tutorial.djangogirls.org

Introduction to HTML · Django Girls Tutorial

DjangoGirls

8-10 minuti

What's a template, you may ask?

A template is a file that we can re-use to present different information in a consistent format – for example, you could use a template to help you write a letter, because although each letter might contain a different message and be addressed to a different person, they will share the same format.

A Django template's format is described in a language called HTML (that's the HTML we mentioned in the first chapter, **How the Internet works**).

What is HTML?

HTML is a simple code that is interpreted by your web browser – such as Chrome, Firefox or Safari – to display a web page for the user.

HTML stands for "HyperText Markup Language". **HyperText** means it's a type of text that supports hyperlinks between pages. **Markup** means we have taken a document and marked it up with code to tell something (in this case, a browser) how to interpret the page. HTML code is built with **tags**, each one starting with < and ending with >. These tags represent markup **elements**.

Your first template!

Creating a template means creating a template file. Everything is a file, right? You have probably noticed this already.

Templates are saved in `blog/templates/blog` directory. So first create a directory called `templates` inside your blog directory. Then create another directory called `blog` inside your templates directory:

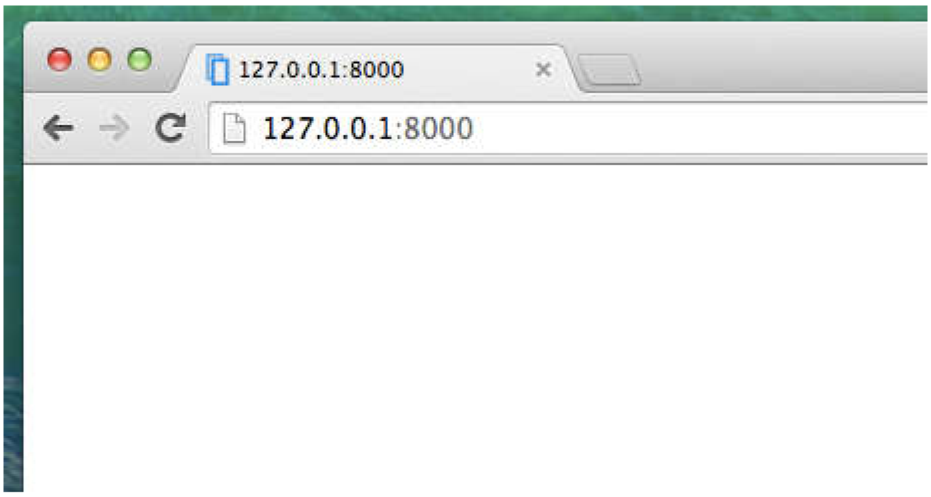
```
blog
└───templates
    └───blog
```

(You might wonder why we need two directories both called `blog` – as you will discover later, this is simply a useful naming convention that makes life easier when things start to get more complicated.)

And now create a `post_list.html` file (just leave it blank for now) inside the `blog/templates/blog` directory.

See how your website looks now: http://127.0.0.1:8000/

> If you still have an error `TemplateDoesNotExist`, try to restart your server. Go into command line, stop the server by pressing Ctrl+C (Control and C keys together) and start it again by running a `python manage.py runserver` command.



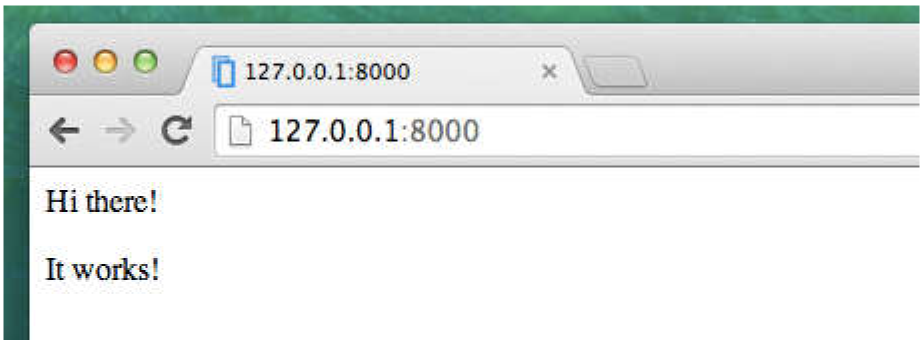
No error anymore! Congratulations :) However, your website isn't actually publishing anything except an empty page, because your template is empty too. We need to fix that.

Add the following to your template file:

blog/templates/blog/post_list.html

```
<html>
    <p>Hi there!</p>
    <p>It works!</p>
</html>
```

So how does your website look now? Visit it to find out: http://127.0.0.1:8000/



Hi there!

It works!

It worked! Nice work there :)

- The most basic tag, `<html>`, is always the beginning of any web page and `</html>` is always the end. As you can see, the whole content of the website goes between the beginning tag `<html>` and closing tag `</html>`
- `<p>` is a tag for paragraph elements; `</p>` closes each paragraph

Head and body

Each HTML page is also divided into two elements: **head** and **body**.

- **head** is an element that contains information about the document that is not displayed on the screen.
- **body** is an element that contains everything else that is displayed as part of the web page.

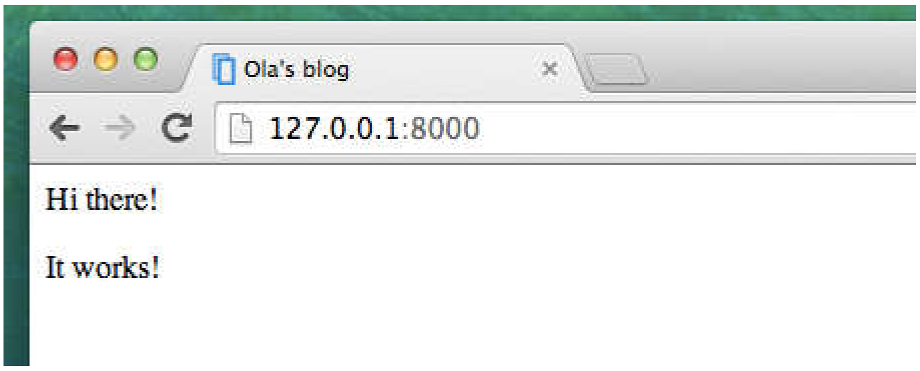
We use `<head>` to tell the browser about the configuration of the page, and `<body>` to tell it what's actually on the page.

For example, you can put a web page title element inside the `<head>`, like this:

blog/templates/blog/post_list.html

```
<html>
    <head>
        <title>Ola's blog</title>
    </head>
    <body>
        <p>Hi there!</p>
        <p>It works!</p>
    </body>
</html>
```

Save the file and refresh your page.



Hi there!

It works!

Notice how the browser has understood that "Ola's blog" is the title of your page? It has interpreted `<title>Ola's blog</title>` and placed the text in the title bar of your browser (it will also be used for bookmarks and so on).

Probably you have also noticed that each opening tag is matched by a *closing tag*, with a `/`, and that elements are *nested* (i.e. you can't close a particular tag until all the ones that were inside it have been closed too).

It's like putting things into boxes. You have one big box, `<html></html>`; inside it there is `<body></body>`, and that contains still smaller boxes: `<p></p>`.

You need to follow these rules of *closing* tags, and of *nesting* elements – if you don't, the browser may not be able to interpret them properly and your page will display incorrectly.

Customize your template

You can now have a little fun and try to customize your template! Here are a few useful tags for that:

- `<h1>A heading</h1>` for your most important heading
- `<h2>A sub-heading</h2>` for a heading at the next level
- `<h3>A sub-sub-heading</h3>` …and so on, up to `<h6>`
- `<p>A paragraph of text</p>`
- `<em>text</em>` emphasizes your text
- `<strong>text</strong>` strongly emphasizes your text
- `<br />` goes to another line (you can't put anything inside br)
- `<a href="https://djangogirls.org">link</a>` creates a link
- `<ul><li>first item</li><li>second item</li></ul>` makes a list, just like this one!
- `<div></div>` defines a section of the page

Here's an example of a full template, copy and paste it into `blog/templates/blog/post_list.html`:

blog/templates/blog/post_list.html

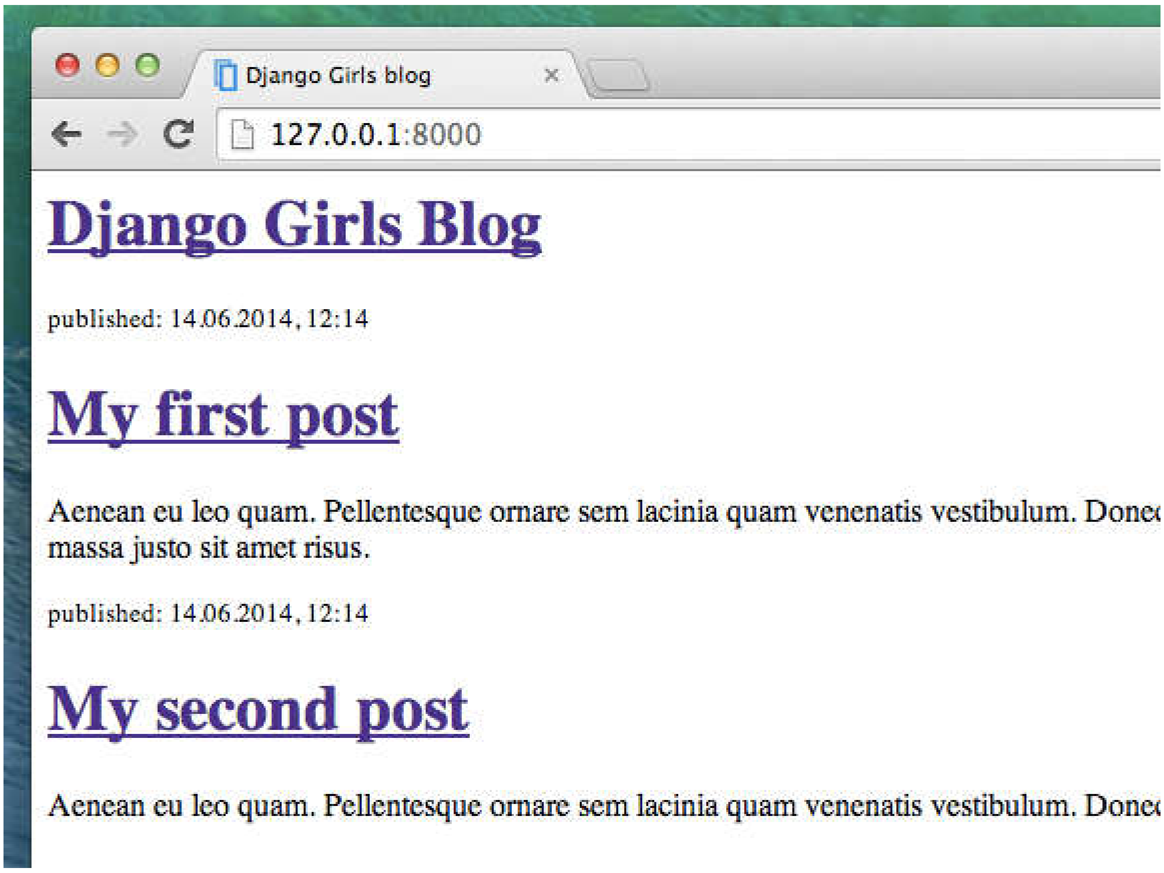
```
<html>
    <head>
        <title>Django Girls blog</title>
    </head>
    <body>
        <div>
            <h1><a href="/">Django Girls
Blog</a></h1>
        </div>

        <div>
            <p>published: 14.06.2014, 12:14</p>
            <h2><a href="">My first post</a>
</h2>
            <p>Aenean eu leo quam. Pellentesque
ornare sem lacinia quam venenatis vestibulum.
Donec id elit non mi porta gravida at eget
metus. Fusce dapibus, tellus ac cursus commodo,
tortor mauris condimentum nibh, ut fermentum
massa justo sit amet risus.</p>
        </div>

        <div>
            <p>published: 14.06.2014, 12:14</p>
            <h2><a href="">My second post</a>
</h2>
            <p>Aenean eu leo quam. Pellentesque
ornare sem lacinia quam venenatis vestibulum.
```

```
Donec id elit non mi porta gravida at eget
metus. Fusce dapibus, tellus ac cursus commodo,
tortor mauris condimentum nibh, ut f.</p>
        </div>
    </body>
</html>
```

We've created three `div` sections here.

- The first `div` element contains the title of our blog – it's a heading and a link
- Another two `div` elements contain our blogposts with a published date, `h2` with a post title that is clickable and two `p`s (paragraph) of text, one for the date and one for our blogpost.

It gives us this effect:



Django Girls Blog

published: 14.06.2014, 12:14

My first post

Aenean eu leo quam. Pellentesque ornare sem lacinia quam venenatis vestibulum. Donec
massa justo sit amet risus.

published: 14.06.2014, 12:14

My second post

Aenean eu leo quam. Pellentesque ornare sem lacinia quam venenatis vestibulum. Donec

Yaaay! But so far, our template only ever displays exactly **the same information** – whereas earlier we were talking about templates as allowing us to display **different** information in the

same format.

What we really want to do is display real posts added in our Django admin – and that's where we're going next.

One more thing: deploy!

It'd be good to see all this out and live on the Internet, right? Let's do another PythonAnywhere deploy:

Commit, and push your code up to Github

First off, let's see what files have changed since we last deployed (run these commands locally, not on PythonAnywhere):

command-line

```
$ git status
```

Make sure you're in the `djangogirls` directory and let's tell `git` to include all the changes within this directory:

command-line

```
$ git add --all .
```

> `--all` means that `git` will also recognize if you've deleted files (by default, it only recognizes new/modified files). Also remember (from chapter 3) that `.` means the current directory.

Before we upload all the files, let's check what `git` will be uploading (all the files that `git` will upload should now appear in green):

command-line

```
$ git status
```

We're almost there, now it's time to tell it to save this change in its history. We're going to give it a "commit message" where we describe what we've changed. You can type anything you'd like at this stage, but it's helpful to type something descriptive so that you can remember what you've done in the future.

command-line

```
$ git commit -m "Changed the HTML for the site."
```

Make sure you use double quotes around the commit message.

Once we've done that, we upload (push) our changes up to GitHub:

command-line

```
$ git push
```

Pull your new code down to PythonAnywhere, and reload your web app

- Open up the PythonAnywhere consoles page and go to your **Bash console** (or start a new one). Then, run:

  command-line

  ```
  $ cd ~/my-first-blog
  $ git pull
  [...]
  ```

  And watch your code get downloaded. If you want to check that it's arrived, you can hop over to the **Files tab** and view your code on PythonAnywhere.

- Finally, hop on over to the Web tab and hit **Reload** on your web app.

  Your update should be live! Go ahead and refresh your website in the browser. Changes should be visible. :)

tutorial.djangogirls.org

Introduction to Python · Django Girls Tutorial

DjangoGirls

26-33 minuti

> Part of this chapter is based on tutorials by Geek Girls Carrots (https://github.com/ggcarrots/django-carrots).

Let's write some code!

Python prompt

> For readers at home: this part is covered in the Python Basics: Integers, Strings, Lists, Variables and Errors video.

To start playing with Python, we need to open up a *command line* on your computer. You should already know how to do that – you learned it in the Intro to Command Line chapter.

Once you're ready, follow the instructions below.

We want to open up a Python console, so type in `python` on Windows or `python3` on Mac OS/Linux and hit `enter`.

command-line

```
$ python3
Python 3.6.1 (...)
Type "help", "copyright", "credits" or "license"
for more information.
>>>
```

Your first Python command!

After running the Python command, the prompt changed to `>>>`. For us this means that for now we may only use commands in the Python language. You don't have to type in `>>>` – Python will do that for you.

If you want to exit the Python console at any point, just type `exit()` or use the shortcut `Ctrl + Z` for Windows and `Ctrl + D` for Mac/Linux. Then you won't see `>>>` any longer.

For now, we don't want to exit the Python console. We want to learn more about it. Let's start with something really simple. For example, try typing some math, like `2 + 3` and hit `enter`.

command-line

```
>>> 2 + 3
5
```

Nice! See how the answer popped out? Python knows math! You could try other commands like:

- `4 * 5`
- `5 - 1`
- `40 / 2`

To perform exponential calculation, say 2 to the power 3, we type:

command-line

```
>>> 2 ** 3
8
```

Have fun with this for a little while and then get back here. :)

As you can see, Python is a great calculator. If you're wondering what else you can do…

Strings

How about your name? Type your first name in quotes like this:

command-line

```
>>> "Ola"
'Ola'
```

You've now created your first string! It's a sequence of characters that can be processed by a computer. The string must always begin and end with the same character. This may be single (`'`) or double (`"`) quotes (there is no difference!) The quotes tell Python that what's inside of them is a string.

Strings can be strung together. Try this:

command-line

```
>>> "Hi there " + "Ola"
'Hi there Ola'
```

You can also multiply strings with a number:

command-line

```
>>> "Ola" * 3
'OlaOlaOla'
```

If you need to put an apostrophe inside your string, you have two ways to do it.

Using double quotes:

command-line

```
>>> "Runnin' down the hill"
"Runnin' down the hill"
```

or escaping the apostrophe with a backslash (`\`):

command-line

```
>>> 'Runnin\' down the hill'
"Runnin' down the hill"
```

Nice, huh? To see your name in uppercase letters, simply type:

command-line

```
>>> "Ola".upper()
'OLA'
```

You just used the `upper` **method** on your string! A method (like `upper()`) is a sequence of instructions that Python has to perform on a given object (`"Ola"`) once you call it.

If you want to know the number of letters contained in your name, there is a **function** for that too!

command-line

```
>>> len("Ola")
3
```

Wonder why sometimes you call functions with a `.` at the end of a string (like `"Ola".upper()`) and sometimes you first call a function and place the string in parentheses? Well, in some cases, functions belong to objects, like `upper()`, which can only be performed on strings. In this case, we call the function a **method**. Other times, functions don't belong to anything specific and can be used on different types of objects, just like `len()`. That's why we're giving `"Ola"` as a parameter to the `len` function.

Summary

OK, enough of strings. So far you've learned about:

- **the prompt** – typing commands (code) into the Python prompt results in answers in Python

- **numbers and strings** – in Python numbers are used for math and strings for text objects
- **operators** – like `+` and `*`, combine values to produce a new one
- **functions** – like `upper()` and `len()`, perform actions on objects.

These are the basics of every programming language you learn. Ready for something harder? We bet you are!

Errors

Let's try something new. Can we get the length of a number the same way we could find out the length of our name? Type in `len(304023)` and hit `enter`:

command-line

```
>>> len(304023)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: object of type 'int' has no len()
```

We got our first error! It says that objects of type "int" (integers, whole numbers) have no length. So what can we do now? Maybe we can write our number as a string? Strings have a length, right?

command-line

```
>>> len(str(304023))
6
```

It worked! We used the `str` function inside of the `len` function. `str()` converts everything to strings.

- The `str` function converts things into **strings**
- The `int` function converts things into **integers**

> Important: we can convert numbers into text, but we can't necessarily convert text into numbers – what would `int('hello')` be anyway?

Variables

An important concept in programming is variables. A variable is nothing more than a name for something so you can use it later. Programmers use these variables to store data, make their code more readable and so they don't have to keep remembering what things are.

Let's say we want to create a new variable called `name`:

command-line

```
>>> name = "Ola"
```

You see? It's easy! It's simply: name equals Ola.

As you've noticed, your program didn't return anything like it did before. So how do we know that the variable actually exists? Simply enter `name` and hit `enter`:

command-line

```
>>> name
'Ola'
```

Yippee! Your first variable! :) You can always change what it refers to:

command-line

```
>>> name = "Sonja"
>>> name
```

```
'Sonja'
```

You can use it in functions too:

command-line

```
>>> len(name)
5
```

Awesome, right? Of course, variables can be anything – numbers too! Try this:

command-line

```
>>> a = 4
>>> b = 6
>>> a * b
24
```

But what if we used the wrong name? Can you guess what would happen? Let's try!

command-line

```
>>> city = "Tokyo"
>>> ctiy
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'ctiy' is not defined
```

An error! As you can see, Python has different types of errors and this one is called a **NameError**. Python will give you this error if you try to use a variable that hasn't been defined yet. If you encounter this error later, check your code to see if you've mistyped any names.

Play with this for a while and see what you can do!

The print function

Try this:

command-line

```
>>> name = 'Maria'
>>> name
'Maria'
>>> print(name)
Maria
```

When you just type `name`, the Python interpreter responds with the string *representation* of the variable 'name', which is the letters M-a-r-i-a, surrounded by single quotes, ''. When you say `print(name)`, Python will "print" the contents of the variable to the screen, without the quotes, which is neater.

As we'll see later, `print()` is also useful when we want to print things from inside functions, or when we want to print things on multiple lines.

Lists

Beside strings and integers, Python has all sorts of different types of objects. Now we're going to introduce one called **list**. Lists are exactly what you think they are: objects which are lists of other objects. :)

Go ahead and create a list:

command-line

```
>>> []
[]
```

Yes, this list is empty. Not very useful, right? Let's create a list of lottery numbers. We don't want to repeat ourselves all the time, so we will put it in a variable, too:

command-line

```
>>> lottery = [3, 42, 12, 19, 30, 59]
```

All right, we have a list! What can we do with it? Let's see how many lottery numbers there are in a list. Do you have any idea which function you should use for that? You know this already!

command-line

```
>>> len(lottery)
6
```

Yes! `len()` can give you a number of objects in a list. Handy, right? Maybe we will sort it now:

command-line

```
>>> lottery.sort()
```

This doesn't return anything, it just changed the order in which the numbers appear in the list. Let's print it out again and see what happened:

command-line

```
>>> print(lottery)
[3, 12, 19, 30, 42, 59]
```

As you can see, the numbers in your list are now sorted from the lowest to highest value. Congrats!

Maybe we want to reverse that order? Let's do that!

command-line

```
>>> lottery.reverse()
>>> print(lottery)
[59, 42, 30, 19, 12, 3]
```

Easy, right? If you want to add something to your list, you can do this by typing this command:

command-line

```
>>> lottery.append(199)
>>> print(lottery)
[59, 42, 30, 19, 12, 3, 199]
```

If you want to show only the first number, you can do this by using **indexes**. An index is the number that says where in a list an item occurs. Programmers prefer to start counting at 0, so the first object in your list is at index 0, the next one is at 1, and so on. Try this:

command-line

```
>>> print(lottery[0])
59
>>> print(lottery[1])
42
```

As you can see, you can access different objects in your list by using the list's name and the object's index inside of square brackets.

To delete something from your list you will need to use **indexes** as we learned above and the `pop()` method. Let's try an example and reinforce what we learned previously; we will be deleting the first number of our list.

command-line

```
>>> print(lottery)
[59, 42, 30, 19, 12, 3, 199]
>>> print(lottery[0])
59
>>> lottery.pop(0)
59
>>> print(lottery)
```

```
[42, 30, 19, 12, 3, 199]
```

That worked like a charm!

For extra fun, try some other indexes: 6, 7, 1000, -1, -6 or -1000. See if you can predict the result before trying the command. Do the results make sense?

You can find a list of all available list methods in this chapter of the Python documentation: https://docs.python.org/3/tutorial/datastructures.html

Dictionaries

> For readers at home: this part is covered in the [Python Basics: Dictionaries](#) video.

A dictionary is similar to a list, but you access values by looking up a key instead of a numeric index. A key can be any string or number. The syntax to define an empty dictionary is:

command-line

```
>>> {}
{}
```

This shows that you just created an empty dictionary. Hurray!

Now, try writing the following command (try substituting your own information, too):

command-line

```
>>> participant = {'name': 'Ola', 'country':
'Poland', 'favorite_numbers': [7, 42, 92]}
```

With this command, you just created a variable named `participant` with three key–value pairs:

- The key `name` points to the value `'Ola'` (a `string` object),

- `country` points to `'Poland'` (another `string`),
- and `favorite_numbers` points to `[7, 42, 92]` (a `list` with three numbers in it).

  You can check the content of individual keys with this syntax:

  command-line

```
>>> print(participant['name'])
Ola
```

  See, it's similar to a list. But you don't need to remember the index – just the name.

  What happens if we ask Python the value of a key that doesn't exist? Can you guess? Let's try it and see!

  command-line

```
>>> participant['age']
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
KeyError: 'age'
```

  Look, another error! This one is a **KeyError**. Python is helpful and tells you that the key `'age'` doesn't exist in this dictionary.

  When should you use a dictionary or a list? Well, that's a good point to ponder. Just have a solution in mind before looking at the answer in the next line.

- Do you just need an ordered sequence of items? Go for a list.
- Do you need to associate values with keys, so you can look them up efficiently (by key) later on? Use a dictionary.

  Dictionaries, like lists, are *mutable*, meaning that they can be changed after they are created. You can add new key–value pairs to a dictionary after it is created, like this:

command-line

```
>>> participant['favorite_language'] = 'Python'
```

Like lists, using the `len()` method on the dictionaries returns the number of key–value pairs in the dictionary. Go ahead and type in this command:

command-line

```
>>> len(participant)
4
```

I hope it makes sense up to now. :) Ready for some more fun with dictionaries? Read on for some amazing things.

You can use the `pop()` method to delete an item in the dictionary. Say, if you want to delete the entry corresponding to the key `'favorite_numbers'`, just type in the following command:

command-line

```
>>> participant.pop('favorite_numbers')
[7, 42, 92]
>>> participant
{'country': 'Poland', 'favorite_language':
'Python', 'name': 'Ola'}
```

As you can see from the output, the key–value pair corresponding to the 'favorite_numbers' key has been deleted.

As well as this, you can also change a value associated with an already-created key in the dictionary. Type this:

command-line

```
>>> participant['country'] = 'Germany'
>>> participant
{'country': 'Germany', 'favorite_language':
```

```
'Python', 'name': 'Ola'}
```

As you can see, the value of the key `'country'` has been altered from `'Poland'` to `'Germany'`. :) Exciting? Hurrah! You just learned another amazing thing.

Summary

Awesome! You know a lot about programming now. In this last part you learned about:

- **errors** – you now know how to read and understand errors that show up if Python doesn't understand a command you've given it
- **variables** – names for objects that allow you to code more easily and to make your code more readable
- **lists** – lists of objects stored in a particular order
- **dictionaries** – objects stored as key–value pairs

Excited for the next part? :)

Compare things

> For readers at home: this part is covered in the Python Basics: Comparisons video.

A big part of programming involves comparing things. What's the easiest thing to compare? Numbers, of course. Let's see how that works:

command-line

```
>>> 5 > 2
True
>>> 3 < 1
False
>>> 5 > 2 * 2
```

```
True
>>> 1 == 1
True
>>> 5 != 2
True
```

We gave Python some numbers to compare. As you can see, not only can Python compare numbers, but it can also compare method results. Nice, huh?

Do you wonder why we put two equal signs `==` next to each other to compare if numbers are equal? We use a single `=` for assigning values to variables. You always, **always** need to put two of them – `==` – if you want to check if things are equal to each other. We can also state that things are unequal to each other. For that, we use the symbol `!=`, as shown in the example above.

Give Python two more tasks:

command-line

```
>>> 6 >= 12 / 2
True
>>> 3 <= 2
False
```

`>` and `<` are easy, but what do `>=` and `<=` mean? Read them like this:

- x `>` y means: x is greater than y
- x `<` y means: x is less than y
- x `<=` y means: x is less than or equal to y
- x `>=` y means: x is greater than or equal to y

Awesome! Wanna do one more? Try this:

command-line

```
>>> 6 > 2 and 2 < 3
True
>>> 3 > 2 and 2 < 1
False
>>> 3 > 2 or 2 < 1
True
```

You can give Python as many numbers to compare as you want, and it will give you an answer! Pretty smart, right?

- **and** – if you use the `and` operator, both comparisons have to be True in order for the whole command to be True
- **or** – if you use the `or` operator, only one of the comparisons has to be True in order for the whole command to be True

Have you heard of the expression "comparing apples to oranges"? Let's try the Python equivalent:

command-line

```
>>> 1 > 'django'
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: '>' not supported between instances
of 'int' and 'str'
```

Here you see that just like in the expression, Python is not able to compare a number (`int`) and a string (`str`). Instead, it shows a **TypeError** and tells us the two types can't be compared together.

Boolean

Incidentally, you just learned about a new type of object in Python. It's called **Boolean**, and it is probably the easiest type there is.

There are only two Boolean objects:

- True
- False

But for Python to understand this, you need to always write it as 'True' (first letter uppercase, with the rest of the letters lowercased). **true, TRUE, and tRUE won't work – only True is correct.** (The same applies to 'False' as well, of course.)

Booleans can be variables, too! See here:

command-line

```
>>> a = True
>>> a
True
```

You can also do it this way:

command-line

```
>>> a = 2 > 5
>>> a
False
```

Practice and have fun with Booleans by trying to run the following commands:

- `True and True`
- `False and True`
- `True or 1 == 1`
- `1 != 2`

Congrats! Booleans are one of the coolest features in programming, and you just learned how to use them!

> For readers at home: this part is covered in the Python Basics: Saving files and "If" statement video.

So far we've been writing all our python code in the interpreter, which limits us to entering one line of code at a time. Normal programs are saved in files and executed by our programming language **interpreter** or **compiler**. So far we've been running our programs one line at a time in the Python **interpreter**. We're going to need more than one line of code for the next few tasks, so we'll quickly need to:

- Exit the Python interpreter
- Open up our code editor of choice
- Save some code into a new python file
- Run it!

To exit from the Python interpreter that we've been using, simply type the `exit()` function

command-line

```
>>> exit()
$
```

This will put you back into the command prompt.

Earlier, we picked out a code editor from the code editor section. We'll need to open the editor now and write some code into a new file:

editor

```
print('Hello, Django girls!')
```

Obviously, you're a pretty seasoned Python developer now, so feel free to write some code that you've learned today.

Now we need to save the file and give it a descriptive name. Let's call the file **python_intro.py** and save it to your desktop. We can name the file anything we want, but the important part here is to make sure the file ends in **.py**. The **.py** extension tells our operating system that this is a **Python executable file** and Python can run it.

> **Note** You should notice one of the coolest thing about code editors: colors! In the Python console, everything was the same color; now you should see that the `print` function is a different color from the string. This is called "syntax highlighting", and it's a really useful feature when coding. The color of things will give you hints, such as unclosed strings or a typo in a keyword name (like the `def` in a function, which we'll see below). This is one of the reasons we use a code editor. :)

With the file saved, it's time to run it! Using the skills you've learned in the command line section, use the terminal to **change directories** to the desktop.

On a Mac, the command will look something like this:

command-line

```
$ cd ~/Desktop
```

On Linux, it will be like this (the word "Desktop" might be translated to your local language):

command-line

```
$ cd ~/Desktop
```

Change directory: Windows Command Prompt

On Windows Command Prompt, it will be like this:

command-line

```
> cd %HomePath%\Desktop
```

Change directory: Windows Powershell

And on Windows Powershell, it will be like this:

command-line

```
> cd $Home\Desktop
```

If you get stuck, just ask for help.

Now use Python to execute the code in the file like this:

command-line

```
$ python3 python_intro.py
Hello, Django girls!
```

Note: on Windows 'python3' is not recognized as a command. Instead, use 'python' to execute the file:

command-line

```
> python python_intro.py
```

Alright! You just ran your first Python program that was saved to a file. Feel awesome?

You can now move on to an essential tool in programming:

If … elif … else

Lots of things in code should be executed only when given conditions are met. That's why Python has something called **if statements**.

Replace the code in your **python_intro.py** file with this:

python_intro.py

```
if 3 > 2:
```

If we were to save and run this, we'd see an error like this:

command-line

```
$ python3 python_intro.py
```

```
File "python_intro.py", line 2
         ^
SyntaxError: unexpected EOF while parsing
```

Python expects us to give further instructions to it which are executed if the condition `3 > 2` turns out to be true (or `True` for that matter). Let's try to make Python print "It works!". Change your code in your **python_intro.py** file to this:

python_intro.py

```
if 3 > 2:
    print('It works!')
```

Notice how we've indented the next line of code by 4 spaces? We need to do this so Python knows what code to run if the result is true. You can do one space, but nearly all Python programmers do 4 to make things look neat. A single `tab` will also count as 4 spaces.

Save it and give it another run:

command-line

```
$ python3 python_intro.py
It works!
```

Note: Remember that on Windows, 'python3' is not recognized as a command. From now on, replace 'python3' with 'python' to execute the file.

What if a condition isn't True?

In previous examples, code was executed only when the conditions were True. But Python also has `elif` and `else` statements:

python_intro.py

```
if 5 > 2:
    print('5 is indeed greater than 2')
else:
    print('5 is not greater than 2')
```

When this is run it will print out:

command-line

```
$ python3 python_intro.py
5 is indeed greater than 2
```

If 2 were a greater number than 5, then the second command would be executed. Easy, right? Let's see how `elif` works:

python_intro.py

```
name = 'Sonja'
if name == 'Ola':
    print('Hey Ola!')
elif name == 'Sonja':
    print('Hey Sonja!')
else:
    print('Hey anonymous!')
```

and executed:

command-line

```
$ python3 python_intro.py
Hey Sonja!
```

See what happened there? `elif` lets you add extra conditions that run if the previous conditions fail.

You can add as many `elif` statements as you like after your initial `if` statement. For example:

python_intro.py

```
volume = 57
if volume < 20:
    print("It's kinda quiet.")
elif 20 <= volume < 40:
    print("It's nice for background music")
elif 40 <= volume < 60:
    print("Perfect, I can hear all the details")
elif 60 <= volume < 80:
    print("Nice for parties")
elif 80 <= volume < 100:
    print("A bit loud!")
else:
    print("My ears are hurting! :(")
```

Python runs through each test in sequence and prints:

command-line

```
$ python3 python_intro.py
Perfect, I can hear all the details
```

Comments are lines beginning with `#`. You can write whatever you want after the `#` and Python will ignore it. Comments can make your code easier for other people to understand.

Let's see how that looks:

python_intro.py

```

if volume < 20 or volume > 80:
    volume = 50
    print("That's better!")
```

You don't need to write a comment for every line of code, but they are useful for explaining why your code is doing something, or providing a summary when it's doing something complex.

Summary

In the last few exercises you learned about:

- **comparing things** – in Python you can compare things by using `>`, `>=`, `==`, `<=`, `<` and the `and`, `or` operators
- **Boolean** – a type of object that can only have one of two values: `True` or `False`
- **Saving files** – storing code in files so you can execute larger programs.
- **if … elif … else** – statements that allow you to execute code only when certain conditions are met.
- **comments** - lines that Python won't run which let you document your code

Time for the last part of this chapter!

Your own functions!

> For readers at home: this part is covered in the Python Basics: Functions video.

Remember functions like `len()` that you can execute in Python? Well, good news – you will learn how to write your own functions now!

A function is a sequence of instructions that Python should execute. Each function in Python starts with the keyword `def`, is given a name, and can have some parameters. Let's start with an easy one. Replace the code in **python_intro.py** with the following:

python_intro.py

```
def hi():
    print('Hi there!')
    print('How are you?')

hi()
```

Okay, our first function is ready!

You may wonder why we've written the name of the function at the bottom of the file. This is because Python reads the file and executes it from top to bottom. So in order to use our function, we have to re-write it at the bottom.

Let's run this now and see what happens:

command-line

```
$ python3 python_intro.py
Hi there!
How are you?
```

Note: if it didn't work, don't panic! The output will help you to figure why:

- If you get a `NameError`, that probably means you typed something wrong, so you should check that you used the same name when creating the function with `def hi():` and when calling it with `hi()`.
- If you get an `IndentationError`, check that both of the `print` lines have the same whitespace at the start of a line: python wants all the code inside the function to be neatly aligned.
- If there's no output at all, check that the last `hi()` *isn't* indented - if it is, that line will become part of the function too, and it will never get run.

Let's build our first function with parameters. We will use the previous example – a function that says 'hi' to the person running it – with a name:

python_intro.py

```
def hi(name):
```

As you can see, we now gave our function a parameter that we called `name`:

python_intro.py

```
def hi(name):
    if name == 'Ola':
        print('Hi Ola!')
    elif name == 'Sonja':
        print('Hi Sonja!')
    else:
        print('Hi anonymous!')

hi()
```

Remember: The `print` function is indented four spaces within the `if` statement. This is because the function runs when the condition is met. Let's see how it works now:

command-line

```
$ python3 python_intro.py
Traceback (most recent call last):
File "python_intro.py", line 10, in <module>
  hi()
TypeError: hi() missing 1 required positional
argument: 'name'
```

Oops, an error. Luckily, Python gives us a pretty useful error message. It tells us that the function `hi()` (the one we defined) has one required argument (called `name`) and that we forgot to pass it when calling the function. Let's fix it at the bottom of the file:

python_intro.py

```
hi("Ola")
```

And run it again:

command-line

```
$ python3 python_intro.py
Hi Ola!
```

And if we change the name?

python_intro.py

```
hi("Sonja")
```

And run it:

command-line

```
$ python3 python_intro.py
Hi Sonja!
```

Now, what do you think will happen if you write another name in there? (Not Ola or Sonja.) Give it a try and see if you're right. It should print out this:

command-line

```
Hi anonymous!
```

This is awesome, right? This way you don't have to repeat yourself every time you want to change the name of the person the function is supposed to greet. And that's exactly why we need functions – you never want to repeat your code!

Let's do something smarter – there are more names than two, and writing a condition for each would be hard, right?

python_intro.py

```
def hi(name):
    print('Hi ' + name + '!')


hi("Rachel")
```

Let's call the code now:

command-line

```
$ python3 python_intro.py
Hi Rachel!
```

Congratulations! You just learned how to write functions! :)

Loops

> For readers at home: this part is covered in the Python Basics: For Loop video.

This is the last part already. That was quick, right? :)

Programmers don't like to repeat themselves. Programming is all about automating things, so we don't want to greet every person by their name manually, right? That's where loops come in handy.

Still remember lists? Let's do a list of girls:

python_intro.py

```
girls = ['Rachel', 'Monica', 'Phoebe', 'Ola',
'You']
```

We want to greet all of them by their name. We have the `hi` function to do that, so let's use it in a loop:

python_intro.py

```
for name in girls:
```

The `for` statement behaves similarly to the `if` statement; code below both of these need to be indented four spaces.

Here is the full code that will be in the file:

python_intro.py

```
def hi(name):
    print('Hi ' + name + '!')


girls = ['Rachel', 'Monica', 'Phoebe', 'Ola',
'You']
for name in girls:
    hi(name)
    print('Next girl')
```

And when we run it:

command-line

```
$ python3 python_intro.py
Hi Rachel!
Next girl
Hi Monica!
Next girl
Hi Phoebe!
Next girl
Hi Ola!
Next girl
Hi You!
Next girl
```

As you can see, everything you put inside a `for` statement with an indent will be repeated for every element of the list `girls`.

You can also use `for` on numbers using the `range` function:

python_intro.py

```
for i in range(1, 6):
    print(i)
```

Which would print:

command-line

```
1
2
3
4
5
```

`range` is a function that creates a list of numbers following one after the other (these numbers are provided by you as parameters).

Note that the second of these two numbers is not included in the list that is output by Python (meaning `range(1, 6)` counts from 1 to 5, but does not include the number 6). That is because "range" is half-open, and by that we mean it includes the first value, but not the last.

Summary

That's it. **You totally rock!** This was a tricky chapter, so you should feel proud of yourself. We're definitely proud of you for making it this far!

You might want to briefly do something else – stretch, walk around for a bit, rest your eyes – before going on to the next chapter. :)

tutorial.djangogirls.org

Python installation · Django Girls Tutorial

DjangoGirls

5-7 minuti

We're finally here!

But first, let us tell you what Python is. Python is a very popular programming language that can be used for creating websites, games, scientific software, graphics, and much, much more.

Python originated in the late 1980s and its main goal is to be readable by human beings (not only machines!). This is why it looks much simpler than other programming languages. This makes it easy to learn, but don't worry – Python is also really powerful!

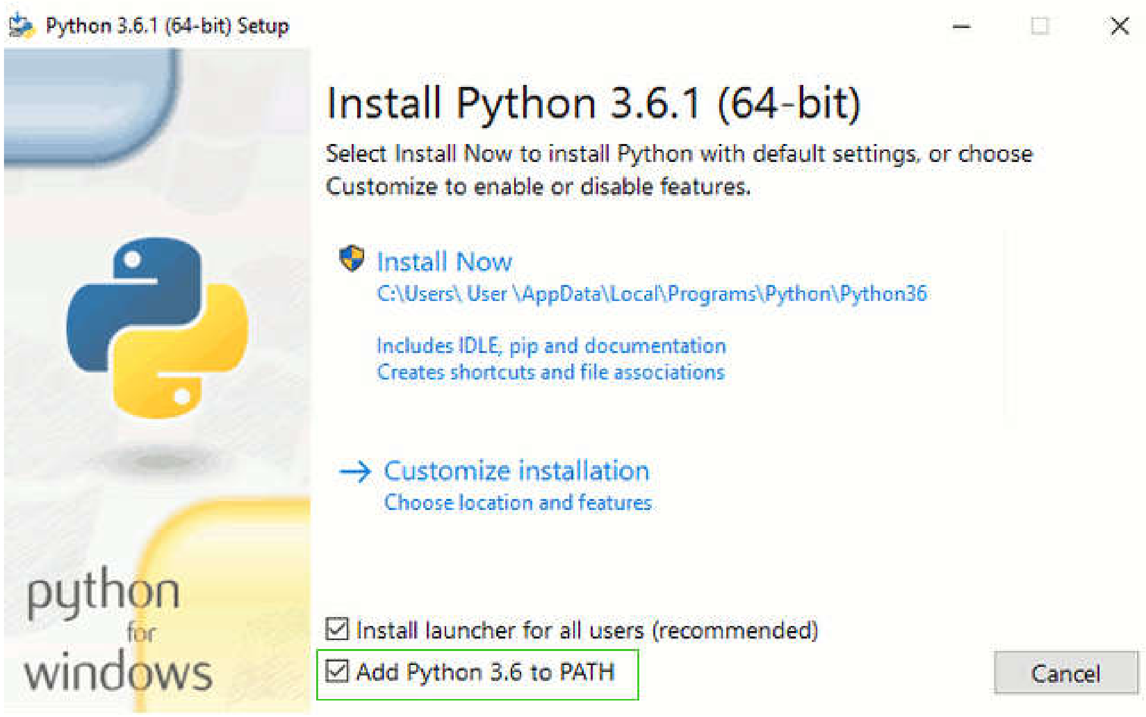
> **Note** If you're using a Chromebook, skip this chapter and make sure you follow the Chromebook Setup instructions.
>
> **Note** If you already worked through the Installation steps, there's no need to do this again – you can skip straight ahead to the next chapter!
>
> For readers at home: this chapter is covered in the Installing Python & Code Editor video.
>
> This section is based on a tutorial by Geek Girls Carrots (https://github.com/ggcarrots/django-carrots)

Django is written in Python. We need Python to do anything in

Django. Let's start by installing it! We want you to install Python 3.6, so if you have any earlier version, you will need to upgrade it.

First check whether your computer is running a 32-bit version or a 64-bit version of Windows, by pressing the Windows key + Pause/Break key which will open your System info, and look at the "System type" line. You can download Python for Windows from the website https://www.python.org/downloads/windows/. Click on the "Latest Python 3 Release - Python x.x.x" link. If your computer is running a **64-bit** version of Windows, download the **Windows x86-64 executable installer**. Otherwise, download the **Windows x86 executable installer**. After downloading the installer, you should run it (double-click on it) and follow the instructions there.

One thing to watch out for: During the installation you will notice a window marked "Setup". Make sure you tick the "Add Python 3.6 to PATH" checkbox and click on "Install Now", as shown here:


Python 3.6.1 (64-bit) Setup

Install Python 3.6.1 (64-bit)

Select Install Now to install Python with default settings, or choose Customize to enable or disable features.

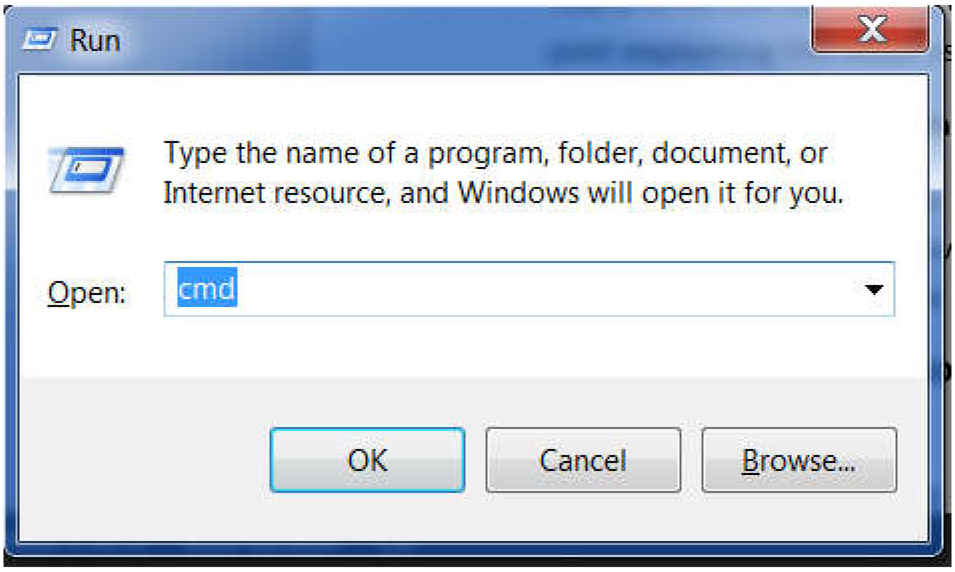
Install Now
C:\Users\ User \AppData\Local\Programs\Python\Python36

Includes IDLE, pip and documentation
Creates shortcuts and file associations

→ Customize installation
Choose location and features

☑ Install launcher for all users (recommended)
☑ Add Python 3.6 to PATH

In upcoming steps, you'll be using the Windows Command Line (which we'll also tell you about). For now, if you need to type in some commands, go to Start menu → Windows System → Command Prompt. You can also hold in the Windows key and press the "R"-key until the "Run" window pops up. To open the Command Line, type "cmd" and press enter in the "Run" window. (On newer versions of Windows, you might have to search for "Command Prompt" since it's sometimes hidden.)



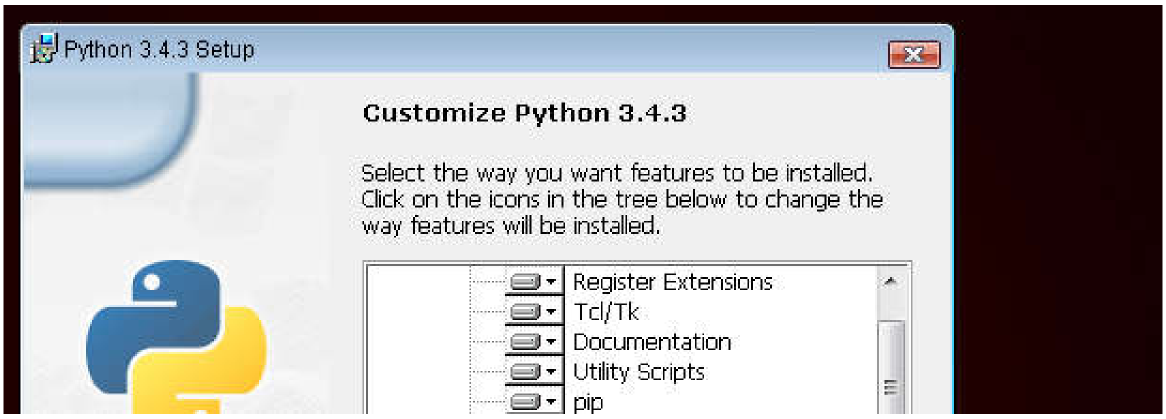
Type the name of a program, folder, document, or Internet resource, and Windows will open it for you.

Open: cmd

Note: if you are using an older version of Windows (7, Vista, or any older version) and the Python 3.6.x installer fails with an error, you can try either:

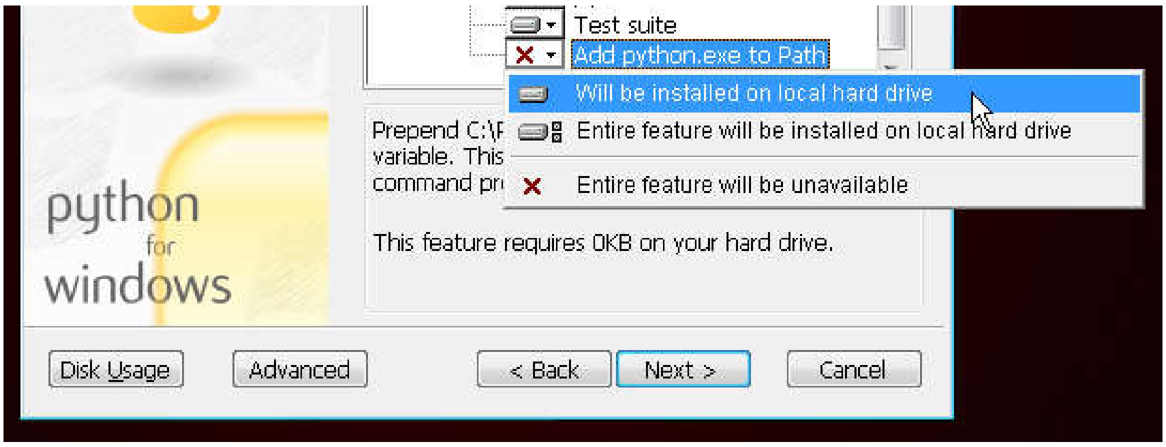
1. install all Windows Updates and try to install Python 3.6 again; or
2. install an older version of Python, e.g., 3.4.6.

If you install an older version of Python, the installation screen may look a bit different than shown above. Make sure you scroll down to see "Add python.exe to Path", then click the button on the left and pick "Will be installed on local hard drive":



Customize Python 3.4.3

Select the way you want features to be installed. Click on the icons in the tree below to change the way features will be installed.


Test suite
Add python.exe to Path
Will be installed on local hard drive
Entire feature will be installed on local hard drive
Entire feature will be unavailable
Prepend C:\F
variable. This
command pr
This feature requires 0KB on your hard drive.
python for windows
Disk Usage
Advanced
< Back
Next >
Cancel

> **Note** Before you install Python on OS X, you should ensure your Mac settings allow installing packages that aren't from the App Store. Go to System Preferences (it's in the Applications folder), click "Security & Privacy," and then the "General" tab. If your "Allow apps downloaded from:" is set to "Mac App Store," change it to "Mac App Store and identified developers."

You need to go to the website https://www.python.org/downloads/release/python-361/ and download the Python installer:

- Download the *Mac OS X 64-bit/32-bit installer* file,
- Double click *python-3.6.1-macosx10.6.pkg* to run the installer.

It is very likely that you already have Python installed out of the box. To check if you have it installed (and which version it is), open a console and type the following command:

command-line

```
$ python3 --version
Python 3.6.1
```

If you have a different 'micro version' of Python installed, e.g. 3.6.0, then you don't have to upgrade. If you don't have Python installed, or if you want a different version, you can install it as follows:

Install Python: Debian or Ubuntu

Type this command into your console:

command-line

```
$ sudo apt-get install python3.6
```

Use this command in your console:

command-line

```
$ sudo dnf install python3
```

If you're on older Fedora versions you might get an error that the command dnf is not found. In that case you need to use yum instead.

Use this command in your console:

command-line

```
$ sudo zypper install python3
```

Verify the installation was successful by opening a command prompt and running the `python3` command:

command-line

```
$ python3 --version
Python 3.6.1
```

NOTE: If you're on Windows and you get an error message that `python3` wasn't found, try using `python` (without the `3`) and check if it still might be a version of Python 3.6.

If you have any doubts, or if something went wrong and you have no idea what to do next, please ask your coach! Sometimes things don't go smoothly and it's better to ask for help from someone with more experience.

```
# Compara due interi tra di loro e stamp il ris.
def ritorna_il_piu_grande(x,y):
	if x > y:
		#print(str(x) + " è più grande di " + str(y))
		return x
	# elif x == y:
	#	print(str(x) + " è uguale a " + str(y))
	else:
		#print(str(x) + " è più piccolo di " + str(y))
		return y

# Stampa in loop i numeri fino a max
def stampa_numeri(max):
	for i in range(0,max):
		print(i)

#stampa_numeri(10)

# definisco lista di alunni
alunni = ['Giuliano','Robert',
			'Danile', 'Melania',
			'Vanessa', 'Simona',
			'Lucas', 'Andrea']

def add_students():
	# finché gli alunni sono meno di 10, aggiungi nuovo
	while len(alunni) < 10:
		alunni.append('Alunno')

	# Espelliamo Andrea a caso
	alunni.pop(7)
	alunni.sort()

	for nome in alunni:
		print(nome)

#add_students()

# Saluta tutti
def salutatore():
	for nome in alunni:
		print("Ciao " + nome)
	return None

#salutatore()

# Funzione che restituisce il quadrato dell'argomento
def f(x):
	y = x*x
	return y

y = f(2)

print(y)
```