível ver no exemplo `TestFOOBAR` em conjunto com o padrão de nomenclatura para métodos de teste.

Agrupando e Executando testes

Existem diversas formas agrupar, porém, como esse é um tutorial básico e introdutório, vamos ficar com os exemplos mais simples.

Você já sabe que se executar o comando `pytest` ele irá fazer aquela busca nos diretórios e executar os métodos e classes que atendem ao padrão do `pytest`, contudo, existem outras formas de chamar seus testes.

Vou separar em exemplos para poder explicar um pouco melhor.

Exemplo 1:

Leve em consideração a seguinte estrutura de um projeto de teste em python:

```
tests/
    __init__.py
    user/
        __init__.py
        test_new_user.py
    store/
        __init__.py
        test_new_store.py
        test_delete_store.py
        some_helper.py
```

Suponhamos que você precise rodar somente os testes dentro do diretório `store`. Para isso, podemos executar o `pytest` da seguinte forma:



[Get unlimited access](#)[Open in app](#)

Seguindo a mesma ideia, para rodar somente um arquivo, você pode executar da seguinte forma:

```
pytest tests/store/test_new_store.py
```

Agora imagine que você está debugando um teste e não quer executar todos métodos de teste de um arquivo, para isso você pode executar o seguinte comando:

```
pytest tests/store/test_new_store.py::test_metodo_de_teste
```

Repare que o argumento `::test_metodo_de_teste` foi adicionado no comando, onde `test_metodo_de_teste` é o nome do método específico dentro do arquivo `test_new_store.py`.

Exemplo 2 :

Levando em consideração a estrutura do exemplo acima, digamos que você quer executar testes que estão espalhados em diversos arquivos, mas você não quer executar todos os métodos de testes, pois você gostaria de criar uma suite de teste de sanidade ou "*smoke tests*".

Para realizar esse tipo de agrupamento, precisamos usar uma funcionalidade do `pytest` chamada `marks` que é, basicamente, uma anotação de metadados de uma classe ou um método.

Com `pytest.marks` é possível marcar um teste para não ser executado (`skip`), definir parâmetro de execução data-driven (`parametrize`) e/ou definir um metadado "*customizado*".

Esse metadado customizado é o que iremos utilizar para fazer a *tag* dos métodos e *tags* de teste.

Exemplo 1:



[Get unlimited access](#)[Open in app](#)

```
@pytest.mark.smoke
def test_send_http():
    assert True

# Exemplo baseado na documentação
https://docs.pytest.org/en/stable/example/markers.html#mark-examples
```

Neste exemplo estamos dizendo que o método de teste `test_send_http` está "marcado" como `smoke`, porém, se você tentar executar sem registrar esse `mark` customizado, ele vai reclamar que não sabe o que '`smoke`' significa.

Para isso você precisa criar um arquivo chamado `pytest.ini` na raiz do seu repositório de teste.

O conteúdo pode ser algo assim:

```
[pytest]
addopts = --strict-markers
markers =
    smoke
```

Neste exemplo de arquivo `pytest.ini` também estou incluindo o argumento `--strict-markers` para dizer o `pytest` não aceitar nenhum marker que não esteja explícito neste documento, evitando problemas com digitação errada.

Exemplo 2:

Imagine que você tem uma classe com uns 5~10 métodos de teste e todos são do mesmo grupo de execução. Você não precisa colocar uma anotação em cada método, você pode, simplesmente, colocar a anotação na classe ou como um atributo da classe:

```
import pytest

class TestNotification:
```



[Get unlimited access](#)[Open in app](#)

```
@pytest.mark.smoke
class TestNewStore:

    def test_new_store_with_something(self):
        assert True
```

Para a classe `TestNewUser` uma constante chamada `pytestmark` foi definida com `pytest.mark.slow` e com essa informação o `pytest` irá aplicar a marcação para todos os métodos da classe.

O mesmo aconteceu para classe `TestNewStore` porém, ao invés de usar uma constante específica do `pytest`, a anotação é feita a nível da classe e não dos métodos.

Rodando os testes agrupados:

Para executar seus testes com *marks* específicos, você pode rodar da seguinte forma:

```
pytest -m slow
```

Pytest Fixtures

Uma das grandes barreiras para pessoas que caem de paraquedas no `pytest` é entender como executar processos antes e depois do teste.

Quem vem do `rspec` (ou similares) está acostumado com hooks tipo `beforeAll` e `afterMethod` mas a abordagem do `pytest` para fazer *setup* e *teardown* (antes e depois do teste) é diferente.

Vamos dar uma olhada em alguns exemplos.



[Get unlimited access](#)[Open in app](#)

```
@pytest.fixture
def smtp_connection():
    import smtplib

    return smtplib.SMTP("smtp.gmail.com", 587, timeout=5)
```

```
@pytest.fixture
def create_user():
    return {'name': 'Novo Usuário'}
```



```
def test_ehlo(smtp_connection):
    response, msg = smtp_connection.ehlo()
    assert response == 250
```

#Exemplo baseado da documentação oficial
<https://docs.pytest.org/en/stable/fixture.html>

Neste exemplo temos um teste chamado `test_ehlo` que recebe um argumento chamado `smtp_connection` e esse argumento é o nome de um método anotado como uma fixture .

Isso quer dizer que o método `smtp_connection` será executado antes do método de teste `test_ehlo` e é possível utilizar o objeto retornado pela fixture.

Vale também resaltar que a fixture `create_user` não será executada pois ela não foi invocada.

Exemplo 2:

```
import pytest

@pytest.fixture
def define_target():
    return {'url': 'smtp.gmail.com', 'port': 587}
```



```
@pytest.fixture
def smtp_connection(define_target):
    import smtplib
    url = define_target['url']
    port = define_target['port']
```



[Get unlimited access](#)[Open in app](#)

#Exemplo baseado da documentação oficial
<https://docs.pytest.org/en/stable/fixture.html>

Outra capacidade das *fixtures* do *pytest* é possibilidade de "empilhar" *fixtures* e fazer o teste chamar somente a última.

Note que a *fixture* `smtp_connection` recebe a *fixture* `define_target` como argumento e usa ele para definir os parametros da chamada `smtplib.SMTP`.

Também leve em consideração que as *fixtures* estão no mesmo arquivo de teste e isso pode ser um problema quando existe uma grande quantidade de teste cases e suites.

Esse problema aumenta quando *fixtures* são empilhadas e o *pytest* obriga você a fazer o *import* de todas se elas estiverem em outro arquivo. Esse problema é resolvido em partes quando utilizado o plugin* `conftest` (vem junto com o *pytest* mas é considerado plugin) que será abordado mais a frente.

Exemplo 3:

```
import pytest

def create_user():
    return {'user': 'foobar'}

def delete_user(user):
    ...

@pytest.fixture
def user_setup():
    user = create_user()
    yield user
    delete_user(user)

def test_new_user(user_setup):
    assert user_setup['user'] == 'foobar'
```

No exemplo acima é utilizado a função `yield` que pode ser aplicada para realizar



[Get unlimited access](#)[Open in app](#)

chamado com o argumento `user` e, depois que o teste se finaliza, a *fixture* continua sua execução e invoca o método `delete_user`.

Exemplo 4 (conftest):

Para evitar o "*import hell*" que pode acontecer ao se ter muitas *fixtures* que são empilhadas, é uma boa prática utilizar o plugin `conftest` como mencionado anteriormente.

Não é preciso fazer nenhuma instalação adicional pois é um plugin que já vem com o `pytest` por padrão.

Para entender melhor como ele funciona, vamos levar em consideração a estrutura abaixo:

```
tests/
    __init__.py
    conftest.py
    user/
        __init__.py
        test_ne.py
    store/
        __init__.py
        test_new_store.py
        test_delete_store.py
        some_helper.py
```

Veja que o arquivo `conftest` está na raiz dos testes e você não precisa fazer *import* de nenhuma fixture que esteja nesse arquivo. É só adicionar o nome da fixture como argumento do seu teste e pronto.

As *fixtures* deste arquivo serão disponibilizadas para todos os subdiretórios, e você pode ter múltiplos arquivos `conftest` em lugares estratégicos do seu repositório, respeitando a regra de disponibilização apenas para diretórios filhos e não diretórios pais.



[Get unlimited access](#)[Open in app](#)

coisas antes do exception. Depois eu conheci os debuggers e minha vida mudou. Agora imagina a minha felicidade quando descobri o pytest por padrão já vem com `pdb` hookado por linha de comando caso algum teste seu falhe! Para isso basta executar o `pytest` com o argumento `--pdb` :

```
pytest -m slow --pdb
```

Críticas e sugestões são bem vindas!

Se esse post não *loopar* a gente faz uma série mais completa!!

Aguardamos seu feedback!

Thanks to Samanta Cicilia

169 | 1 | ...





venv— Criação de ambientes virtuais

Novo na versão 3.3.

Código-fonte: [Lib/venv/](#)

O módulo [venv](#) fornece suporte para a criação de “ambientes virtuais” leves com seus próprios diretórios de site, opcionalmente isolados dos diretórios de site do sistema. Cada ambiente virtual possui seu próprio binário Python (que corresponde à versão do binário usado para criar esse ambiente) e pode ter seu próprio conjunto independente de pacotes Python instalados nos diretórios do site.

Veja [PEP 405](#) para mais informações sobre ambientes virtuais do Python.

Ver também: [Python Packaging User Guide: Creating and using virtual environments](#)

Criando ambientes virtuais

A criação de [ambientes virtuais](#) é feita executando o comando `venv`:

```
python3 -m venv /path/to/new/virtual/environment
```

A execução desse comando cria o diretório de destino (criando qualquer diretório pai que ainda não exista) e coloca um arquivo `pyvenv.cfg` nele com uma chave `home` apontando para a instalação do Python a partir da qual o comando foi executado (um nome comum para o diretório de destino é `.venv`). Ele também cria um subdiretório `bin` (ou `Scripts` no Windows) que contém uma cópia/link simbólico de binário/binários do Python (conforme apropriado para a plataforma ou argumentos usados no momento da criação do ambiente). Ele também cria um subdiretório (inicialmente vazio) `lib/pythonX.Y/site-packages` (no Windows, é `Lib\site-packages`). Se um diretório existente for especificado, ele será reutilizado.

Obsoleto desde a versão 3.6: `pyvenv` was the recommended tool for creating virtual environments for Python 3.3 and 3.4, and is [deprecated in Python 3.6](#).

Alterado na versão 3.5: O uso de `venv` agora é recomendado para a criação de ambientes virtuais.

No Windows, invoque o comando `venv` da seguinte forma:

```
c:\>c:\Python35\python -m venv c:\path\to\myenv
```

Como alternativa, se você configurou as variáveis `PATH` e `PATHEXT` para a sua [instalação do Python](#):

```
c:\>python -m venv c:\path\to\myenv
```

O comando, se executado com `-h`, mostrará as opções disponíveis:

```
usage: venv [-h] [--system-site-packages] [--symlinks | --copies] [--clear]
            [--upgrade] [--without-pip] [--prompt PROMPT] [--upgrade-deps]
            ENV_DIR [ENV_DIR ...]
```

Creates virtual Python environments in one or more target directories.

positional arguments:
`ENV_DIR` A directory to create the environment in.



-h, --help	show this help message and exit
--system-site-packages	Give the virtual environment access to the system site-packages dir.
--symlinks	Try to use symlinks rather than copies, when symlinks are not the default for the platform.
--copies	Try to use copies rather than symlinks, even when symlinks are the default for the platform.
--clear	Delete the contents of the environment directory if it already exists, before environment creation.
--upgrade	Upgrade the environment directory to use this version of Python, assuming Python has been upgraded in-place.
--without-pip	Skips installing or upgrading pip in the virtual environment (pip is bootstrapped by default)
--prompt PROMPT	Provides an alternative prompt prefix for this environment.
--upgrade-deps	Upgrade core dependencies: pip setuptools to the latest version in PyPI

Once an environment has been created, you may wish to activate it, e.g. by sourcing an activate script in its bin directory.

Alterado na versão 3.9: Adiciona a opção --upgrade-deps para atualizar pip + setuptools para a última no PyPI

Alterado na versão 3.4: Instala o pip por padrão, adicionadas as opções --without-pip e --copies.

Alterado na versão 3.4: Nas versões anteriores, se o diretório de destino já existia, era levantado um erro, a menos que a opção --clear ou --upgrade fosse fornecida.

Nota: Embora haja suporte a links simbólicos no Windows, eles não são recomendados. É importante notar que clicar duas vezes em python.exe no Explorador de Arquivos resolverá o link simbólico com entusiasmo e ignorará o ambiente virtual.

Nota: No Microsoft Windows, pode ser necessário ativar o script Activate.ps1, definindo a política de execução para o usuário. Você pode fazer isso executando o seguinte comando do PowerShell:

PS C:> Set-ExecutionPolicy -ExecutionPolicy RemoteSigned -Scope CurrentUser

Consulte [About Execution Policies](#) para mais informações.

O arquivo pyvenv.cfg criado também inclui a chave include-system-site-packages, definida como true se venv for executado com a opção --system-site-packages; caso contrário, false.

A menos que a opção --without-pip seja dada, ensurepip será chamado para inicializar o pip no ambiente virtual.

Vários caminhos podem ser dados para venv, caso em que um ambiente virtual idêntico será criado, de acordo com as opções fornecidas, em cada caminho fornecido.

Depois que um ambiente virtual é criado, ele pode ser “ativado” usando um script no diretório binário do ambiente virtual. A chamada do script é específica da plataforma (<venv> deve ser substituído pelo caminho do diretório que contém o ambiente virtual):

Plataforma	Shell	Comando para ativar o ambiente virtual
Windows	cmd	cd %venv%\Scripts & .\activate



	fish	\$ source <venv>/bin/activate.fish
	csh/tcsh	\$ source <venv>/bin/activate.csh
	PowerShell Core	\$ <venv>/bin/Activate.ps1
Windows	cmd.exe	C:> <venv>\Scripts\activate.bat
	PowerShell	PS C:> <venv>\Scripts\Activate.ps1

Quando um ambiente virtual está ativo, a variável de ambiente `VIRTUAL_ENV` é definida como o caminho do ambiente virtual. Isso pode ser usado para verificar se um está sendo executado dentro de um ambiente virtual.

Você não *precisa* especificamente ativar um ambiente; a ativação apenas anexa o diretório binário do ambiente virtual ao seu caminho, para que “python” invoque o interpretador Python do ambiente virtual e você possa executar scripts instalados sem precisar usar o caminho completo. No entanto, todos os scripts instalados em um ambiente virtual devem ser executáveis sem ativá-lo e executados com o Python do ambiente virtual automaticamente.

Você pode desativar um ambiente virtual digitando “deactivate” no seu shell. O mecanismo exato é específico da plataforma e é um detalhe interno da implementação (normalmente, uma função de script ou shell será usada).

Novo na versão 3.4: Scripts de ativação de fish e csh.

Novo na versão 3.8: Scripts de ativação de PowerShell instalados sob POSIX para suporte a PowerShell Core.

Nota: Um ambiente virtual é um ambiente Python, de modo que o interpretador, as bibliotecas e os scripts Python instalados nele são isolados daqueles instalados em outros ambientes virtuais e (por padrão) quaisquer bibliotecas instaladas em um Python do “sistema”, ou seja, instalado como parte do seu sistema operacional.

Um ambiente virtual é uma árvore de diretórios que contém arquivos executáveis em Python e outros arquivos que indicam que é um ambiente virtual.

Ferramentas de instalação comuns, como `setuptools` e `pip`, funcionam conforme o esperado em ambientes virtuais. Em outras palavras, quando um ambiente virtual está ativo, eles instalam pacotes Python no ambiente virtual sem a necessidade de instruções explícitas.

Quando um ambiente virtual está ativo (ou seja, o interpretador Python do ambiente virtual está em execução), os atributos `sys.prefix` e `sys.exec_prefix` apontam para o diretório base do ambiente virtual, enquanto `sys.base_prefix` e `sys.base_exec_prefix` apontam para a instalação Python do ambiente não virtual que foi usada para criar o ambiente virtual. Se um ambiente virtual não estiver ativo, então `sys.prefix` é o mesmo que `sys.base_prefix` e `sys.exec_prefix` é o mesmo que `sys.base_exec_prefix` (todos eles apontam para uma instalação Python de ambiente não virtual).

Quando um ambiente virtual está ativo, todas as opções que alteram o caminho da instalação serão ignoradas em todos os arquivos de configuração `distutils` para impedir que projetos sejam inadvertidamente instalados fora do ambiente virtual.

Ao trabalhar em um shell de comando, os usuários podem ativar um ambiente virtual executando um script



de ambiente PATH do shell em execução. Em outras circunstâncias, não há necessidade de ativar um ambiente virtual; scripts instalados em ambientes virtuais têm uma linha “shebang” que aponta para o interpretador Python do ambiente virtual. Isso significa que o script será executado com esse interpretador, independentemente do valor de PATH. No Windows, o processamento da linha “shebang” é suportado se você tiver o Python Launcher for Windows instalado (foi adicionado ao Python no 3.3 - consulte [PEP 397](#) para obter mais detalhes). Portanto, clicar duas vezes em um script instalado em uma janela do Explorador do Windows deve executar o script com o interpretador correto, sem que seja necessário fazer referência ao seu ambiente virtual em PATH.

API

O método de alto nível descrito acima utiliza uma API simples que fornece mecanismos para que criadores de ambientes virtuais de terceiros personalizem a criação do ambiente de acordo com suas necessidades, a classe [EnvBuilder](#).

```
class venv.EnvBuilder(system_site_packages=False, clear=False, symlinks=False,
upgrade=False, with_pip=False, prompt=None, upgrade_deps=False)
```

A classe [EnvBuilder](#) aceita os seguintes argumentos nomeados na instanciação:

- `system_site_packages` – um valor booleano indicando que os pacotes de sites do sistema Python devem estar disponíveis para o ambiente (o padrão é `False`).
- `clear` – um valor booleano que, se verdadeiro, excluirá o conteúdo de qualquer diretório de destino existente, antes de criar o ambiente.
- `symlinks` – um valor booleano que indica se você deseja vincular o binário Python ao invés de copiar.
- `upgrade` – um valor booleano que, se verdadeiro, atualizará um ambiente existente com o Python em execução - para uso quando o Python tiver sido atualizado localmente (o padrão é `False`).
- `with_pip` – um valor booleano que, se verdadeiro, garante que o pip seja instalado no ambiente virtual. Isso usa `ensurepip` com a opção `--default-pip`.
- `prompt` – uma String a ser usada após o ambiente virtual ser ativado (o padrão é `None`, o que significa que o nome do diretório do ambiente seria usado). Se a string especial `"."` for fornecida, o nome da base do diretório atual será usado como prompt.
- `upgrade_deps` – Atualize os módulos base do venv para os mais recentes no PyPI

Alterado na versão 3.4: Adicionado o parâmetro `with_pip`

Novo na versão 3.6: Adicionado o parâmetro `prompt`

Novo na versão 3.9: Adicionado o parâmetro `upgrade_deps`

Os criadores de ferramentas de ambiente virtual de terceiros estarão livres para usar a classe fornecida [EnvBuilder](#) como uma classe base.

O env-builder retornado é um objeto que possui um método, `create`:

create(env_dir)

Cria um ambiente virtual especificando o diretório de destino (absoluto ou relativo ao diretório atual) que deve conter o ambiente virtual. O método `create` cria o ambiente no diretório especificado ou levanta uma exceção apropriada.

O método `create` da classe [EnvBuilder](#) ilustra os ganchos disponíveis para personalização de subclasse:

```
def create(self, env_dir):
```



env_dir is the target directory to create an environment in.

```
"""
env_dir = os.path.abspath(env_dir)
context = self.ensure_directories(env_dir)
self.create_configuration(context)
self.setup_python(context)
self.setup_scripts(context)
self.post_setup(context)
```

Cada um dos métodos `ensure_directories()`, `create_configuration()`, `setup_python()`, `setup_scripts()` e `post_setup()` pode ser substituído.

`ensure_directories(env_dir)`

Cria o diretório do ambiente e todos os diretórios necessários e retorna um objeto de contexto. Este é apenas um suporte para atributos (como caminhos), para uso pelos outros métodos. Os diretórios já podem existir, desde que `clear` ou `upgrade` tenham sido especificados para permitir a operação em um diretório de ambiente existente.

`create_configuration(context)`

Cria o arquivo de configuração `pyvenv.cfg` no ambiente.

`setup_python(context)`

Cria uma cópia ou link simbólico para o executável Python no ambiente. Nos sistemas POSIX, se um executável específico `python3.x` foi usado, links simbólicos para `python` e `python3` serão criados apontando para esse executável, a menos que já existam arquivos com esses nomes.

`setup_scripts(context)`

Instala scripts de ativação apropriados para a plataforma no ambiente virtual.

`upgrade_dependencies(context)`

Atualiza os principais pacotes de dependência do `venv` (atualmente `pip` e `setuptools`) no ambiente. Isso é feito através da distribuição do executável `pip` no ambiente.

Novo na versão 3.9.

`post_setup(context)`

Um método de espaço reservado que pode ser substituído em implementações de terceiros para pré-instalar pacotes no ambiente virtual ou executar outras etapas pós-criação.

Alterado na versão 3.7.2: O Windows agora usa scripts redirecionadores para `python[w].exe` em vez de copiar os binários reais. No 3.7.2, somente `setup_python()` não faz nada a menos que seja executado a partir de uma construção na árvore de origem.

Alterado na versão 3.7.3: O Windows copia os scripts redirecionadores como parte do `setup_python()` em vez de `setup_scripts()`. Este não foi o caso em 3.7.2. Ao usar links simbólicos, será feito link para os executáveis originais.

Além disso, `EnvBuilder` fornece este método utilitário que pode ser chamado de `setup_scripts()` ou `post_setup()` nas subclasses para ajudar na instalação de scripts personalizados no ambiente virtual.

`install_scripts(context, path)`

`path` é o caminho para um diretório que deve conter subdiretórios “common”, “posix” e “nt”, cada um contendo scripts destinados ao diretório bin no ambiente. O conteúdo de “common” e o diretório



correspondente a `os.name` são copiados após alguma substituição de texto dos espaços reservados:

- `__VENV_DIR__` é substituído pelo caminho absoluto do diretório do ambiente.
- `__VENV_NAME__` é substituído pelo nome do ambiente (segmento do caminho final do diretório do ambiente).
- `__VENV_PROMPT__` é substituído pelo prompt (o nome do ambiente entre parênteses e com o seguinte espaço)
- `__VENV_BIN_NAME__` é substituído pelo nome do diretório bin (bin ou Scripts).
- `__VENV_PYTHON__` é substituído pelo caminho absoluto do executável do ambiente.

É permitido que os diretórios existam (para quando um ambiente existente estiver sendo atualizado).

Há também uma função de conveniência no nível do módulo:

```
venv.create(env_dir, system_site_packages=False, clear=False, symlinks=False,
with_pip=False, prompt=None, upgrade_deps=False)
```

Cria um `EnvBuilder` com os argumentos nomeados fornecidos e chame seu método `create()` com o argumento `env_dir`.

Novo na versão 3.3.

Alterado na versão 3.4: Adicionado o parâmetro `with_pip`

Alterado na versão 3.6: Adicionado o parâmetro `prompt`

Alterado na versão 3.9: Adicionado o parâmetro `upgrade_deps`

Um exemplo de extensão de EnvBuilder

O script a seguir mostra como estender `EnvBuilder` implementando uma subclasse que instala setuptools e pip em um ambiente virtual criado:

```
import os
import os.path
from subprocess import Popen, PIPE
import sys
from threading import Thread
from urllib.parse import urlparse
from urllib.request import urlretrieve
import venv

class ExtendedEnvBuilder(venv.EnvBuilder):
    """
    This builder installs setuptools and pip so that you can pip or
    easy_install other packages into the created virtual environment.

    :param nodist: If true, setuptools and pip are not installed into the
                   created virtual environment.
    :param nopip: If true, pip is not installed into the created
                  virtual environment.
    :param progress: If setuptools or pip are installed, the progress of the
                     installation can be monitored by passing a progress
                     callable. If specified, it is called with two
                     arguments: a string indicating some progress, and a
                     context indicating where the string is coming from.
    """

    def __init__(self, *args, **kwargs):
        super().__init__(*args, **kwargs)
        self.nodist = kwargs.pop('nodist', False)
        self.nopip = kwargs.pop('nopip', False)
        self.progress = kwargs.pop('progress', None)
```



itself, and 'stdout' and 'stderr', which are obtained by reading lines from the output streams of a subprocess which is used to install the app.

If a callable is not specified, default progress information is output to sys.stderr.

"""

```
def __init__(self, *args, **kwargs):
    self.nodist = kwargs.pop('nodist', False)
    self.nopip = kwargs.pop('nopip', False)
    self.progress = kwargs.pop('progress', None)
    self.verbose = kwargs.pop('verbose', False)
    super().__init__(*args, **kwargs)
```

```
def post_setup(self, context):
    """
```

Set up any packages which need to be pre-installed into the virtual environment being created.

:param context: The information for the virtual environment creation request being processed.

"""

```
os.environ['VIRTUAL_ENV'] = context.env_dir
if not self.nodist:
    self.install_setuptools(context)
# Can't install pip without setuptools
if not self.nopip and not self.nodist:
    self.install_pip(context)
```

```
def reader(self, stream, context):
    """
```

Read Lines from a subprocess' output stream and either pass to a progress callable (if specified) or write progress information to sys.stderr.

"""

```
progress = self.progress
while True:
    s = stream.readline()
    if not s:
        break
    if progress is not None:
        progress(s, context)
    else:
        if not self.verbose:
            sys.stderr.write('.')
        else:
            sys.stderr.write(s.decode('utf-8'))
            sys.stderr.flush()
stream.close()
```

```
def install_script(self, context, name, url):
    _, _, path, _, _ = urlparse(url)
```

```
fn = os.path.split(path)[-1]
```

```
binpath = context.bin_path
```

```
distpath = os.path.join(binpath, fn)
```

```
# Download script into the virtual environment's binaries folder
```

```
urlretrieve(url, distpath)
```

```
progress = self.progress
```

```
if self.verbose:
```

```
    term = '\n'
```

```
else:
```

```
    term = ''
```



3.10.7



Ir

```
else:
    sys.stderr.write('Installing %s ...%s' % (name, term))
    sys.stderr.flush()
# Install in the virtual environment
args = [context.env_exe, fn]
p = Popen(args, stdout=PIPE, stderr=PIPE, cwd=binpath)
t1 = Thread(target=self.reader, args=(p.stdout, 'stdout'))
t1.start()
t2 = Thread(target=self.reader, args=(p.stderr, 'stderr'))
t2.start()
p.wait()
t1.join()
t2.join()
if progress is not None:
    progress('done.', 'main')
else:
    sys.stderr.write('done.\n')
# Clean up - no longer needed
os.unlink(distpath)

def install_setuptools(self, context):
    """
    Install setuptools in the virtual environment.

    :param context: The information for the virtual environment
                    creation request being processed.
    """
    url = 'https://bitbucket.org/pypa/setuptools/downloads/ez_setup.py'
    self.install_script(context, 'setuptools', url)
    # clear up the setuptools archive which gets downloaded
    pred = lambda o: o.startswith('setuptools-') and o.endswith('.tar.gz')
    files = filter(pred, os.listdir(context.bin_path))
    for f in files:
        f = os.path.join(context.bin_path, f)
        os.unlink(f)

def install_pip(self, context):
    """
    Install pip in the virtual environment.

    :param context: The information for the virtual environment
                    creation request being processed.
    """
    url = 'https://bootstrap.pypa.io/get-pip.py'
    self.install_script(context, 'pip', url)

def main(args=None):
    compatible = True
    if sys.version_info < (3, 3):
        compatible = False
    elif not hasattr(sys, 'base_prefix'):
        compatible = False
    if not compatible:
        raise ValueError('This script is only for use with '
                        'Python 3.3 or later')
    else:
        import argparse

        parser = argparse.ArgumentParser(prog=__name__,
                                        description='Creates virtual Python '
                                                    'environments in one or '
                                                    'more target ')
```



```
        help='A directory in which to create the '
              'virtual environment.')
parser.add_argument('--no-setuptools', default=False,
                    action='store_true', dest='nodist',
                    help="Don't install setuptools or pip in the "
                          "virtual environment.")
parser.add_argument('--no-pip', default=False,
                    action='store_true', dest='nopip',
                    help="Don't install pip in the virtual "
                          "environment.")
parser.add_argument('--system-site-packages', default=False,
                    action='store_true', dest='system_site',
                    help='Give the virtual environment access to the '
                          'system site-packages dir.')
if os.name == 'nt':
    use_symlinks = False
else:
    use_symlinks = True
parser.add_argument('--symlinks', default=use_symlinks,
                    action='store_true', dest='symlinks',
                    help='Try to use symlinks rather than copies,
                          when symlinks are not the default for
                          the platform.')
parser.add_argument('--clear', default=False, action='store_true',
                    dest='clear', help='Delete the contents of the
                          virtual environment
                          directory if it already
                          exists, before virtual
                          environment creation.')
parser.add_argument('--upgrade', default=False, action='store_true',
                    dest='upgrade', help='Upgrade the virtual
                          environment directory to
                          use this version of
                          Python, assuming Python
                          has been upgraded
                          in-place.')
parser.add_argument('--verbose', default=False, action='store_true',
                    dest='verbose', help='Display the output
                          from the scripts which
                          install setuptools and pip.')
options = parser.parse_args(args)
if options.upgrade and options.clear:
    raise ValueError('you cannot supply --upgrade and --clear together.')
builder = ExtendedEnvBuilder(system_site_packages=options.system_site,
                             clear=options.clear,
                             symlinks=options.symlinks,
                             upgrade=options.upgrade,
                             nodist=options.nodist,
                             nopip=options.nopip,
                             verbose=options.verbose)
for d in options.dirs:
    builder.create(d)

if __name__ == '__main__':
    rc = 1
    try:
        main()
        rc = 0
    except Exception as e:
        print('Error: %s' % e, file=sys.stderr)
    sys.exit(rc)
```



3.10.7

Ir
