Digital Career Institute

Python Course - Tools & Libraries

Digital Career Institute

Goal of the Submodule

The goal of this submodule is to help the learners to get familiar with programming tools that are necessary to comfortably work with Python. By the end of this submodule, the learners should be able to understand:

- What command line interface is.
- What the difference between module, script, package and library is.
- How to use and assess given library.
- How to work with pip.
- How to use imports.
- How to work with openpyxl.
- How to create your own package.

Topics

- Command Line Interface
- Modules, scripts, packages, libraries.
- General thoughts about libraries.
- Pip - package manager for Python
- Imports in Python.
- Working with Excel files using openpyxl.
- Working with packages.

Command Line Interface

Topics

- Read parameters in CLI context
- cmd
- sys
- getopt

Digital Career Institute

Command Line Interface (CLI)

Digital Career Institute

- CLI provides a way for a user to **interact** with a program running in a text-based shell interpreter.
- Some examples of shell interpreters are Bash on Linux or Command Prompt on Windows.
- A command line interface is enabled by the shell interpreter that exposes a command prompt.

Command prompt

Digital Career Institute

- A **command prompt** (or just *prompt*) is a sequence of (one or more) characters used in a command-line interface to indicate **readiness** to accept commands.

- It literally prompts the user to take action.

- A prompt usually ends with one of the characters $, %, #, : , > or - and often includes other information, such as the path of the current working directory and the hostname.

Command prompt

- **Command prompt** can be characterized by the following elements:
  - A **command** or program
  - Zero or more command line **arguments**
  - An **output** representing the result of the command
  - Textual documentation referred to as **usage** or **help**
  - Not every command line interface may provide all these elements

Command prompt - example no. 1

- In this example, the Python interpreter takes option -c for **command**, which says to execute the Python command line arguments following the option -c as a Python program.

```
artur@artur-MSI:~$ python3 -c "print('Welcome to DCI course')"
Welcome to DCI course
```

Command prompt - example no. 1

Digital Career Institute

- This example shows how to invoke Python with -h to display the help:

```
artur@artur-MSI:~$ python3 -h
usage: python3 [option] ... [-c cmd | -m mod | file | -] [arg] ...
Options and arguments (and corresponding environment variables):
-b     : issue warnings about str(bytes_instance), str(bytearray_instance)
         and comparing bytes/bytearray with str. (-bb: issue errors)
-B     : don't write .pyc files on import; also PYTHONDONTWRITEBYTECODE=x
-c cmd : program passed in as string (terminates option list)
-d     : debug output from parser; also PYTHONDEBUG=x
-E     : ignore PYTHON* environment variables (such as PYTHONPATH)
-h     : print this help message and exit (also --help)
```

Command Line Arguments

Digital Career Institute

- The arguments that are given after the name of the program in the command line shell of the operating system are known as **Command Line Arguments**. Python provides various ways of dealing with these types of arguments. The three most common are:
  - sys.argv
  - getopt module
  - argparse module

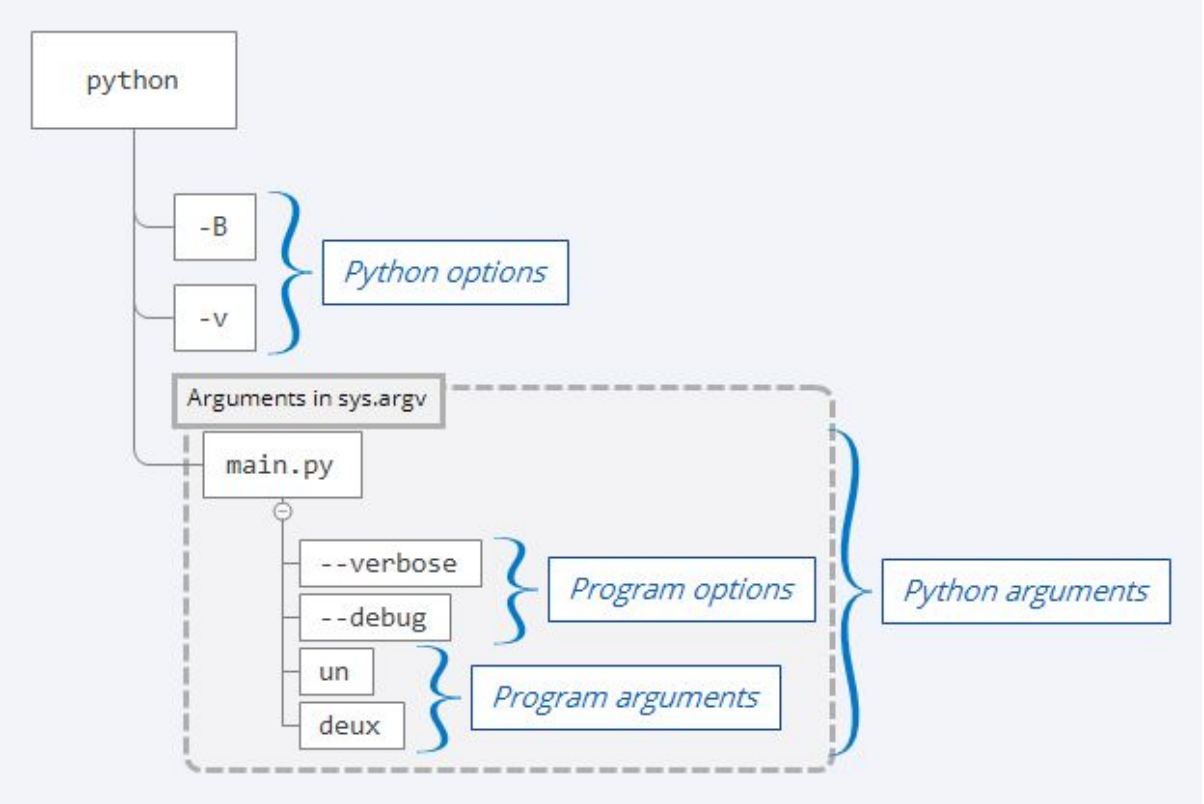
Python Command Line Arguments

Digital Career Institute

Python command line arguments are a subset of the command line interface. They can be composed of different types of arguments:

1. **Options** modify the behavior of a particular command or program.
2. **Arguments** represent the source or destination to be processed.
3. **Subcommands** allow a program to define more than one command with the respective set of options and arguments.

Python Command Line Arguments



sys module

Digital Career Institute

- The **sys module** in Python provides various functions and variables that are used to manipulate different parts of the Python runtime environment.

- It allows operating on the interpreter as it provides access to the variables and functions that interact strongly with the interpreter.

sys.argv

Digital Career Institute

- **argv** is a variable provided by the **sys** module which holds a list of all the arguments passed to the command line (including the script name).

- So even if you don't pass any arguments to your script. The argv variable always contains **at least one** element i.e the script name.

- The arguments in argv are always parsed as **string**. So be careful if you are expecting your input to be of any other type. You may need to cast or convert the elements according to your requirements.

sys.argv

Digital Career Institute

- The sys module exposes an array named **argv** that includes the following:
  - **argv[0]** contains the name of the current Python program.
  - **argv[1:]**, the rest of the list, contains any and all Python command line arguments passed to the program.

sys.argv - example

```
1 # argv.py
2 import sys
3
4 print(f"Name of the script      : {sys.argv[0]=}")
5 print(f"Arguments of the script : {sys.argv[1:]=}")
```

- **Line 2** imports the internal Python module sys.
- **Line 4** extracts the name of the program by accessing the first element of the list sys.argv.
- **Line 5** displays the Python command line arguments by fetching all the remaining elements of the list sys.argv.

sys.argv - example

- After execution of code in file argv.py :

```
artur@artur-MSI:~/Desktop/DCI$ python3 argv.py some arguments here
Name of the script      : sys.argv[0]='argv.py'
Arguments of the script : sys.argv[1:]=['some', 'arguments', 'here']
```

sys.argv - summing arguments (code)

Digital Career Institute

```python
import sys

# total arguments
n = len(sys.argv)
print("Total arguments passed:", n)

# Arguments passed
print("Name of Python script:", sys.argv[0])

# Addition of numbers
sum_of_arguments = 0

for i in range(1, n):
    sum_of_arguments += int(sys.argv[i])

print("Result:", sum_of_arguments)

```

sys.argv - summing arguments (results)

Digital Career Institute

```
artur@artur-MSI:~/Desktop/DCI$ python3 argv-summing.py 1 2 3 4
Total arguments passed: 5
Name of Python script: argv-summing.py
Result: 10
```

sys.argv

- sys.argv is of the type **<list>** so you can access the elements just as you would from any other list. For example, **sys.argv[1]**

- It does not provide any inherent mechanism to make any of the arguments as required or optional and we also cannot limit the number of arguments supplied to our script.

- Sys.argv can be more than sufficient if your problem definition is simple enough. But if your requirements are a bit more advanced than just adding two numbers, you may need to use **getopt** or **argparse**.

getopt module

- **getopt** provides us with features that make it **easier** to process command line arguments in Python.
- **getopt** is a module that comes bundled with any standard python installation and therefore you **need not** install it explicitly.
- A major advantage of getopt over just using sys.argv is getopt supports **switch style options** (for example: -s or --sum).
- Hence getopt supported options are **position-independent**. The example $ ls -li works the same as $ ls -il

getopt module

Digital Career Institute

- The options are of two types:
    - Options that need a value to be passed with them. These are defined by the option name suffixed with **=** (for example: **num1=**)
    - Options that behave as a flag and do not need a value. These are defined by passing the option name **without the suffix =** (for example: --subtract)

getopt module

Digital Career Institute

- The options can have two variations:
    - **shortopts** are one letter options, denoted by prefixing a single - to an option name (for example, $ ls -l)
    - **longopts** are a more descriptive representation of an option, denoted by prefixing two – to an option name (for example, $ ls --long-list)

getopt (usage)

Digital Career Institute

- getopt module provides a **getopt**(args, shortopts, longopts=[]) function which we can use to define our options:
- Code of getopt() **function** usage:
  (opts, args) = getopt.getopt(sys.argv[1:], 'ha:b:s', ['help','num1=', 'num2=', 'subtract'])
- sys.argv holds the unformatted list of all the arguments passed to a python script.

getopt (usage)

Digital Career Institute

- There are two variables used (**opts**, **args**) because getopt.getopt function returns two elements:

    - one containing a <list> of **options**
    - second has a <list> of **arguments** that are not specified in our getopt initialization.

getopt (usage)

Digital Career Institute

- You can specify **shortopts** as a colon(:) separated single letter characters.
- You can specify **longopts** as a comma-separated list of words with the suffix **=**
- **longopts** without the suffix = are considered as **a flag** and they should be passed without any value.
- Now, to use the options passed to our program we can just iterate over the opts variable like any other list.

getopt (usage)

- Now, to use the options passed to our program we can just iterate over the opts variable like any other list:

```
for (o, a) in opts:
```

- Here **o** will hold our option name and **a** will have any value assigned to the option.

- Also notice that as **--subtract** is being used as a flag it will not have any value. This flag is used to decide whether to print the sum of the inputs **or** the difference in them.

getopt (usage)

Digital Career Institute

```
artur@artur-MSI:~/Desktop/DCI$ python3 get_opt.py -a 6 -b 7

 Sum of two numbers is : 13

artur@artur-MSI:~/Desktop/DCI$ python3 get_opt.py -a 6 -b 7 --subtract

 Difference in two numbers is : -1
```

- Full example is available under the link in documentation!

argparse module

Digital Career Institute

- The argparse module makes it easy to write user-friendly command-line interfaces.
- The program defines what arguments it requires, and argparse will figure out how to parse those out of sys.argv.
- The argparse module also automatically generates help and usage messages and issues errors when users give the program invalid arguments.

argparse (example)

Digital Career Institute

- The following code is a Python program that takes a list of integers and produces either the sum or the max:

```
import argparse

parser = argparse.ArgumentParser(description='Process some integers.')
parser.add_argument('integers', metavar='N', type=int, nargs='+',
                    help='an integer for the accumulator')
parser.add_argument('--sum', dest='accumulate', action='store_const',
                    const=sum, default=max,
                    help='sum the integers (default: find the max)')

args = parser.parse_args()
print(args.accumulate(args.integers))
```

argparse (example)

- Assuming the Python code above is saved into a file called prog.py, it can be run at the command line and provides useful help messages:

```
$ python prog.py -h
usage: prog.py [-h] [--sum] N [N ...]

Process some integers.

positional arguments:
 N           an integer for the accumulator

optional arguments:
 -h, --help  show this help message and exit
 --sum       sum the integers (default: find the max)
```

argparse (example)

- When run with the appropriate arguments, it prints either the sum or the max of the command-line integers:

```
$ python prog.py 1 2 3 4
4

$ python prog.py 1 2 3 4 --sum
10
```

argparse (example)

Digital Career Institute

- If invalid arguments are passed in, it will issue an error:

```
$ python prog.py a b c
usage: prog.py [-h] [--sum] N [N ...]
prog.py: error: argument N: invalid int value: 'a'
```

- You can find more information about argparse module in the official tutorial (link in the **documentation**)

Scripts

Minimal Example of Script

```
# name of this file is calculate.py
def add(a, b):
                                    return      a      +      b


result = add(3, 6)
print("The result is: ",result)
```

You can run the script **`calculate.py`** (from the previous slide) in a shell:

```
$ python3 calculate.py
The result is:  9
```

Using ‘__name__’ Variable in Scripts

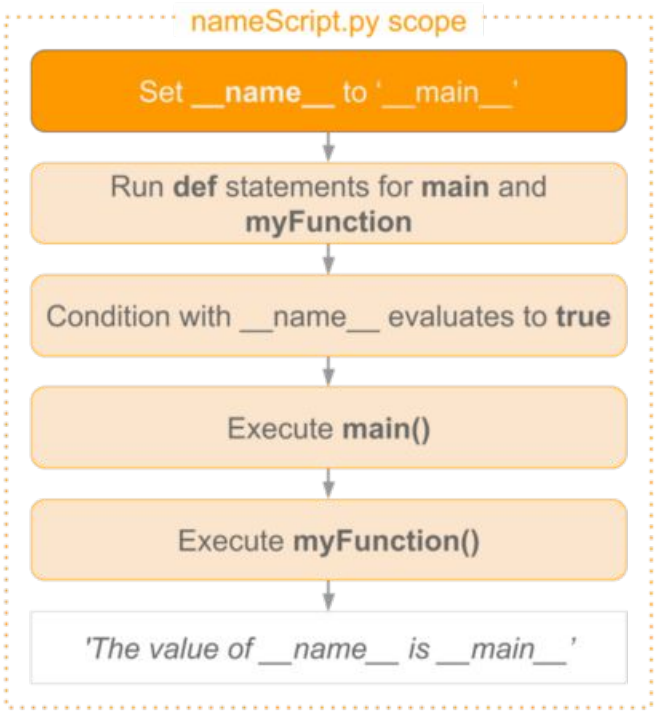
Digital Career Institute

- The `__name__` variable (two underscores before and after) is a special Python variable. It gets its value depending on how we execute the containing script.

- Functions of a script can be reused in other scripts by importing them.

- Thanks to this special variable, you can **decide** whether you want to run the script. Or that you want to import the functions defined in the script.

Scenario #1 of Using '__name__' Variable



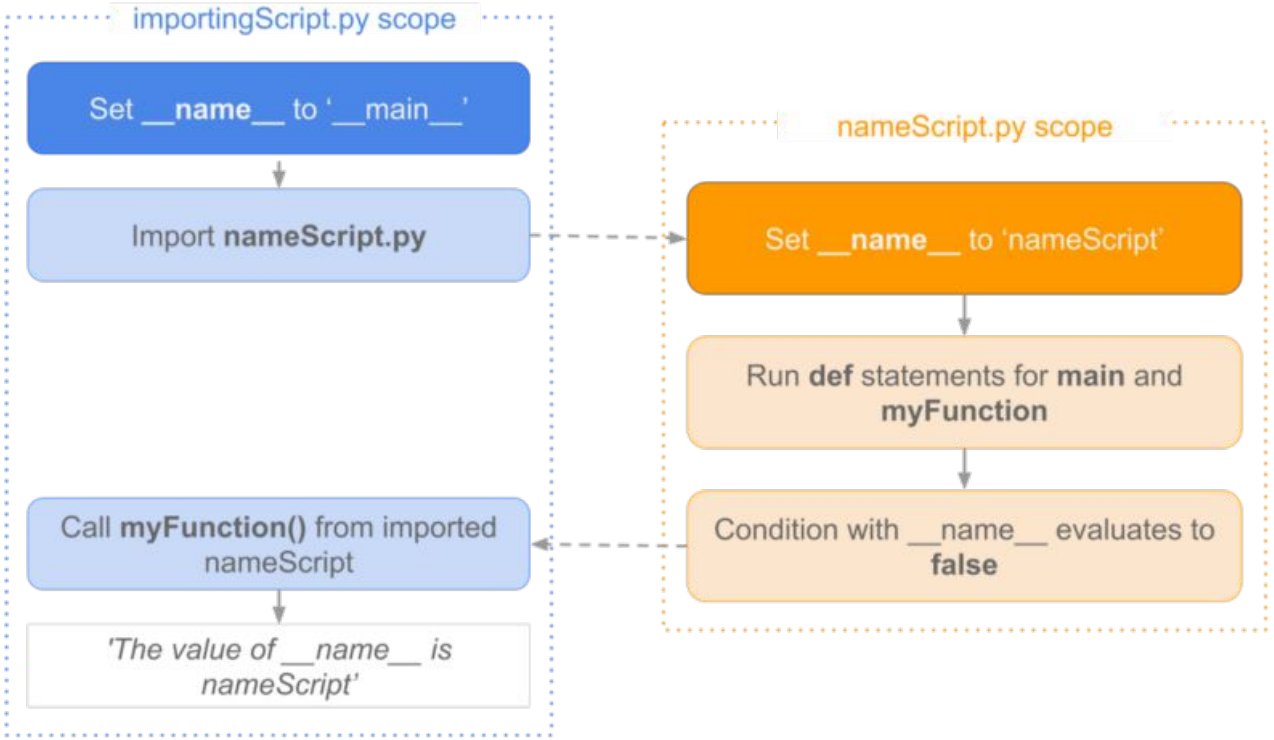
Scenario #2 of Using '__name__' Variable

Digital Career Institute

If we want to **re-use** `myFunction` in another script, for example `importingScript.py`, we can import `nameScript.py` as a module:

```
# importingScript.py

import nameScript as ns

ns.myFunction()
```

Scenario #2 of Using '__name__' Variable



Namespaces in Python

Namespace Basics

Digital Career Institute

- A **namespace** is a collection of currently defined symbolic names along with information about the object that each name references.

- You can think of a namespace as a **dictionary** in which the keys are the object names and the values are the objects themselves. Each key-value pair maps a name to its corresponding object.

Namespace Basics

Digital Career Institute

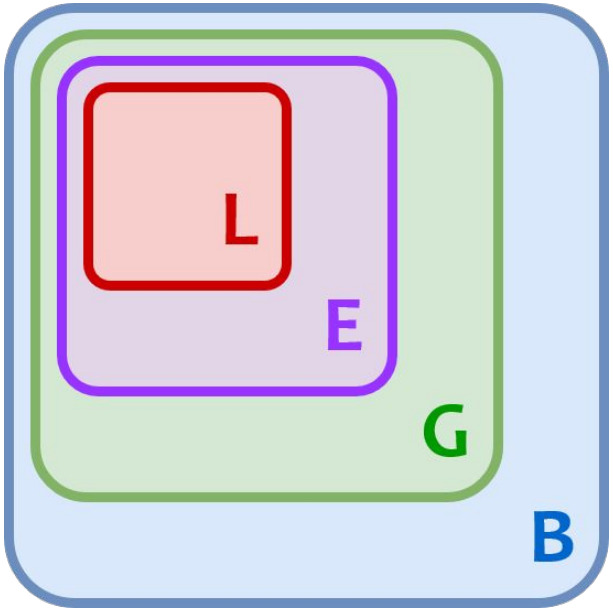
In a Python program, there are four types of **namespaces**:

- Built-In
- Global
- Enclosing
- Local

LEGB Rule

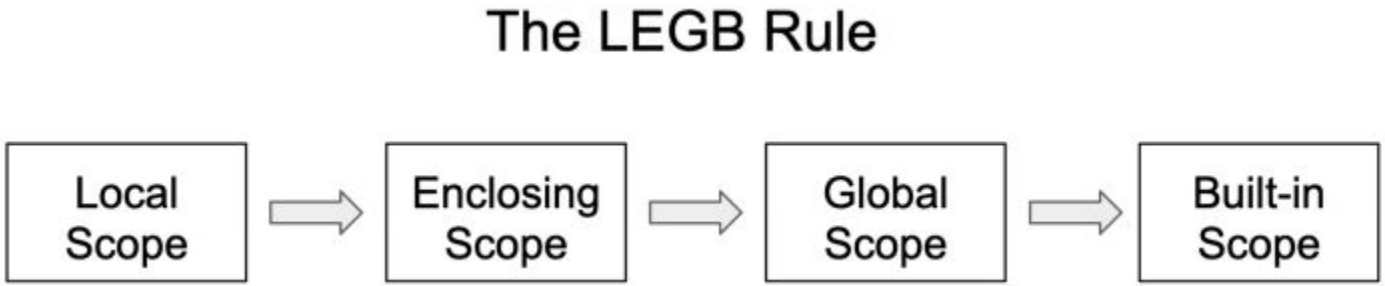
- The interpreter searches for a name from the inside out, looking in the **l**ocal, **e**nclosing, **g**lobal, and finally the **b**uilt-in scope:

- If the interpreter doesn't find the name in any of these locations, then Python raises a **NameError** exception.

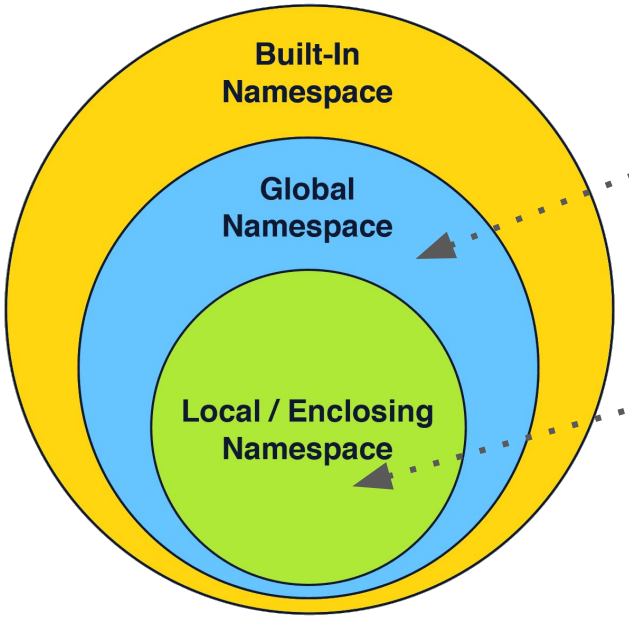


LEGB Rule

LEGB rule defines the **look-up order for given names**, as shown in the following diagram:



Basics




```
a = 'global a'
y = 'global y'

def test_namespace():
    a = 'enclosing a'

    def inner_namespace():
        a = 'local a'
        print(a)
        print(y)

    inner_namespace()

    print(a)

test_namespace()

print(a)
```

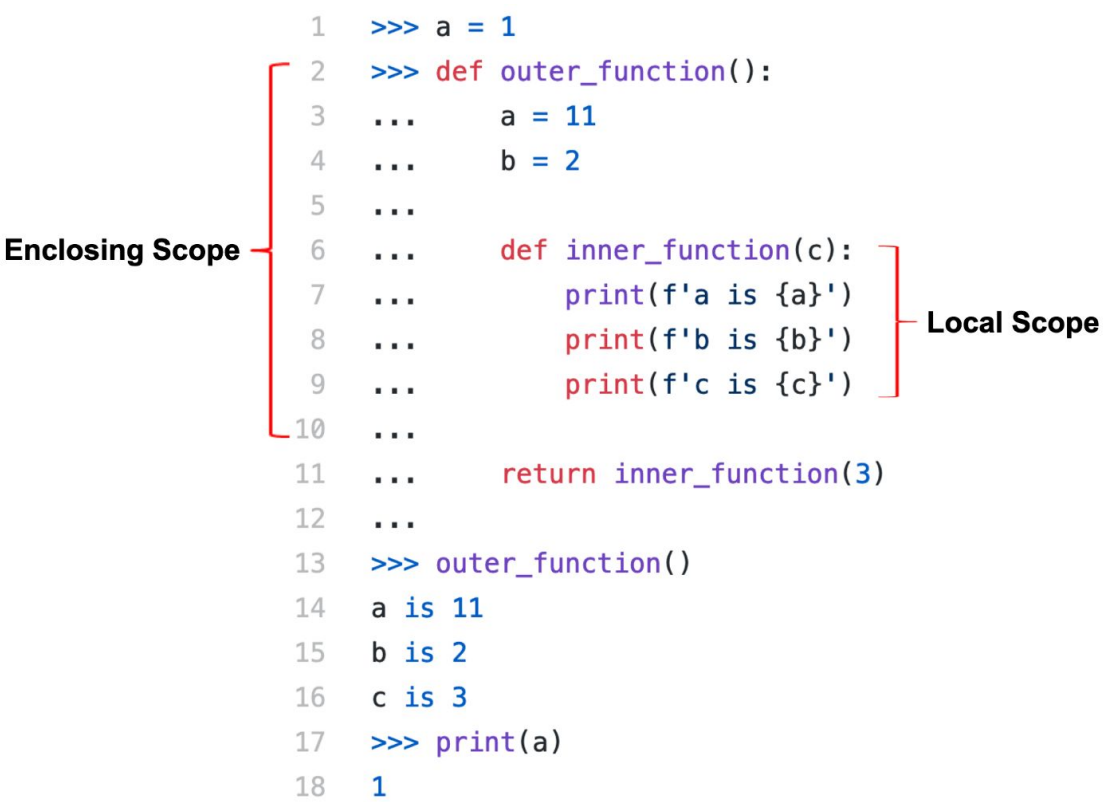
Global

Enclosing

Local

```
local a
global y
enclosing a
global a
```

Namespace Basics - Example #2




```
>>> a = 1
>>> def outer_function():
...     a = 11
...     b = 2
...
...     def inner_function(c):
...         print(f'a is {a}')
...         print(f'b is {b}')
...         print(f'c is {c}')
...
...     return inner_function(3)
...
>>> outer_function()
a is 11
b is 2
c is 3
>>> print(a)
1
```

Built-in Namespace

Digital Career Institute

- The **built-in namespace** contains the names of **all** of Python's **built-in objects**. These are available at all times when Python is running.

- You can list the objects in the built-in namespace with the following command:

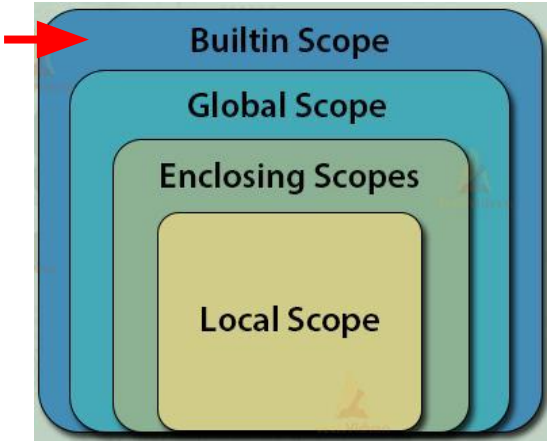
```
>>> dir(__builtins__)
['ArithmeticError', 'AssertionError', 'AttributeError', 'BaseException',
'BlockingIOError',...
```

Built-in Namespace

- You'll see some objects here that you may recognize from previous lectures, for example, built-in functions like `max()` and `len()`, and object types like `int` and `str`.

- The Python interpreter creates the built-in namespace when it starts up.

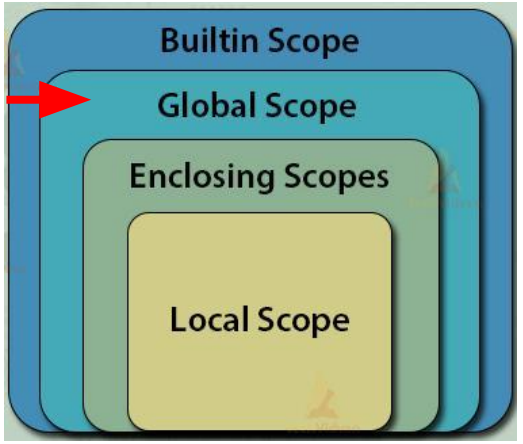
- This namespace remains in existence until the interpreter terminates.



Global Namespace

- The **global namespace** contains any names defined at the level of the main program.

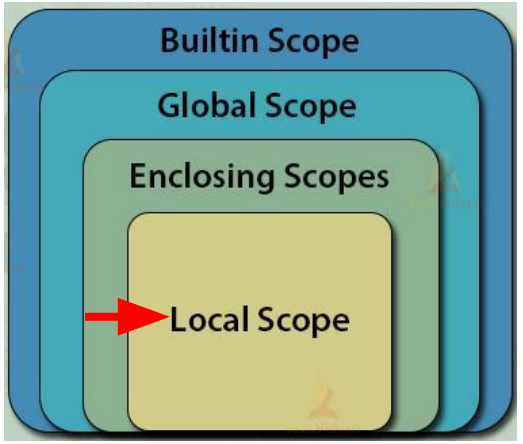
- Python creates the global namespace when the main program body **starts**, and it remains in existence until the interpreter terminates.



Local Namespace

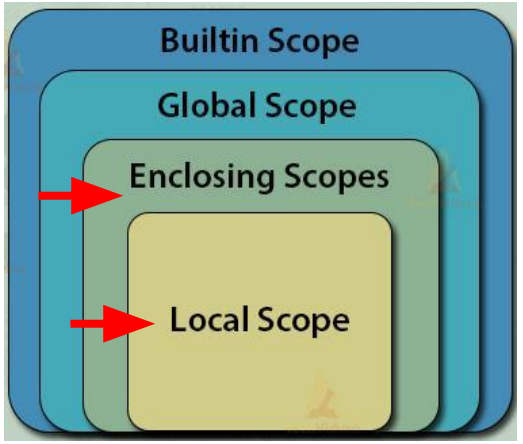
- The Python interpreter creates a new namespace **whenever** a function executes.

- That namespace is **local** to the function and remains in existence until the function terminates.



Local and Enclosing Namespaces

- Functions don’t exist independently from one another only at the level of the main program.
- You can also define one function inside another.
- Look at the example on the next slide.

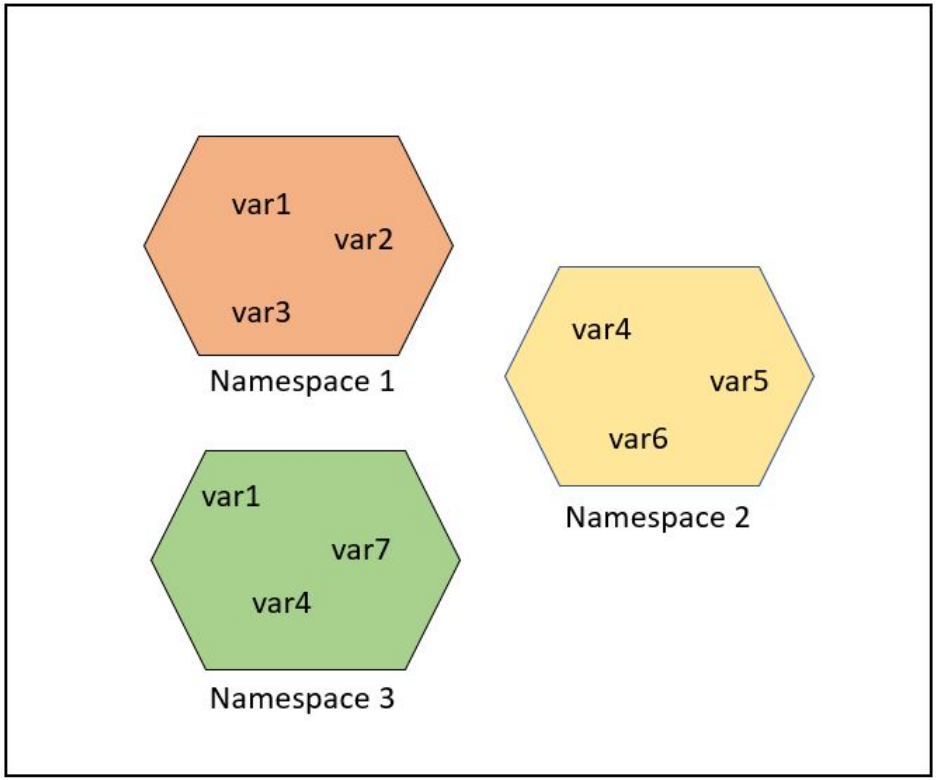


Local and Enclosing Namespaces

Digital Career Institute

```
def f():
    print('Start f()')
    def g():
        print('Start g()')
        print('End g()')
        return
    g()
    print('End f()')
    return
f()
```

Multiple Namespaces

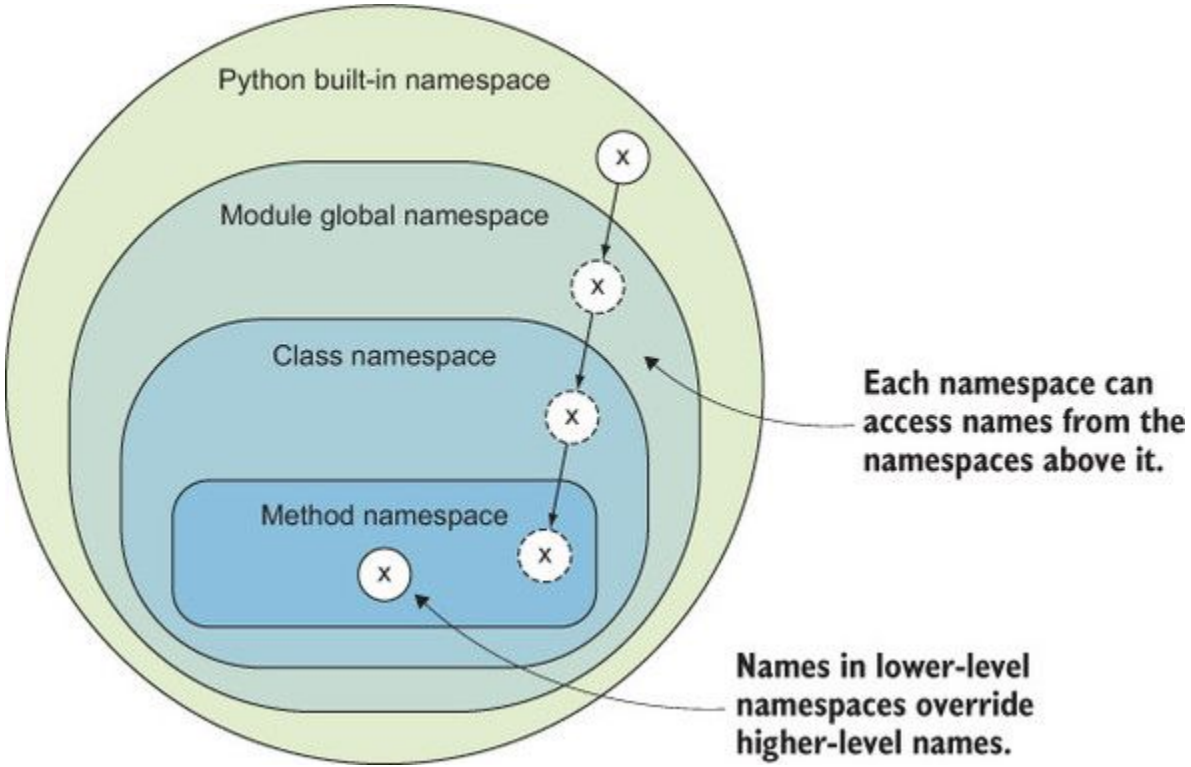


Content of the Namespace

Digital Career Institute

```
>>> dir()
['__annotations__', '__builtins__', '__doc__', '__loader__', '__name__',
'__package__', '__spec__']

>>> dir(math)

['__doc__', '__loader__', '__name__', '__package__', '__spec__', 'acos',
'acosh', 'asin', 'asinh', 'atan', 'atan2', 'atanh', 'ceil', 'comb',
'copysign', ..., 'lgamma', 'log', 'log10', 'log1p', 'log2', 'modf', 'nan',
'perm', 'pi', 'pow', 'prod', 'radians', 'remainder', 'sin', 'sinh', 'sqrt',
'tan', 'tanh', 'tau', 'trunc']
```

Variable Scope



Modules

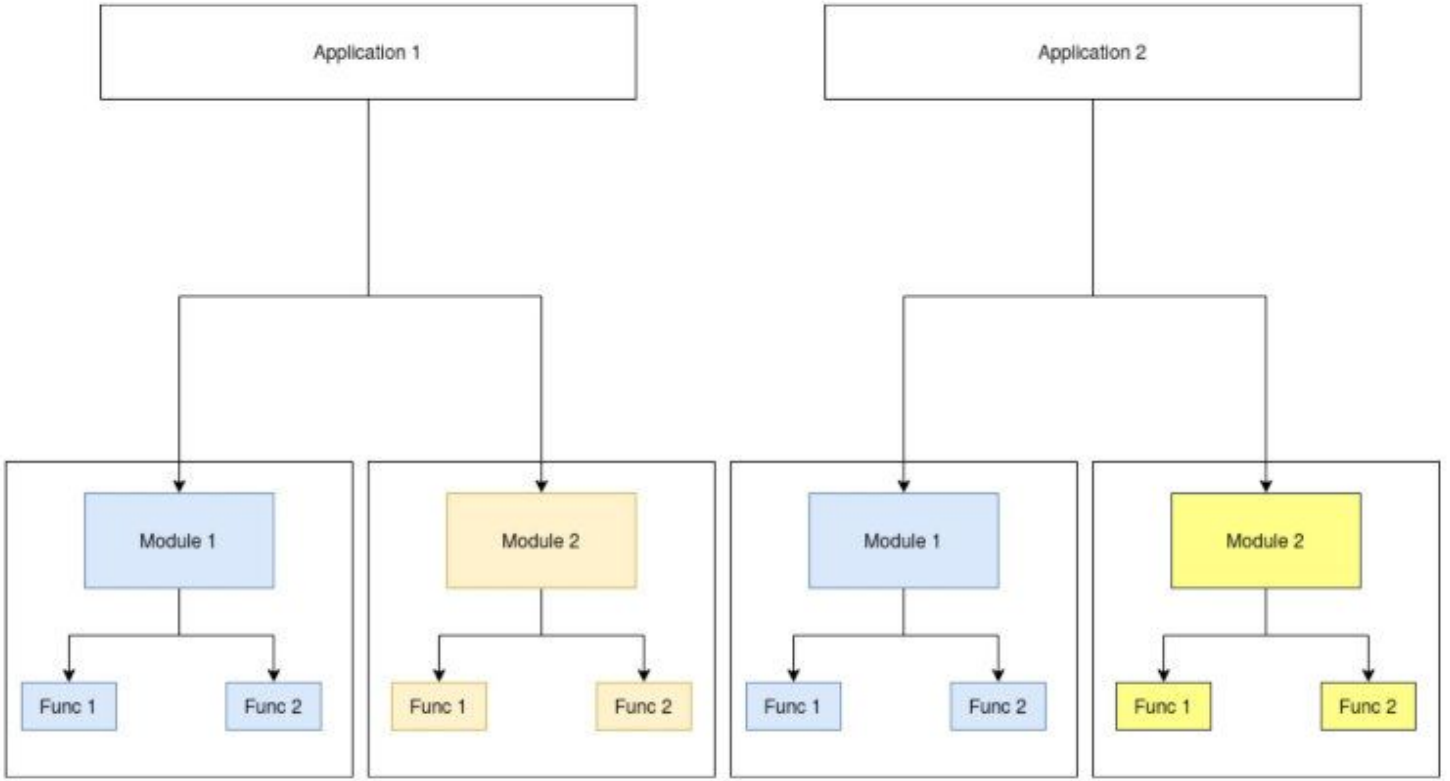
Modular Programming

Digital Career Institute

- **Modular programming** refers to the process of breaking a large, unwieldy programming task into separate, smaller, more manageable subtasks or **modules**.

- Individual modules can then be connected together like building blocks to create a **larger** application.

Modular Programming



Modular Programming

Digital Career Institute

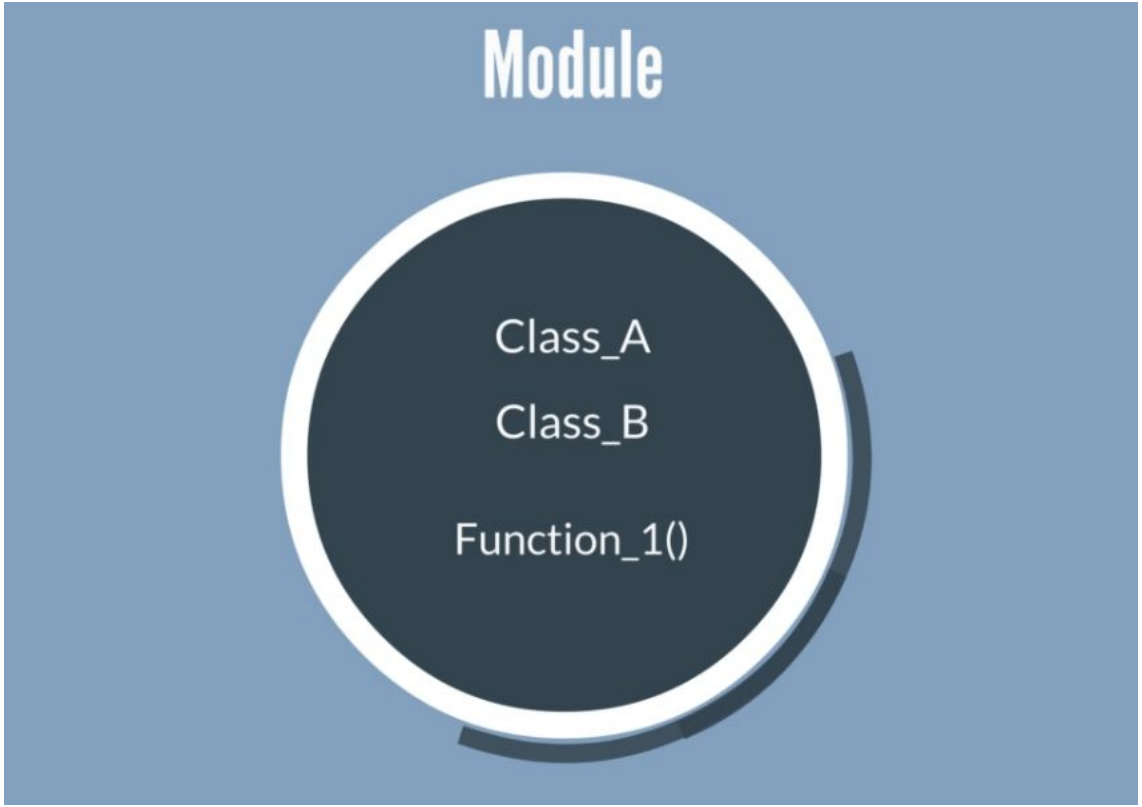
Pillars of modularized code	
Simplicity	Maintainability
Reusability	Scoping

Module Definition

The Python.org glossary defines **module** as follows:

- An object that serves as an organizational **unit** of Python code.
- Modules have a **namespace** containing arbitrary Python objects.
- Modules are loaded into Python by the process of **importing**.

The Module



Module Creation

There are actually three different ways to create a **module** in Python:

- A module can be **written** in **Python** itself.
- A module can be **written** in **C** and loaded dynamically at run-time, like for example the re (regular expression) module.
- A **built-in** module is intrinsically contained in the interpreter, like the for example `itertools` or `math` module.

A Module in Python

Digital Career Institute

- A **module** is a Python file that's intended to be **imported** into scripts or other modules.

- It often defines members like classes, functions, and variables intended to be used **in other** files that import it.

	Scripts	**Modules**
Executable	Yes	No
Importable	Yes	Yes

A Module - Basic Example

```
# name of this file is add.py

def add(a, b):

    return a + b
```

The module is **not intended** to be run directly!

Accessing the Module

Digital Career Institute

- A module's contents are accessed the same way in all cases mentioned before - with the **import** statement.

- In practice, a **module** usually corresponds to one .py file containing Python code.

- The **true power** of modules is that they can be **imported** and **reused** in other code.

Import Statements

Import Statements

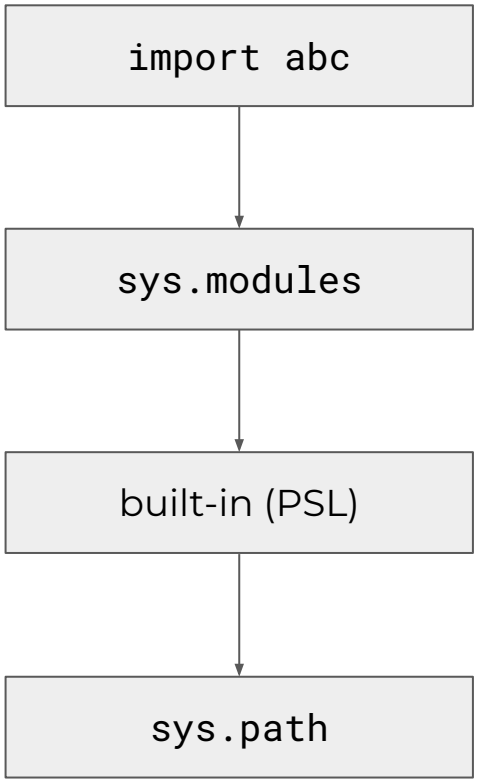
- **Module** contents are made available to the caller with the `import` statement.

- The **caller** of a module is the unit of program code in which the call to the module occurred.

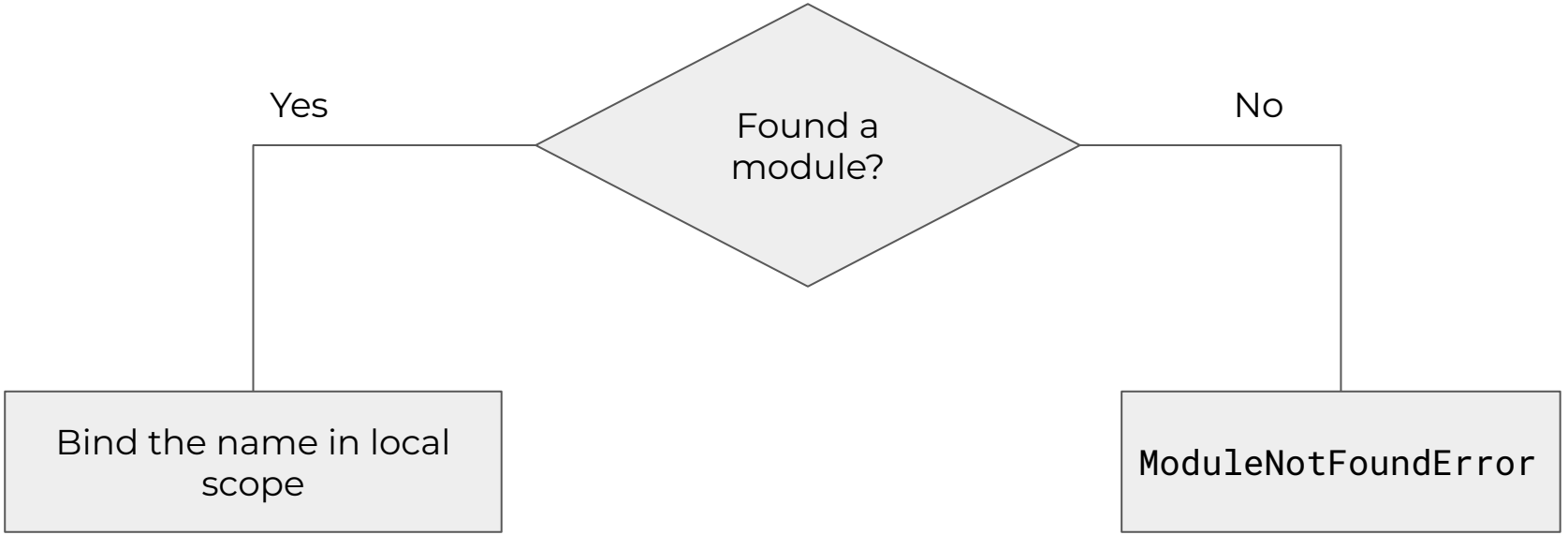
- The simplest form of importing module:

```
import <module_name>
```

How Import Works



How Import Works



Import Statements

```
import <module_name>
```

- Note that this ***does not*** make the module contents ***directly*** accessible to the caller.

- Each module has its own **private symbol table**, which serves as the global symbol table for all objects defined *in the module*.

- Thus, a module creates a separate **namespace**.

Import Objects from Module

Digital Career Institute

```
>>> import math

>>> math
# dot notation

>>> math.pi

3.141592653589793
```

To be accessed in the local context, names of objects defined in the module must be prefixed by the **name** of the module (**`math.pi`**).

Import Individual Objects

```
from <module_name> import <name(s)>
```

An alternate form of the import statement allows individual objects from the module to be imported **directly** into the caller's symbol table.

```
>>> from math import pi
>>> pi
3.141592653589793
```

Import all Objects from Module

```
from <module_name> import *
```

- This will place the names of **all** objects from `<module_name>` into the **local** symbol table, with the exception of any that begin with the underscore (_) character.

- This **isn't** necessarily **recommended** in large-scale production code. It's a bit dangerous because you are entering all names from given module into the local symbol table.

Import Object with Alternate Name

```
from <module_name> import <name> as <alt_name>
```

- It is also possible to import individual objects but enter them into the local symbol table with **alternate** names.

- This makes it possible to place names **directly** into the local symbol table but **avoid conflicts** with previously existing names.

```
>>> from math import pi as PI_NUMBER
>>> PI_NUMBER
3.141592653589793
```

Import Module with Alternate Name

```
import <module_name> as <alt_name>
```

You can also import an entire module under an alternate name:

```
>>> import math as mathematics
>>> mathematics.pi
3.141592653589793
```

Import from Function Definition

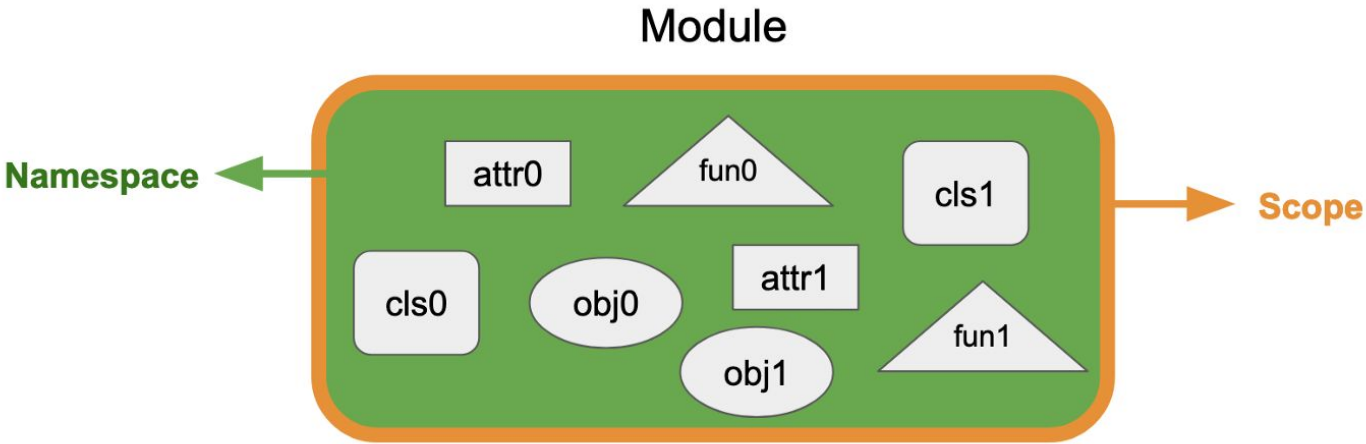
Digital Career Institute

Module contents can be imported from **within** a function definition. In that case, the import does not occur until the function is **called**:

```
>>> def foo():
...     from math import pi
...     print(pi * 2)
...
>>> foo()
6.283185307179586
```

Modules as a Namespace

In addition to being a module, `math` acts as a **namespace** that keeps all the attributes of the module together.



Other Way of Importing

Digital Career Institute

There are other ways to use it that allow you to import **specific parts** of a module:

```
>>> from math import pi
>>> math.pi
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'math' is not defined
>>> pi
3.141592653589793
```

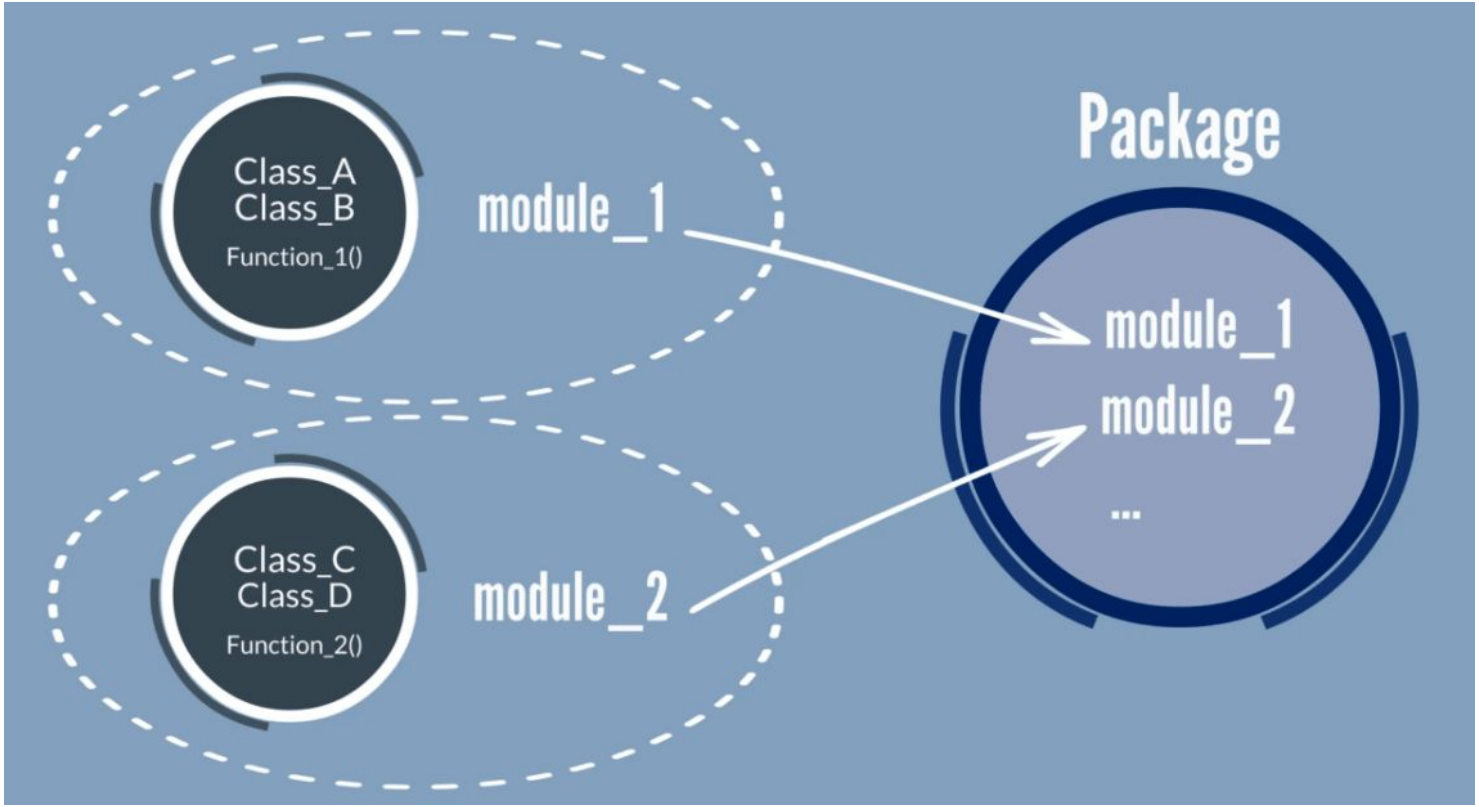
Packages

What is a Package?

Digital Career Institute

- The Python.org glossary defines **package** as follows:
    - “A Python module which can contain submodules or recursively, subpackages”.
- Technically, a package is a Python module with an `__path__` attribute.
- Note that a package is **still** a module. As a user, you usually don’t need to worry about whether you’re importing a module or a package.

What is a Package?



Package Definition

Digital Career Institute

- A **package** is a collection of related modules that **work together** to provide certain functionality.

- These modules are contained within a folder and can be imported just like any other modules.

- This folder will often contain a special `__init__.py` file that tells Python it's a package, potentially containing more modules nested within subfolders.

Imports inside Packages

Digital Career Institute

Absolute imports	
Pros	Cons
It is easy to tell exactly where the imported resource is	Can get quite verbose, depending on the complexity of the directory structure
PEP 8 explicitly recommends absolute imports	
Remain valid even if the current location of the import statement changes	

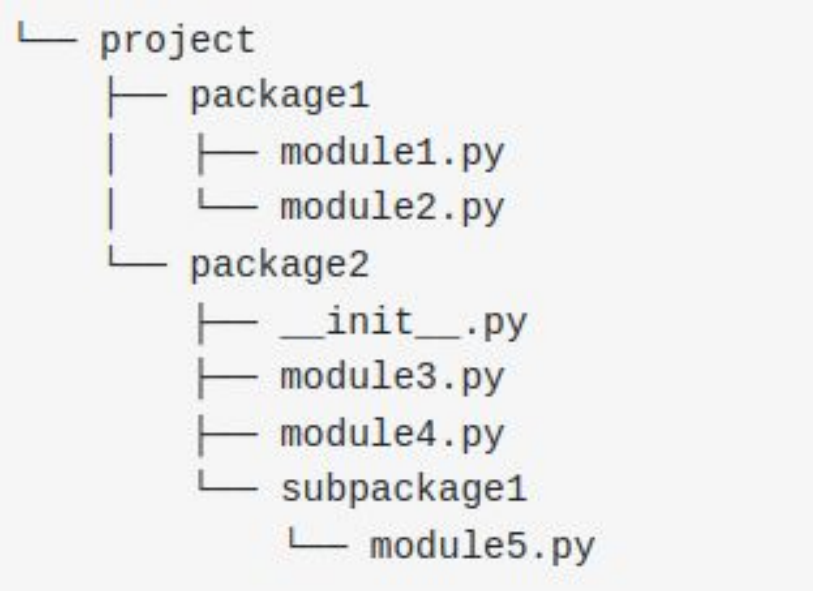
Imports inside Packages

Digital Career Institute

Relative imports	
Pros	Cons
They are quite brief	Can be messy, particularly for shared projects where directory structure is likely to change
	Not as readable as absolute ones
	Not easy to tell the location of the imported resources

What are Absolute Imports?

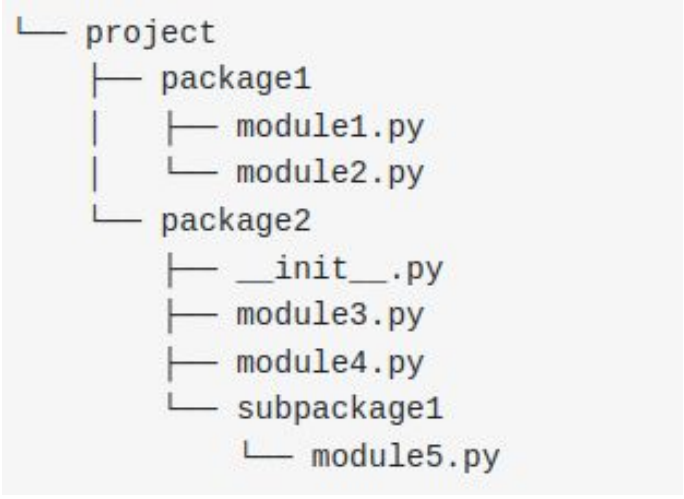
An **absolute** import specifies the resource to be imported using its **full path** from the project's root folder. Consider the structure below:



What are Absolute Imports?

- There's a directory, `project`, which contains two sub-directories, `package1` and `package2`.

- The `package1` directory has two files, `module1.py` and `module2.py`.



Absolute imports

- The `package2` directory has three files: **two modules**, `module3.py` and `module4.py`, and an **initialization file** `__init__.py`.

- It also contains a directory, **subpackage**, which in turn contains a file `module5.py`.

```
└── project
    ├── package1
    │   ├── module1.py
    │   └── module2.py
    └── package2
        ├── __init__.py
        ├── module3.py
        ├── module4.py
        └── subpackage1
            └── module5.py
```

Absolute Imports

Digital Career Institute

Let's assume the following:

- `package1/module2.py` contains a function, `function1`.
- `package2/__init__.py` contains a class, `class1`.
- `package2/subpackage1/module5.py` contains a function, **`function2`**.

Absolute Imports - Example

Digital Career Institute

The following are practical examples of absolute imports:

```
from package1 import module1

from package1.module2 import function1

from package2 import class1

from package2.subpackage1.module5 import function2
```

Pros and Cons of Absolute Imports

Digital Career Institute

- You must give a **detailed** path for each package or file, from the top-level package folder.
- This is somewhat similar to its file path, but we use a **dot** (.) instead of a **slash** (/).
- Absolute imports are **preferred** because they are quite clear and straightforward.
- PEP 8 explicitly **recommends** absolute imports.

Relative Imports

Digital Career Institute

- A **relative import** specifies the resource to be imported relative to the current location of the file (script) with code - that is, the location where the import statement is.

- The syntax of a **relative** import depends on the **current** location of the file (script) with code as well as the location of the module, package, or object to be imported.

Relative Imports

Digital Career Institute

- Here are a few examples of **relative** imports:

```
from .some_module import some_class

from ..some_package import some_function

from . import some_class
```

- You can see that there is **at least** one **dot** in each import statement above. Relative imports make use of **dot notation** to specify location.

Dot Notation

- A **single dot** means that the module or package referenced is in the same directory as the current location.
- **Two dots** mean that it is in the parent directory of the current location - that is, the directory above.
- **Three dots** mean that it is in the grandparent directory, and so on.
- This will probably be familiar to you if you use a **Unix-like** operating system!

Relative Imports

Let’s assume we have the same directory structure as before:

```
└── project
    ├── package1
    │   ├── module1.py
    │   └── module2.py
    └── package2
        ├── __init__.py
        ├── module3.py
        ├── module4.py
        └── subpackage1
            └── module5.py
```

Relative Imports

Digital Career Institute

Recall the file contents:

- `package1/module2.py` contains a function, `function1`.
- `package2/__init__.py` contains a class, `class1`.
- `package2/subpackage1/module5.py` contains a function, `function2`.

Relative Imports - Example

- You can import `function1` into the `package1/module1.py` file this way:

```
# package1/module1.py

from .module2 import function1
```

- You'd use only **one dot** here because `module2.py` is in the same directory as the current module, which is `module1.py`.

Relative Imports - Example #2

Digital Career Institute

- You can import `class1` and `function2` into the `package2/module3.py` file this way:

```
# package2/module3.py

from . import class1

from .subpackage1.module5 import function2
```

- In the first import statement, the **single dot** means that you are importing `class1` from the **current** package.

- Remember that importing a package essentially imports the package's `__init__.py` file as a module.

Imports Style Guide

Here are a few general rules of thumb for how to style your imports:

- Keep imports at the **top** of the file.
- Write imports on **separate** lines.
- Organize imports into groups:
    - first **standard library** imports.
    - then **third-party** imports.
    - and finally **local** application or **library** imports.

Imports Style Guide

Digital Career Institute

Here are a few general rules of thumb for how to style your imports:

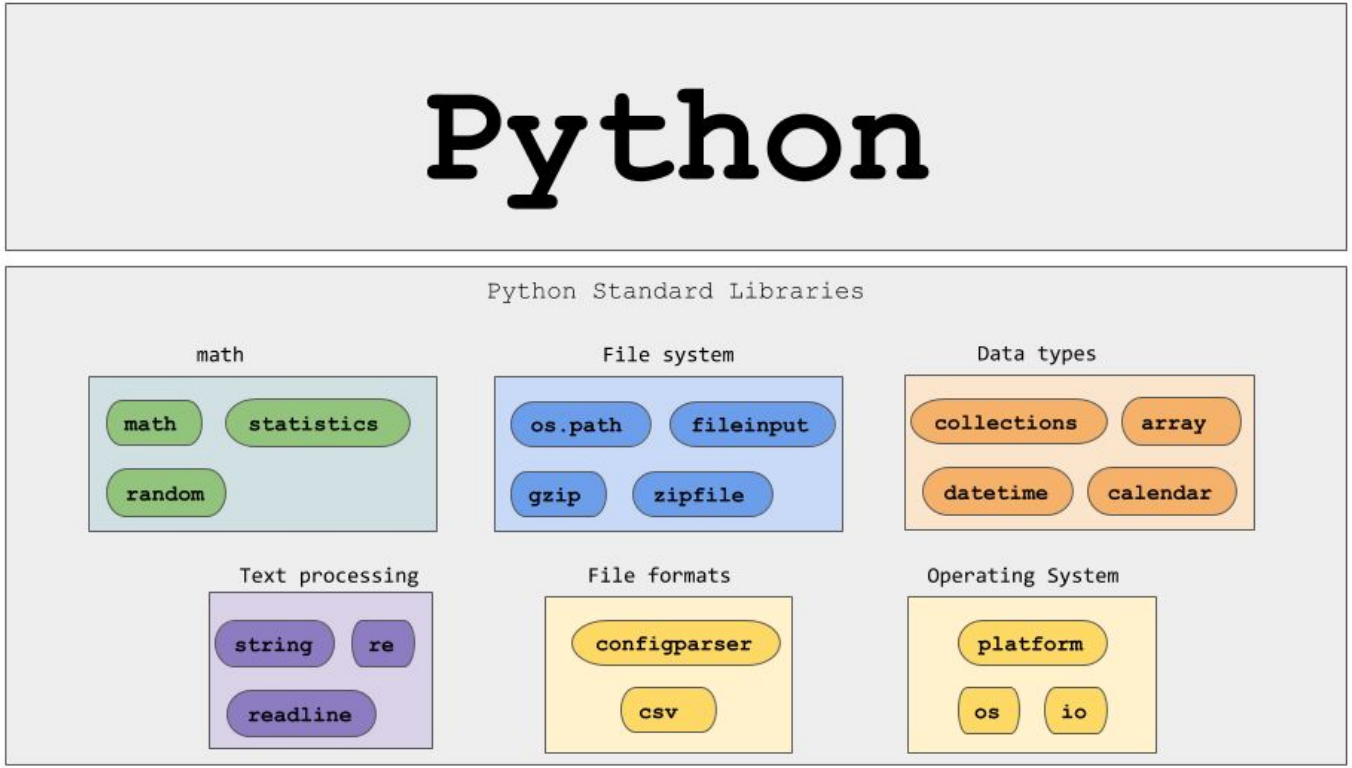
- Order imports **alphabetically** within each group.
- Prefer **absolute** imports over **relative** imports.
- Avoid **wildcard** imports like `from module import *`.

What is Library?

Digital Career Institute

- A **library** is an umbrella term that loosely means **“a bundle of code”.**
- These can have tens or even hundreds of individual modules that can provide a wide range of functionality.
- For example, Django is a backend library to create web applications.

Libraries in Python



pip - Package Manager for Python

Digital Career Institute

What is pip?

Digital Career Institute

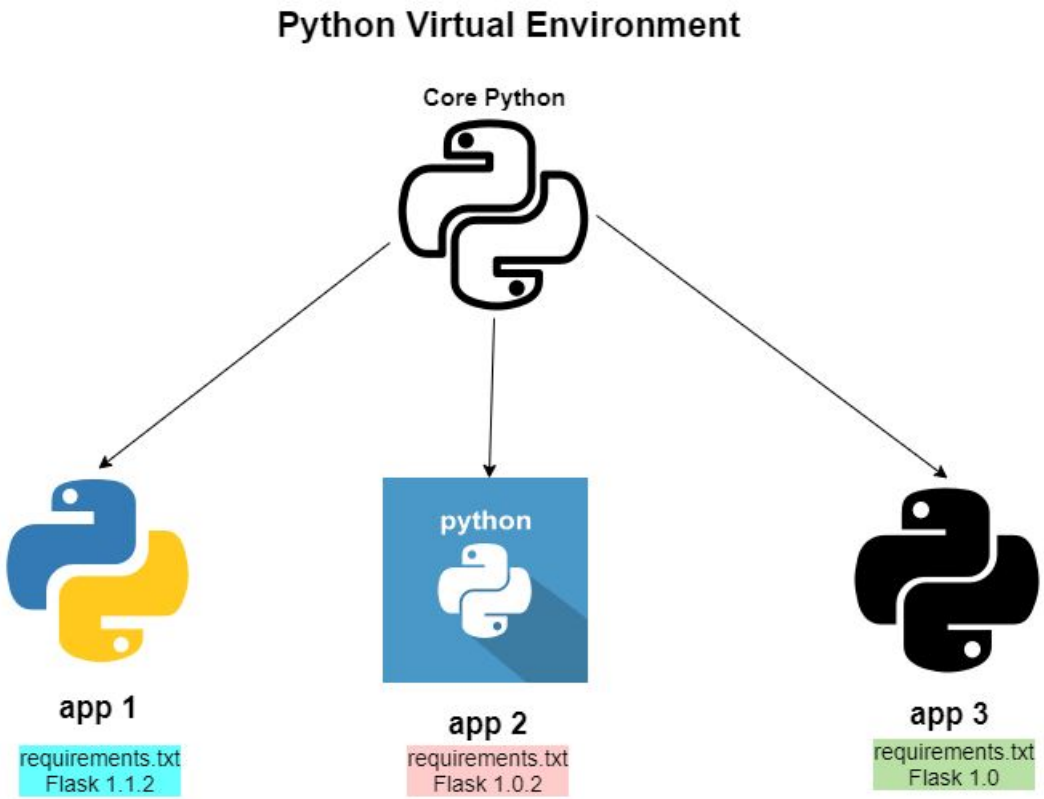
- **`pip`** is a package manager for Python.
- That means it's a tool that allows you to install and manage **additional** libraries and dependencies that are not distributed as part of the standard library.
- The Python installer installs pip, so it should be ready for you to use.

What is PyPI?

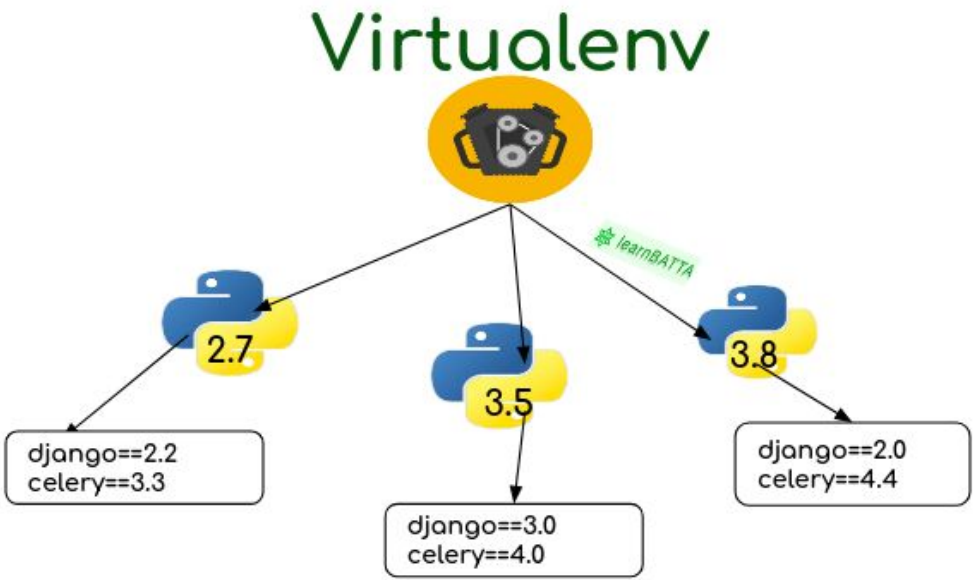
Digital Career Institute

- Python has a very active community that contributes an even **bigger** than Python Standard Library set of packages that can help you with your development needs.
- These packages are published to the Python Package Index, also known as **PyPI** (pronounced ***Pie Pea Eye***).
- PyPI hosts an **extensive** collection of packages that include development frameworks, tools, and libraries.

Virtual environments



Virtual Environment (virtualenv)



- Virtualenv provides dependency **isolation** for Python projects.
- Each project can have its own **dependencies**, regardless of what dependencies every other project has.

Creating Virtual Environments

Digital Career Institute

- To create a virtual environment, decide upon a directory where you want to place it, and run the venv module as a script with the directory path:

```
$ python3 -m venv tutorial-env
```

- This will create the ‘tutorial-env’ directory if it doesn’t exist, and also create directories inside it containing a copy of the Python interpreter, the standard library, and various supporting files.

Using Virtual Environments

Digital Career Institute

- Once you've created a virtual environment, you may **activate** it.
- Activating the virtual environment will change your shell's prompt to show what virtual environment you're using.
- It also modifies the environment so that running python will get you that particular version and installation of Python.
- Once you've stopped working with a virtual environment, you may **deactivate** it.

Using Virtual Environments - Example

Digital Career Institute

```
$ python3 -m venv test2-venv
$ source test2-venv/bin/activate
(test2-venv) $ pip install Django
Collecting Django
  Downloading Django-3.2.6-py3-none-any.whl (7.9 MB)
     |████████████████████████████████| 7.9 MB 1.4 MB/s
Successfully installed Django-3.2.6 asgiref-3.4.1 pytz-2021.1 sqlparse-0.4.1
(test2-venv)$
```

Installing Packages with pip

Digital Career Institute

- pip provides an `install` command to install packages.
- It's recommended to use it in **activated** virtual environments:

```
$ python3 -m venv test2-venv
$ source test2-venv/bin/activate
(test2-venv)$ pip install requests
Collecting requests
Installing collected packages: urllib3, idna, certifi, requests
Successfully installed certifi-2021.5.30 requests-2.26.0 ...
```

Using Virtual Environments - Full Example

Digital Career Institute

```
artur@artur-MSI:~/Desktop/DCI$ python3 -m venv tutorial-env
artur@artur-MSI:~/Desktop/DCI$ source tutorial-env/bin/activate
(tutorial-env) artur@artur-MSI:~/Desktop/DCI$ python
Python 3.8.5 (default, May 27 2021, 13:30:53)
[GCC 9.3.0] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> import sys
>>> sys.path
['', '/usr/lib/python38.zip', '/usr/lib/python3.8', '/usr/lib/python3.8/lib-dynl
oad', '/home/artur/Desktop/DCI/tutorial-env/lib/python3.8/site-packages']
>>> exit()
(tutorial-env) artur@artur-MSI:~/Desktop/DCI$ pip install Django
Collecting Django
  Downloading Django-3.2.4-py3-none-any.whl (7.9 MB)
     |                                | 7.9 MB 1.6 MB/s
Collecting asgiref<4,>=3.3.2
  Downloading asgiref-3.4.0-py3-none-any.whl (25 kB)
Collecting pytz
  Using cached pytz-2021.1-py2.py3-none-any.whl (510 kB)
Collecting sqlparse>=0.2.2
  Using cached sqlparse-0.4.1-py3-none-any.whl (42 kB)
Installing collected packages: asgiref, pytz, sqlparse, Django
Successfully installed Django-3.2.4 asgiref-3.4.0 pytz-2021.1 sqlparse-0.4.1
(tutorial-env) artur@artur-MSI:~/Desktop/DCI$
```

Review of pip Commands

Digital Career Institute

You can learn about pip supported **commands** by running it with help:

```
$ pip help

Usage:

  pip <command> [options]

Commands:

  install                     Install packages.

  download                    Download packages.

  uninstall                   Uninstall packages....
```

Install Packages with pip

Digital Career Institute

- You use pip with an install command followed by the **name** of the package you want to install.

- The pip install <package> command always looks in PyPI for the **latest** version of the package and installs it. If you want to install **specific** version use '==' (for example pip install requests**==**2.25.0)

- It also searches for dependencies listed in the package metadata and installs those dependencies to insure that the package has all the requirements it needs.

Check Installed Packages

Digital Career Institute

As you can see, multiple packages were installed. You can look at the package **metadata** by using the show command in pip:

```
(test2-venv) [artur@artur-PC DCI]$ pip show requests
Name: requests
Version: 2.26.0
Summary: Python HTTP for Humans.
Home-page: https://requests.readthedocs.io
Author: Kenneth Reitz
License: Apache 2.0
Location: /home/artur/Desktop/DCI/test2-venv/lib/python3.9/site-packages
Requires: certifi, charset-normalizer, idna, urllib3
Required-by:
```

Check List of Installed Packages

Use the `list` command to see the packages installed in your environment:

```
(test2-venv) $ pip list
Package            Version
------------------ ---------
certifi            2021.5.30
Django             3.2.6
idna               3.2
pip                21.2.4
pytz               2021.1
requests           2.26.0
urllib3            1.26.6
```

Requirements File

- You want to create a specification of the **dependencies** and versions you used to **develop** and **test** your application, so there are no surprises when you use the application in **production**.

- **Requirements** files allow you to specify exactly which packages and versions should be installed. Running `pip help` shows that there is a `freeze` command that outputs the installed packages in requirements format.

Usage of `freeze` Command

Digital Career Institute

```
(test2-venv) $ pip freeze
asgiref==3.4.1
certifi==2021.5.30
charset-normalizer==2.0.4
Django==3.2.6
idna==3.2
pytz==2021.1
requests==2.26.0
sqlparse==0.4.1
urllib3==1.26.6
```

Redirect Results of `freeze` Command

Digital Career Institute

You can use command `freeze`, redirecting the output to a file to generate a **requirements** file:

```
(test2-venv) $ pip freeze > requirements.txt
(test2-venv) $ cat requirements.txt
asgiref==3.4.1
certifi==2021.5.30
Django==3.2.6
idna==3.2
pytz==2021.1
requests==2.26.0
sqlparse==0.4.1
urllib3==1.26.6
```

Installing Packages Using Requirements

Digital Career Institute

If you have **requirements** file and you want to create new virtualenv with dependencies from this file, then you can use **install** command with the **-r** flag:

```
(test2-venv)$ deactivate
$ python3 -m venv new-test-env
$ source new-test-env/bin/activate
(new-test-env) [artur@artur-PC DCI]$ pip install -r requirements.txt
Collecting Django==3.2.6
...
Installing collected packages: urllib3, certifi, asgiref, requests, Django
Successfully installed Django-3.2.6 ... requests-2.26.0 ... urllib3-1.26.6
```

At the core of the lesson

Digital Career Institute

Lesson learned:

- We know the idea of imports in Python.
- We know the difference between relative and absolute imports.
- We know the difference between script, module and package.
- We know the usage of the `pip` package manager

General Thoughts about Libraries

Software Types

Custom Developed Software	**Third Party Software**
Software that is specifically developed for an organization or other user.	A reusable software component developed to be either freely distributed or sold by an entity **other** than the original vendor of the operating system. Sometimes known as commercial-off-the-shelf or “COTS” software.

Levels of Parties

- The **operating system** defines the **first** party (for example *Windows or Mac*).
- The **second** party is you (the user).
- The **third** party is the vendor who created the software.

Real World Example

Here's how it works if we're talking for example about mobile apps:

- For both iOS or Android, many apps available were not built by Apple or Google, but instead by **third-parties** including software development studios or solo developers.
- You, the **second** party, download the **third** party onto the **first** party, in this case your phone.

Potential Pitfalls of 3rd Party Software

Digital Career Institute

#	Pitfall
1	Staff bloat
2	The terrible third party web from hell
3	Integration, Continuous Integration & Discontinuation

Potential Pitfalls in Custom Development

Digital Career Institute

#	Pitfall
1	Legacy code
2	Cost vs. Quality
3	Timelines and deadlines

Takeaway

Digital Career Institute

Understand Cost/Risk

- A quick Google search returns countless blog posts and forums covering build versus buy, but they all have one recommendation in common: ***Clearly identify the problem!***
- The better you understand, in extreme detail, the problem being solved for your business, the easier it is to choose from a sea of solutions.

Update Strategies

Debian style (freeze update)	Arch Linux style (always update)
Only stable updates	Newest is always the best
Stability is up-to-date	Users live with compatibility and stability issues

Versioning Your Package

Digital Career Institute

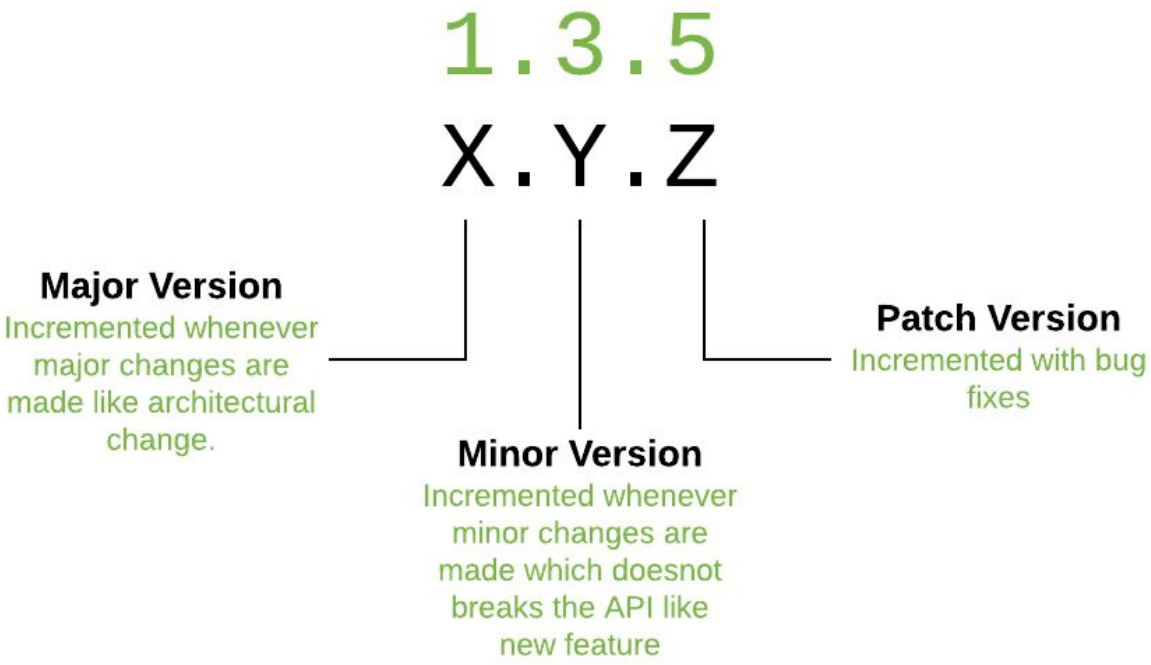
- Your package needs to have a version, and PyPI will only let you do one upload of a particular version for a package.
- In other words, if you want to update your package on PyPI, you need to **increase** the version number first.
- This is a good thing, as it guarantees **reproducibility**: two systems with the same version of a package should behave the same.

Versioning Your Package

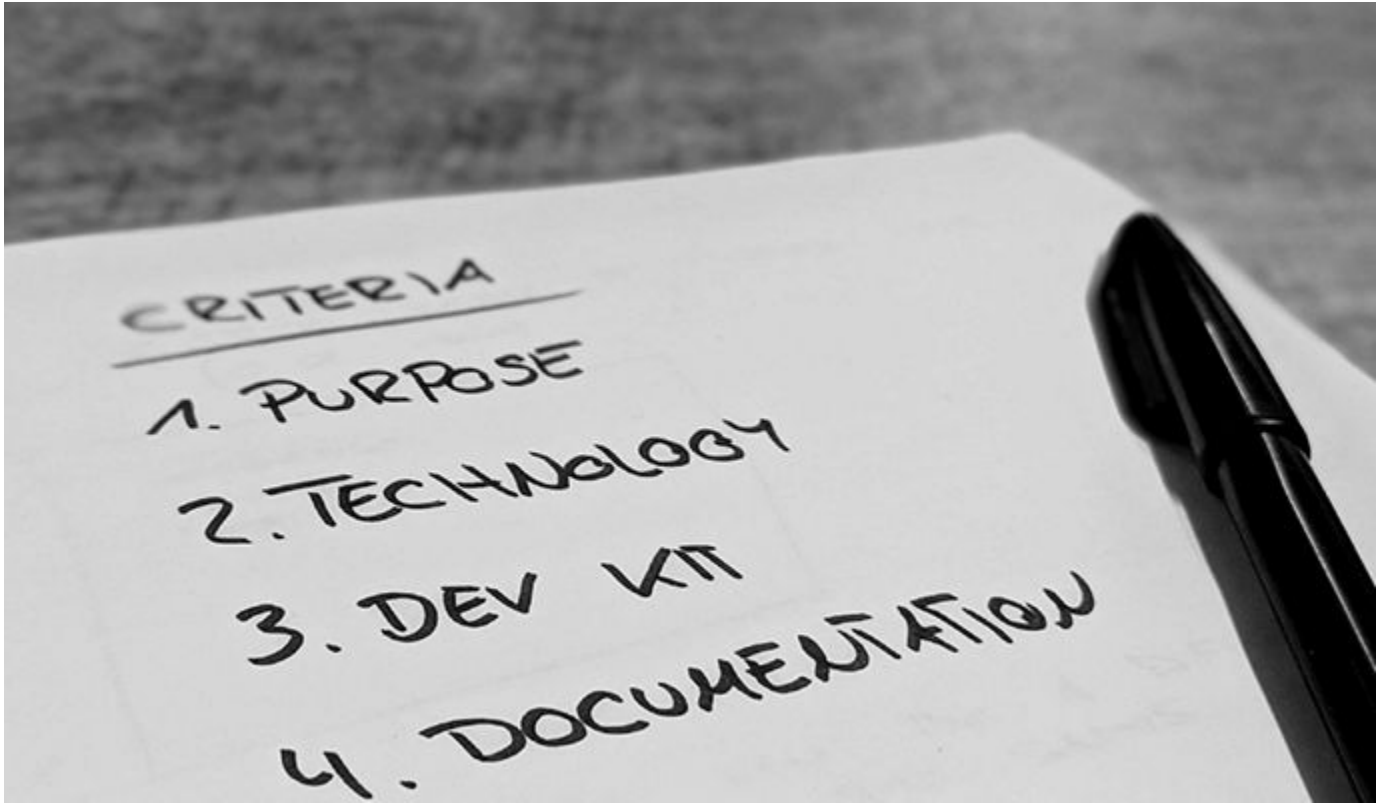
Digital Career Institute

- Semantic versioning is a good default scheme to use.
- The version number is given as three numerical components, for instance 0.1.2.
- The components are called **MAJOR**, **MINOR**, and **PATCH**, and there are simple rules about when to increment each component.

Versioning Your Package



Assessing External Library



Assessing External Library

Digital Career Institute

#	Criteria
1	Purpose
2	Technology
3	Development Kit
4	Documentation
5	Cost
6	Releases
7	Size

Assessing External Library

Digital Career Institute

#	Criteria
8	Community support
9	Professional Support
10	License
11	Performance
12	Browser Compatibility
13	Device Compatibility
14	Data Format

Assessing External Library

Digital Career Institute

#	Criteria
15	Look & Feel
16	Extendibility
17	Future Direction

Flask Library

- Flask is a popular Python **web framework** that allows developers to **quickly build web applications.**

- It is a **lightweight** and **flexible** framework.

- It is ideal for small to medium-sized projects, but also larger projects can be accomplished.

Flask Library

- It is often used to build **RESTful APIs**, **web services**, and other types of web applications.

- Flask is also highly **customizable**, and allows developers to easily add **extensions** and **plugins** to extend its functionality.

Flask Example

app.py file

```
from flask import Flask

app = Flask(__name__)

@app.route('/')

def home():

    return 'Hello, World!'

if __name__ == '__main__':

    app.run(debug=True)
```


- **Run** this code and see the output **Hello, World!** in a **browser** by going to this **url**:

  http://127.0.0.1:5000

Flask Example

```
from flask import Flask

app = Flask(__name__)
```

Here, we import the **Flask class** and create an instance of it called **app**.

```
@app.route('/')
```

We then define a **route** using the **@app.route()** decorator.

```
def home():

    return 'Hello, World!'
```

The route is mapped to the root URL **'/'**, and the **home()** function is called when the route is accessed.

Flask Example

```
def home():
    return 'Hello, World!'
```

The **home()** function simply returns the string **'Hello, World!'** which will be displayed in the user's browser when they access the root URL.

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

Finally, we start the application using the **app.run()** method.

The **debug=True** parameter enables **debugging mode** which makes it easier to **develop** and **troubleshoot** the application.

At the core of the lesson

Lesson learned:

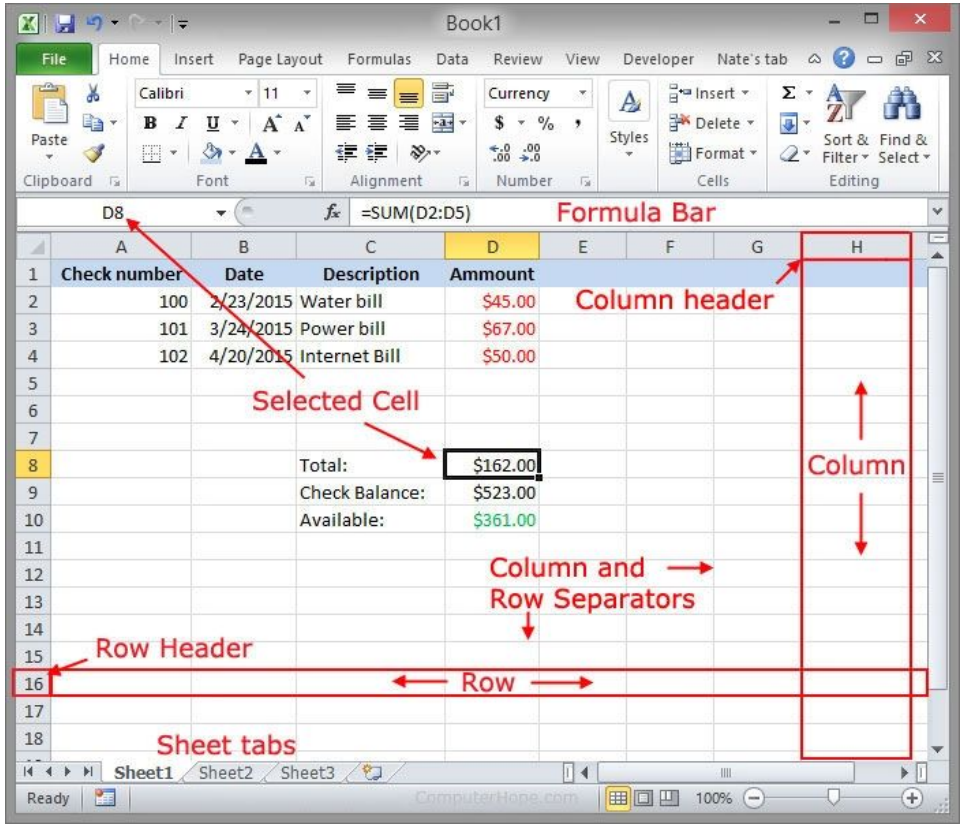
- We know the difference between custom developed software and third party software
- We know pros and cons of both approaches
- We know how to assess given library

Work with Excel files

Digital Career Institute

Spreadsheets

Spreadsheets are a very intuitive and user-friendly way to manipulate large datasets without any prior technical background. That's why they're still so commonly used today.



Basic Terminology

Term	Explanation
Spreadsheet or Workbook	A **Spreadsheet** is the main file you are creating or working with.
Worksheet or Sheet	A **Sheet** is used to split different kinds of content within the same spreadsheet. A **Spreadsheet** can have one or more **Sheets**.
Column	A **Column** is a vertical line, and it's represented by an uppercase letter: *A*.

Basic Terminology

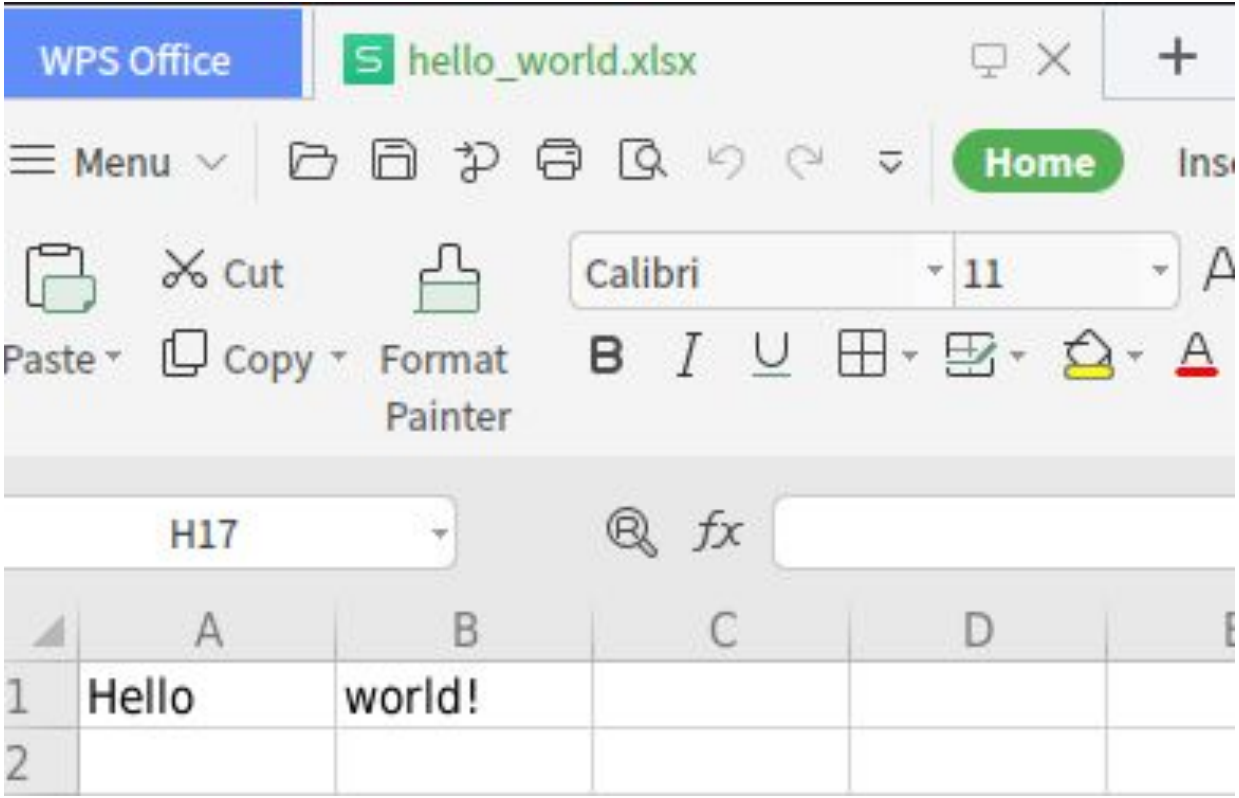
Term	Explanation
Row	A **Row** is a horizontal line, and it's represented by a number: *1*.
Cell	A **Cell** is a combination of **Column** and **Row**, represented by both an uppercase letter and a number: *A1*.

‘Hello world’ in openpyxl

Digital Career Institute

```
from openpyxl import Workbook

# name of the file to create
name = "hello_world.xlsx"

# workbook is always created with at least one worksheet
wb = Workbook()

# You can get it by using the Workbook.active property
sheet = wb.active

sheet["A1"] = "Hello"   # add data to specific cells
sheet["B1"] = "world!"
# save the spreadsheet when you’re done
wb.save(filename=name)
```

‘Hello world’ in openpyxl - Result



Creating Sheets

- You can create new worksheets using the `create_sheet()` method.

- Sheets are given a name automatically when they are created. They are numbered in sequence (`Sheet`, `Sheet1`, `Sheet2`, ...).

- You can change this name at any time with the `Worksheet.title` property

Creating Sheets

```
ws1 = wb.create_sheet("Mysheet")    # insert at the end (default)
ws2 = wb.create_sheet("Sheet", 0)   # insert at first position

ws2.title = "DCI sheet"             # rename

wb.save(filename=name)              # save the spreadsheet
```

Reading Spreadsheets

- Sample to this lecture is available on DCI repo. You can just download it and try some exercises!

	A	B	C	D	E	
1	marketplace	customer_id	review_id	product_id	product_parent	product_title
2	US	3653882	R3O9SGZBVQBV76	B00FALQ1Z	937001370	Invicta Wome
3	US	14661224	RKH8BNC3L5DLF	B00D3RGO	484010722	Kenneth Cole
4	US	27324930	R2HLE8WKZSU3NL	B00DKYC7T	361166390	Ritche 22mm
5	US	7211452	R31U3UH5AZ42LL	B000EQS1J	958035625	Citizen Men's
6	US	12733322	R2SV659OUJ945Y	B00A6GFD7	765328221	Orient ER270(
7	US	6576411	RA51CP8TR5A2L	B00EYSOSE	230493695	Casio Men's G
8	US	11811565	RB2Q7DLDN6TH6	B00WM0QA	549298279	Fossil Women
9	US	49401598	R2RHFJV0UYBK3Y	B00A4EYBF	844009113	INFANTRY Mer
10	US	45925069	R2Z6JOQ94LFHEP	B00MAMPG	263720892	G-Shock Men's

Reading Spreadsheets

Digital Career Institute

```
from openpyxl import load_workbook

workbook = load_workbook(filename="reviews-sample.xlsx")
print(workbook.sheetnames)
sheet = workbook.active
print(sheet["A1"])           # the Cell object
print(sheet["A1"].value)     # the actual value of a cell

# the method .cell() uses index notation to retrieve a cell
print(sheet.cell(row=2, column=3).value)
```

Reading Spreadsheets

Digital Career Institute

Results of code from **previous** slide:

```
['amazon_reviews_us_Watches_v1_00-sample']

<Cell 'amazon_reviews_us_Watches_v1_00-sample'.A1>

marketplace

R3O9SGZBVQBV76
```

Reading Spreadsheets

Digital Career Institute

```
for index in range(1, sheet.max_column + 1):
    cell_obj = sheet.cell(row = 1, column = index)
    # print all columns name
    print(cell_obj.value)
sheet.title = 'reviews'
print(sheet["A1:C2"])


# results

marketplace
customer_id
review_id
product_id
product_parent
product_title
product_category
```

We have 15 columns' names in this sample,
First 7 are shown.

Additional Reading Options

- There are a few arguments you can pass to `load_workbook()` that change the way a spreadsheet is loaded. The most important ones are the following two Booleans:
    - `read_only` loads a spreadsheet in read-only mode allowing you to open very large Excel files.
    - `data_only` ignores loading formulas and instead loads only the resulting values.

Accessing Many Cells

- There are a few different ways you can iterate through the data depending on your needs.
- You can slice the data with a combination of columns and rows:

```
sheet.title = 'reviews'
print(sheet["A1:C2"])

# results
((<Cell 'reviews'.A1>, <Cell 'reviews'.B1>, <Cell 'reviews'.C1>), (<Cell
'reviews'.A2>, <Cell 'reviews'.B2>, <Cell 'reviews'.C2>))
```

Accessing Many Cells

Digital Career Institute

- You can get ranges of rows or columns:

```
# Get all cells from column A
>>> sheet["A"]


>>> # Get all cells for a range of columns
>>> sheet["A:B"]
```

Accessing Many Cells

Digital Career Institute

```
>>> # Get all cells from row 5
>>> sheet[5]

>>> # Get all cells for a range of rows
>>> sheet[5:6]
```

You'll notice that all of the above examples return a single **tuple**.

Accessing Many Cells

Digital Career Institute

There are also multiple ways of using normal Python generators to go through the data. The main methods you can use to achieve this are:

- `iter_rows()`
- `iter_cols()`

Accessing Many Cells

Digital Career Institute

- Both methods can receive the following arguments:
    - `min_row`
    - `max_row`
    - `min_col`
    - `max_col`

- These arguments are used to set boundaries for the iteration!

Accessing Many Cells

- You get **one** tuple element **per row** selected:

```
for row in sheet.iter_rows(min_row=1, max_row=2, min_col=1, max_col=3):
    print(row)


# results
(<Cell 'reviews'.A1>, <Cell 'reviews'.B1>, <Cell 'reviews'.C1>)
(<Cell 'reviews'.A2>, <Cell 'reviews'.B2>, <Cell 'reviews'.C2>)
```

- While when using .**iter_cols()** and iterating through **columns**, you'll get one tuple per column instead.

Accessing Many Cells

- One additional argument you can pass to both methods is the Boolean `values_only`. When it's set to **`True`**, the values of the cell are returned, instead of the `Cell` object:

```
for row in sheet.iter_rows(min_row=1, max_row=2, min_col=1, max_col=3,
values_only=True):
    print(row)


# results
('marketplace', 'customer_id', 'review_id')
('US', 3653882, 'R3O9SGZBVQBV76')
```

Managing Rows and Columns

- One of the most common things you have to do when manipulating spreadsheets is adding or removing rows and columns.

- The openpyxl package allows you to do that in a very straightforward way by using the methods:

  - `.insert_rows()`
  - `.delete_rows()`
  - `.insert_cols()`
  - `.delete_cols()`

Managing Rows and Columns

Digital Career Institute

- Every single one of those methods can receive two arguments:
  - `idx`
  - `amount`

```
>>> # Insert a column before the existing column 1 ("A")

>>> sheet.insert_cols(idx=1)
```

Managing Rows and Columns

Digital Career Institute

```
# Insert 5 columns between column 2 ("B") and 3 ("C")

>>> sheet.insert_cols(idx=3, amount=5)


# Insert a new row in the beginning

>>> sheet.insert_rows(idx=1)
```

Managing Rows and Columns

Digital Career Institute

```
# Delete the created columns

>>> sheet.delete_cols(idx=3, amount=5)

>>> sheet.delete_cols(idx=1)
```

The only thing you need to remember is that when inserting new data (rows or columns), the insertion happens **before** the `idx` parameter.

Managing Rows and Columns

- So, if you do `insert_rows(1)`, it inserts a new row **before** the existing first row.

- It's the same for columns: when you call `insert_cols(2)`, it inserts a new column right **before** the already existing second column (B).

Managing Rows and Columns

Digital Career Institute

- However, when deleting rows or columns, `.delete_...` deletes data **starting from** the index passed as an argument.

- For example, when doing `delete_rows(2)` it deletes row 2, and when doing `delete_cols(3)` it deletes the third column (C).

Managing Sheets

Digital Career Institute

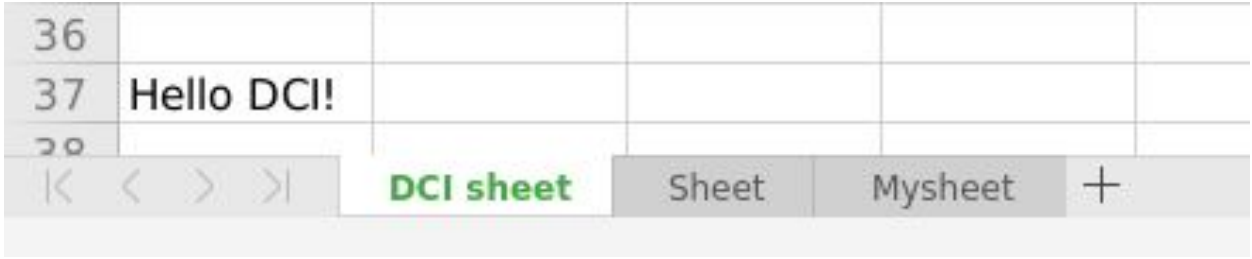
- Sheet management is also one of those things you might need to know, even though it might be something that you don't use that often.

- If you look back at the code examples from this tutorial, you'll notice the following recurring piece of code:

```
sheet = workbook.active
```

Setting Active Worksheet

```
# set "DCI sheet" as active worksheet
ws2 = wb.active
ws2["A37"] = "Hello DCI!"
# save the spreadsheet when you're done
wb.save(filename=name)
```



Managing Sheets

- If you want to **create** or **delete** sheets, then you can also do that with `.create_sheet()` and `.remove()`:

```
operations_sheet = workbook.create_sheet("Operations")
```

- You can also define the position to create the sheet at:

```
hr_sheet = workbook.create_sheet("HR", 0)
```

Managing Sheets

- To remove them, just pass the sheet as an argument to the .remove()

```
workbook.remove(operations_sheet)
```

- One other thing you can do is make duplicates of a sheet using `copy_worksheet()`:

```
>>>products_sheet = workbook["Products"]

>>> workbook.copy_worksheet(products_sheet)

<Worksheet "Products Copy">
```

Freezing Rows and Columns

Digital Career Institute

- Something that you might want to do when working with big spreadsheets is to **freeze** a few rows or columns, so they remain visible when you scroll right or down.

- **Freezing** data allows you to keep an eye on important rows or columns, regardless of where you scroll in the spreadsheet.

Freezing Rows and Columns

Digital Career Institute

```
>>> workbook = load_workbook(filename="sample.xlsx")

>>> sheet = workbook.active

>>> sheet.freeze_panes = "C2"

>>> workbook.save("sample_frozen.xlsx")
```

Freezing Rows and Columns

- If you open the `sample_frozen.xlsx` spreadsheet in your favorite spreadsheet editor, you'll notice that row 1 and columns A and B (C2 was the argument of frooze_panes) are **frozen** and are always **visible** no matter where you navigate within the spreadsheet.

- This feature is handy, for example, to keep headers within sight, so you always know what each column represents.

At the core of the lesson

Lesson learned:

- We know the idea of spreadsheets.
- We know how to work with `openpyxl` package.

Packaging

What is a Package?

Digital Career Institute

- The Python.org glossary defines **package** as follows:
    - “A Python module which can contain submodules or recursively, subpackages”
- Technically, a package is a Python module with an `__path__` attribute.
- Note that a package is **still** a module. As a user, you usually don’t need to worry about whether you’re importing a module or a package.

Package Definition

Digital Career Institute

- A **package** is a collection of related modules that **work together** to provide certain functionality.

- These modules are contained within a folder and can be imported just like any other modules.

- This folder will often contain a special `__init__.py` file that tells Python it's a package, potentially containing more modules nested within subfolders.

Package in Practice

Digital Career Institute

- In **practice**, a package typically corresponds to a file **directory** containing Python files and other directories.
- To create a Python package yourself, you create a directory and a file named `__init__.py` inside it.
- The `__init__.py` file contains the contents of the package when it's treated as a module. It can be left **empty**.

Organizing Code into Packages

Digital Career Institute

- Suppose you have developed a very large application that includes many **modules**.
- As the number of modules **grows**, it becomes difficult to keep track of them all if they are dumped into one location.
- This is particularly so if they have similar names or functionality.
- You might wish for a means of **grouping** and **organizing** them.

Organizing Code into Packages

Digital Career Institute

- **Packages** allow for a hierarchical structuring of the module namespace using **dot notation**.

- In the same way that **modules** help avoid collisions between global variable names, **packages** help avoid collisions between module names.

Creating a Package Locally

- Consider the following arrangement:

Creating a Package Locally

Digital Career Institute

The contents of the modules are:

```
# mod1.py

def foo():

    print('[mod1] foo()')


class Foo:

    pass
```

```
# mod2.py

def bar():

    print('[mod2] bar()')


class Bar:

    pass
```

Importing Module From a Package

Digital Career Institute

You can refer to these two **modules** with **dot notation**:

```
>>> import pkg.mod1, pkg.mod2

>>> pkg.mod1.foo()
[mod1] foo()
>>> x = pkg.mod2.Bar()
>>> x
<pkg.mod2.Bar object at 0x7fe519f37f40>
```

Importing Module From a Package

Digital Career Institute

You can also make an import in a different way:

```
>>> from pkg.mod1 import foo
>>> foo()
[mod1] foo()

>>> from pkg.mod2 import Bar as Qux
>>> x = Qux()
>>> x
<pkg.mod2.Bar object at 0x7fe519e80610>
```

Importing Package

Digital Career Institute

- You can technically import the package as well.
- But this is of little avail. Though this is, strictly speaking, a syntactically correct Python statement, it doesn't do much of **anything** **useful**.
- In particular, it **does not place** any of the modules in pkg into the local namespace:

Importing Package

```
>>> import pkg
Invoking __init__.py for pkg
>>> pkg.mod1
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: module 'pkg' has no attribute 'mod1'
>>> pkg.mod2.Bar()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: module 'pkg' has no attribute 'mod2'
```

Package Initialization

- If a file named **`__init__.py`** is present in a package directory, it is invoked when the package or a module in the package is imported.

- This can be used for execution of package initialization code, such as initialization of package-level data, for example:

```
# __init__.py

print(f'Invoking __init__.py for {__name__}')

A = ['Hello', 'DCI', 'students']
```

Package Initialization

Let’s add this file to the **pkg** directory from the above example:

Package Initialization

Digital Career Institute

Now when the package is imported, the global list A is initialized:

```
>>> import pkg

>>> pkg.A

['Hello', 'DCI', 'students']
```

Package Initialization

- **`__init__.py`** can also be used to effect **automatic** importing of modules from a package.
- Earlier we saw that the statement **`import pkg`** only places the name pkg in the caller's local symbol table and doesn't import any modules.
- **But** if **`__init__.py`** in the **`pkg`** directory contains the following:

```
# __init__.py
print(f'Invoking __init__.py for {__name__}')
import pkg.mod1, pkg.mod2
```

Package Initialization

Then when you execute import `pkg`, modules `mod1` and `mod2` are imported automatically:

```
>>> import pkg

Invoking __init__.py for pkg

>>> pkg.mod1.foo()

[mod1] foo()

>>> pkg.mod2.bar()

[mod2] bar()
```

Regular Package

- Python defines two types of packages, **regular** packages and **namespace** packages.
- Regular packages are **traditional** packages as they existed in Python 3.2 and earlier.
- A regular package is typically implemented as a directory containing an `__init__.py` file.

Regular Package

- When a regular package is imported, this `__init__.py` file is implicitly **executed**, and the objects it defines are bound to names in the package's namespace.

- The `__init__.py` file can contain the same Python code that any other module can contain, and Python will add some additional attributes to the module when it is imported.

Regular Package

```
parent/
    __init__.py
    one/
        __init__.py
    two/
        __init__.py
    three/
        __init__.py
```

- Importing `parent.one` will implicitly execute `parent/__init__.py` and `parent/one/__init__.py`.

- Subsequent imports of `parent.two` or `parent.three` will execute `parent/two/__init__.py` and `parent/three/__init__.py` respectively.

Namespace Package

Digital Career Institute

- As of Python 3.3, we get **namespace** packages.
- These are a special kind of package that allows you to unify two packages with **the same name** at different points on your Python-path.
- For example, on the next slide consider `path1` and `path2` as separate entries on your Python-path

Namespace Package - Example

```
$ tree -L 3 -I "__pycache__"
.
├── path1
│   └── namespace
│       ├── module1.py
│       └── module2.py
├── path2
│   └── namespace
│       ├── module3.py
│       └── module4.py
└── path-script.py
```

Namespace Package

Digital Career Institute

- To add `path1` and `path2` to `PYTHONPATH` we could type following commands in Unix-like systems:

```
$ export PYTHONPATH="${PYTHONPATH}:/home/.../NAMESPACE_PACKAGE/path1"

$ export PYTHONPATH="${PYTHONPATH}:/home/.../NAMESPACE_PACKAGE/path2"
```

- And now we can get the **unification** of two packages with the same name in a single namespace:

```
# in /home/.../NAMESPACE_PACKAGE
>>> from namespace import module1, module2
```

Namespace Package

- If either one of them gain an `__init__.py` that becomes the regular package - and you no longer get the unification as the other directory is ignored.
- If both of them have an `__init__.py`, the **first one** in the `PYTHONPATH` (`sys.path`) is the one used.
- `__init__.py` used to be required to make directory a package
- Namespace packages are packages **without** the `__init__.py`.

At the core of the lesson

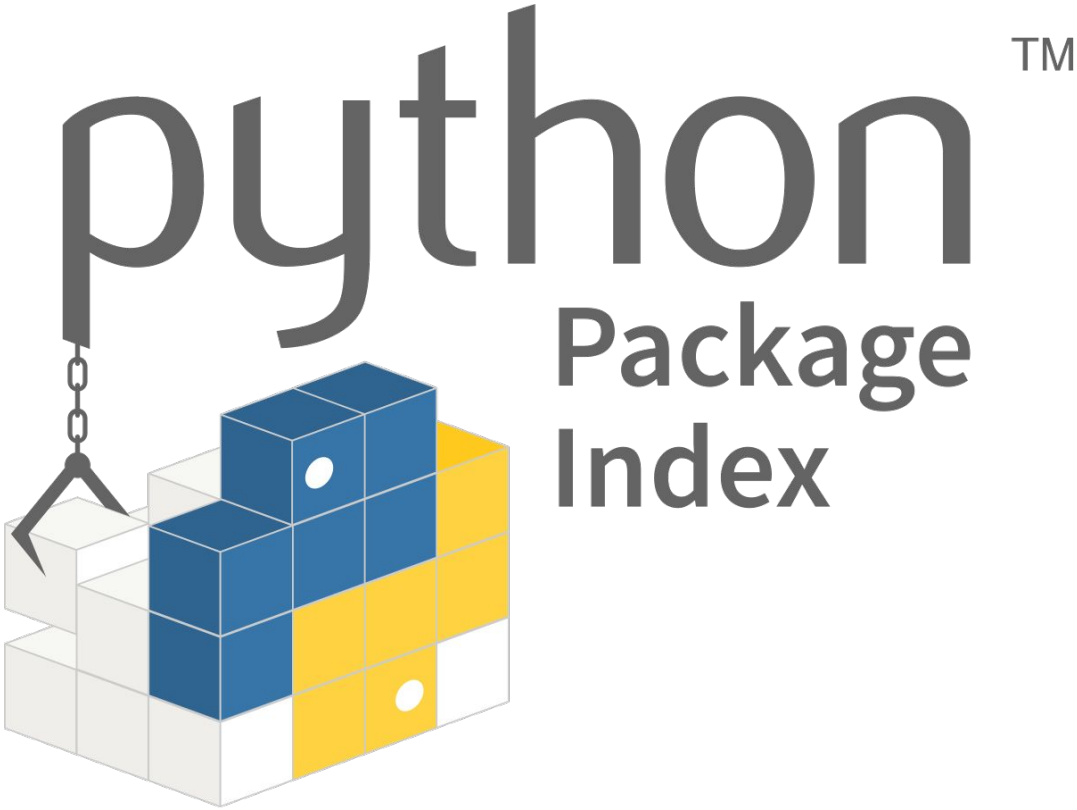
Lesson learned:

- We know the idea of packages in Python.
- We know how to organize our code into packages.
- We know the difference between regular package and namespace package.

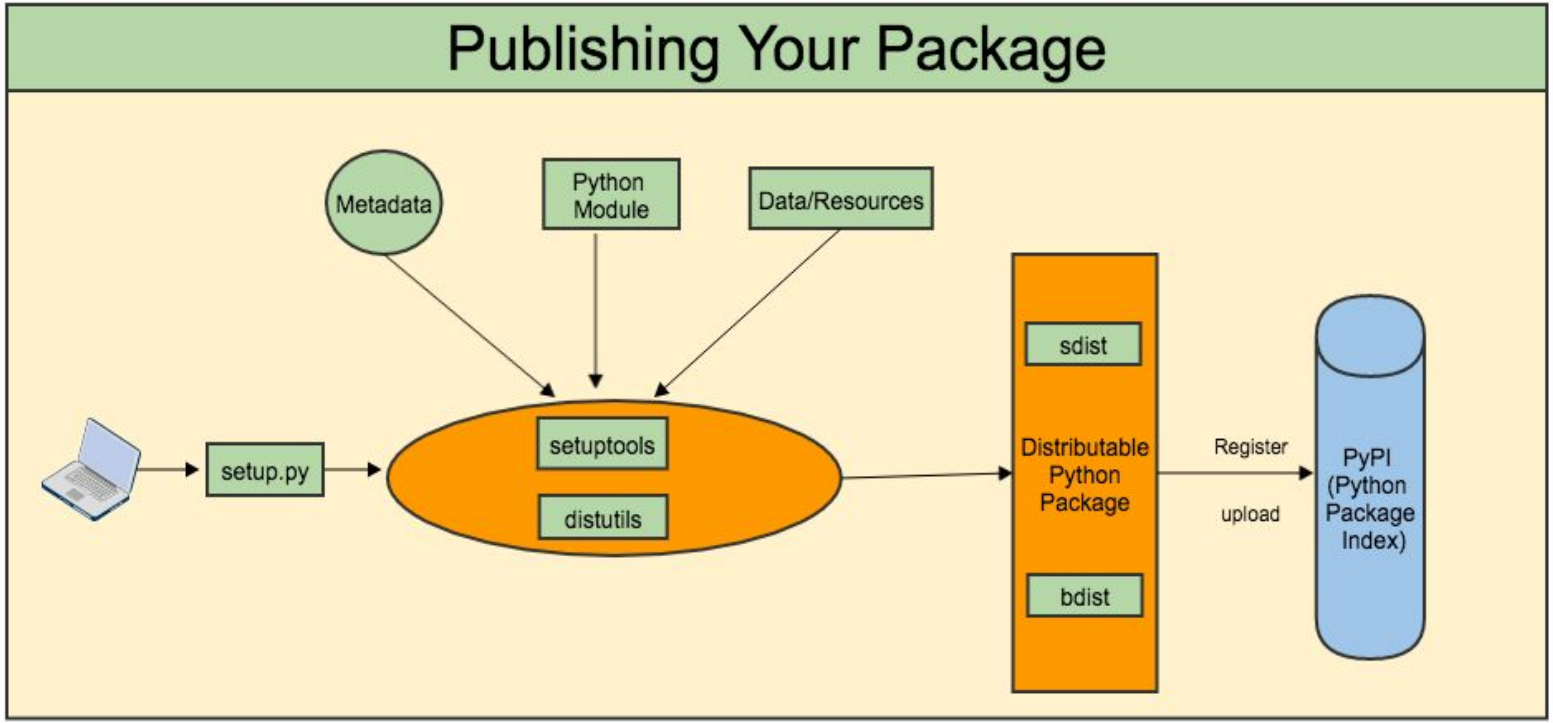
Creating a Package (on PyPI)

PyPi and Package Index

Digital Career Institute



PyPi and Package Index



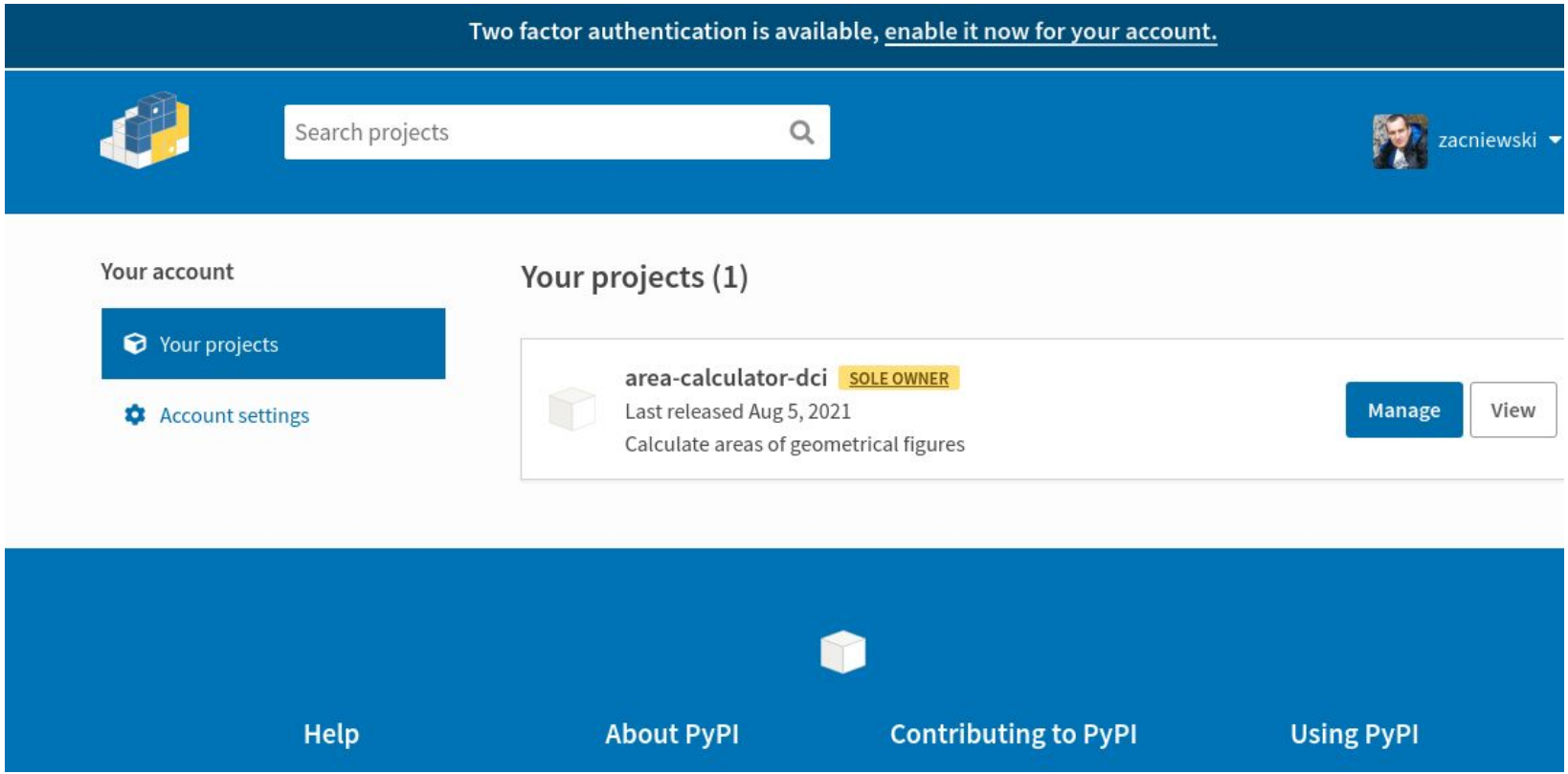
Package Index Instances

There are two instances of the Package Index:

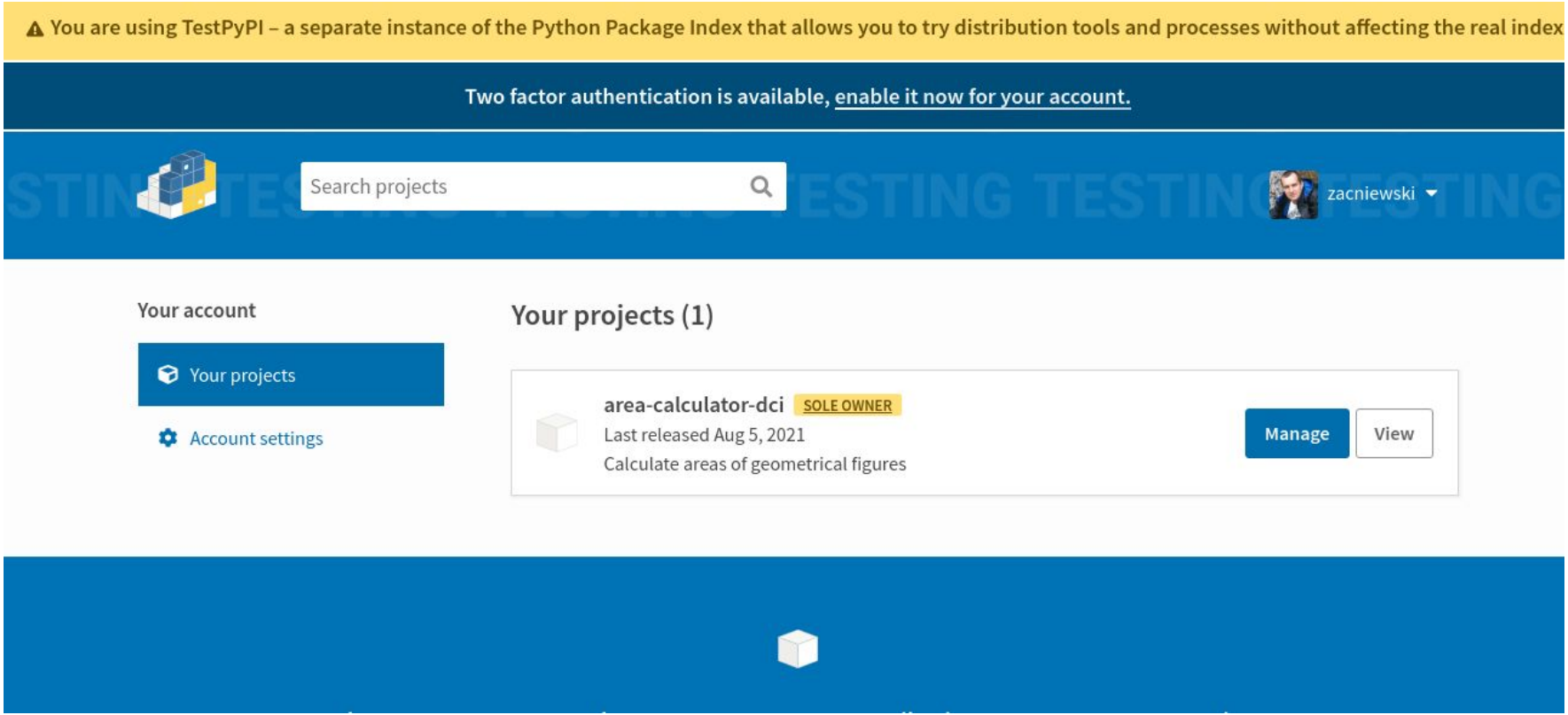
- **PyPI**: Python Package Index hosted at https://pypi.org/

- **TestPyPI**: a separate instance of the Python Package Index (PyPI) that allows you to try out the distribution tools and process without worrying about affecting the real index. TestPyPI is hosted at https://test.pypi.org

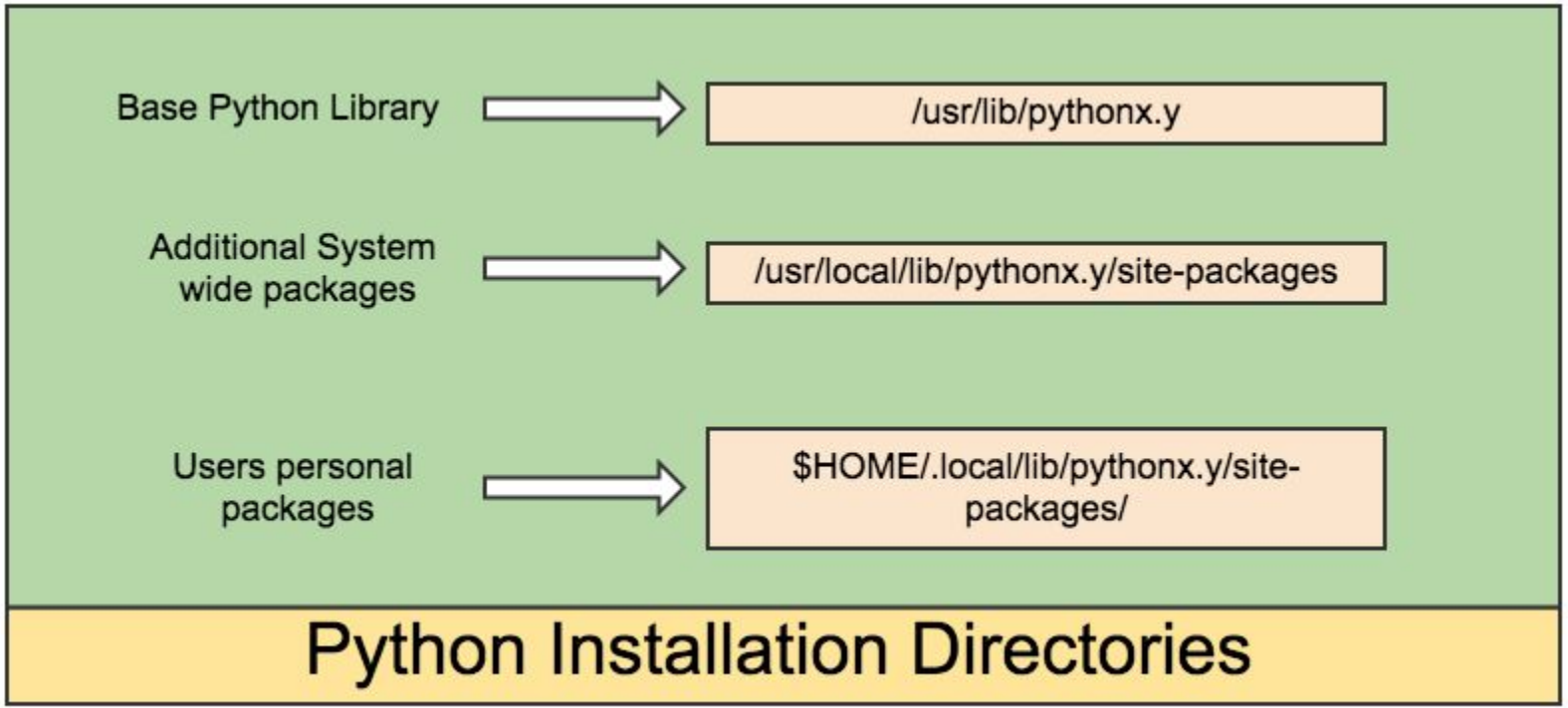
Package Index Instances - PyPI



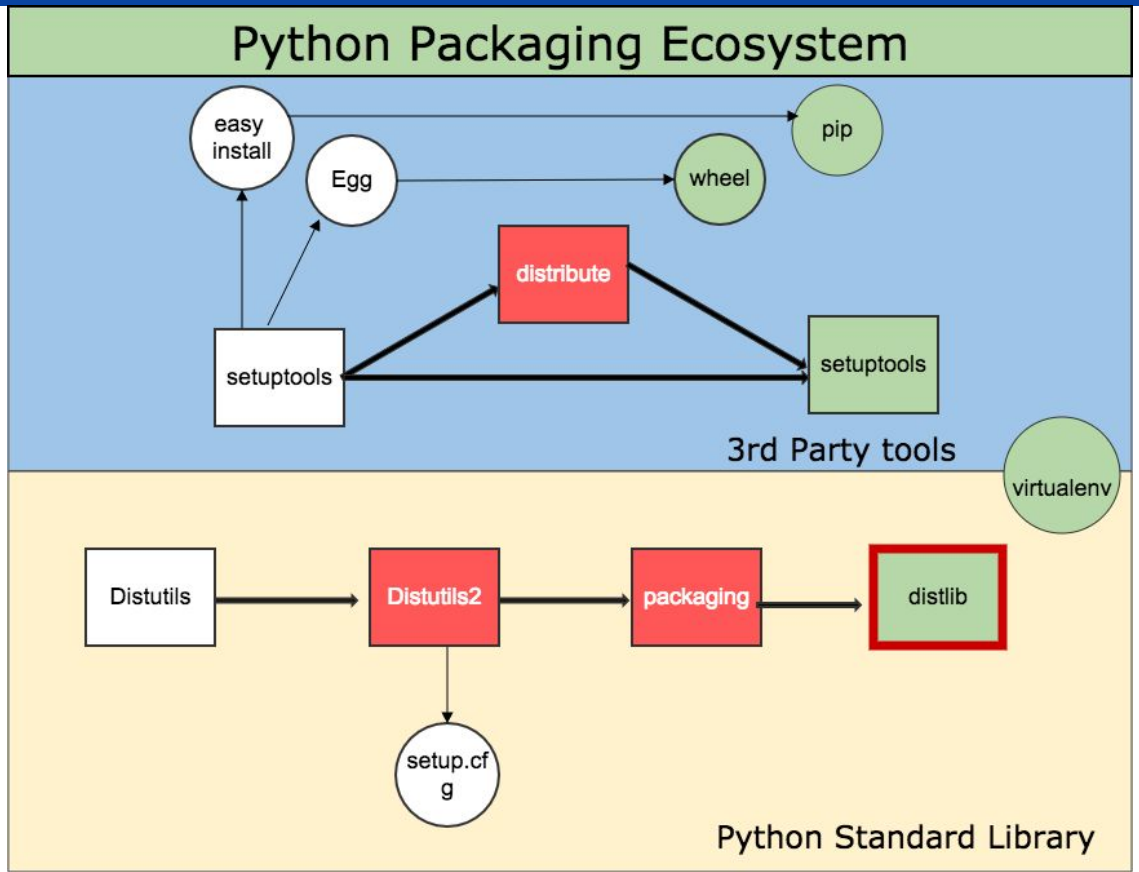
Package Index Instances - TestPyPI



Python Distribution in the Filesystem



Packaging Ecosystem



Module Distribution

- **Module distribution** is a collection of Python modules distributed together as a single downloadable resource and meant to be installed *en masse*.

- Examples of some well-known module distributions are NumPy, SciPy or Pillow

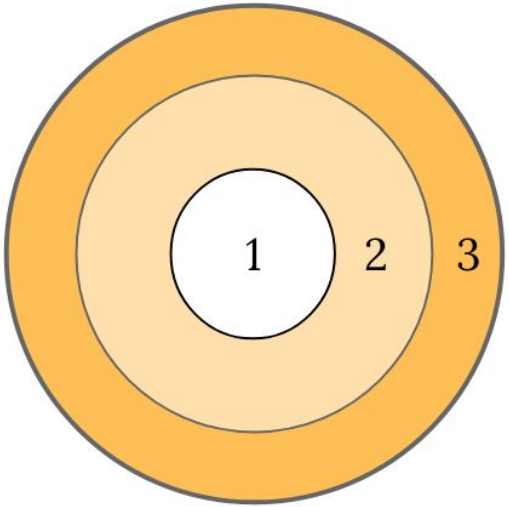
Recommended Tools

Digital Career Institute

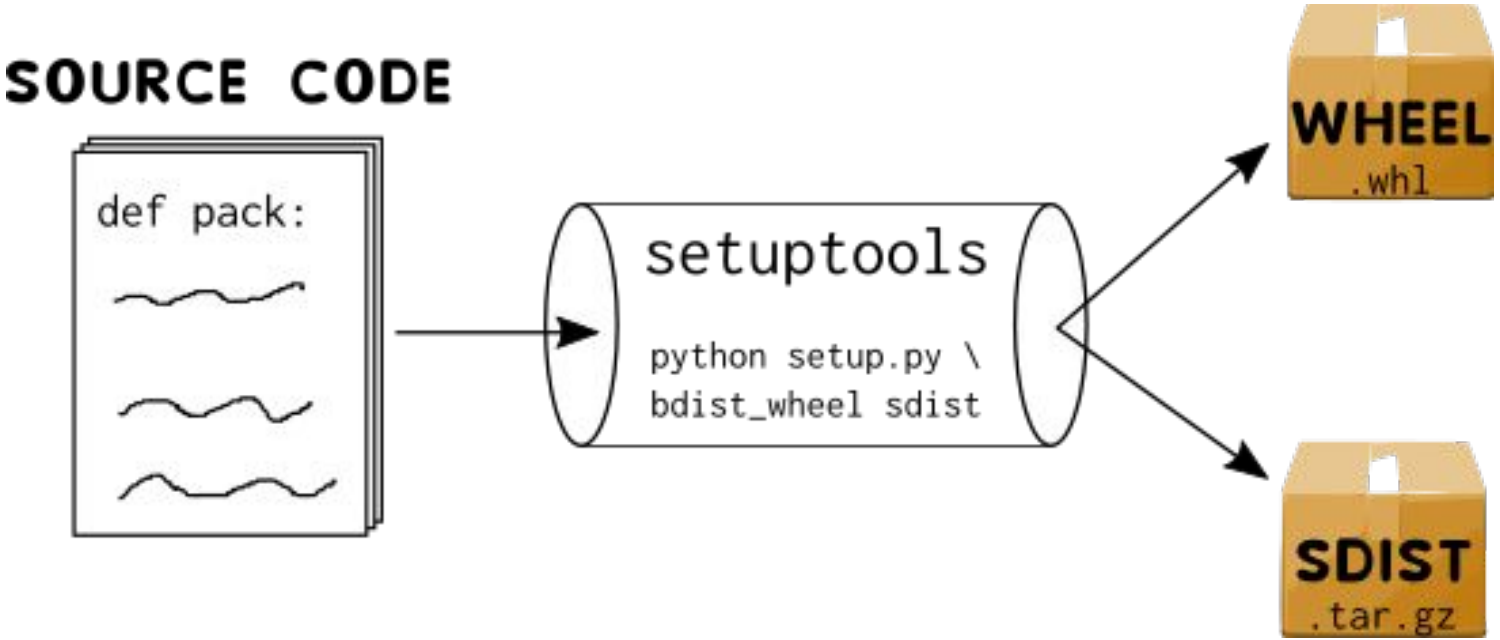
Packaging for Python **tools** and **libraries**

1. **.py** - standalone modules
2. **sdist** - Pure-Python packages
3. **wheel** - Python packages

(With room to spare for static vs. dynamic linking)



Python Packaging Formats



Python Distribution

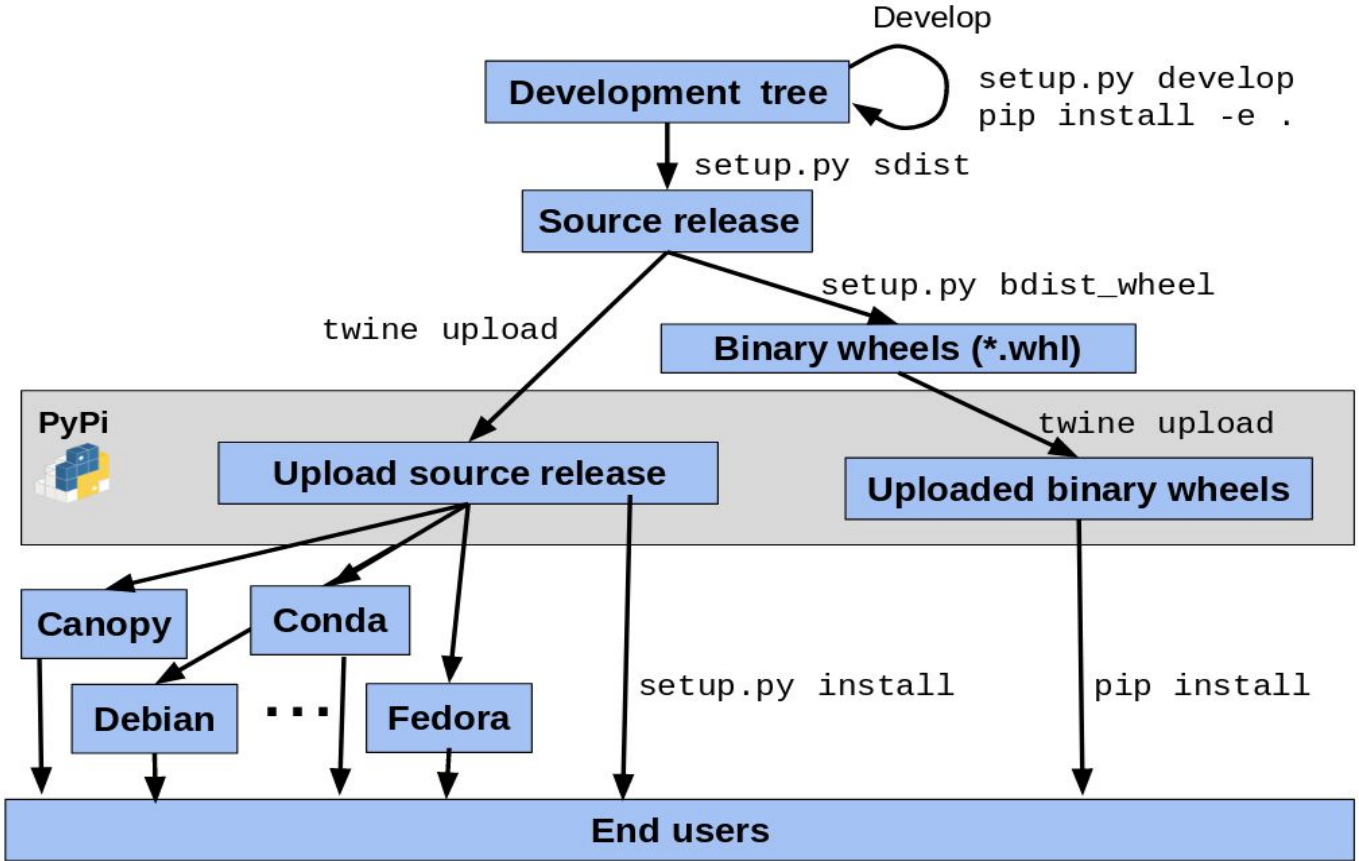
Digital Career Institute

- **pure**:
  - Not specific to a CPU architecture
  - No ABI (Application Binary Interface)

- **non-pure**
  - ABI
  - Platform specific

- Reference here

Python Package Lifecycle



Steps of Publishing Package to PyPI

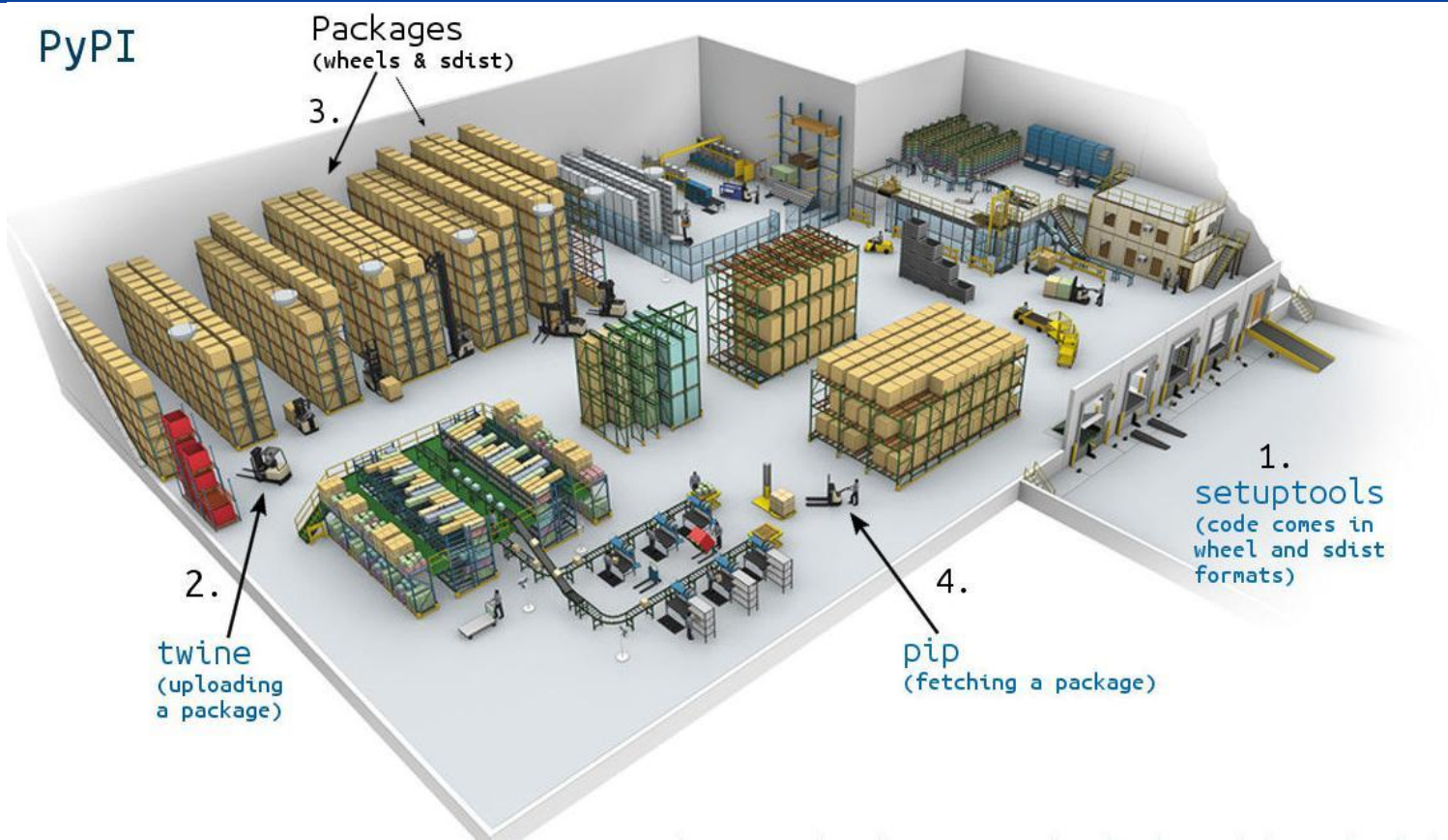
Digital Career Institute

- Following steps show particular stages of publishing your package to PyPI.

- Steps are based on package area-calculator-dci, created by one of DCI freelancers.

Steps of Publishing Package to PyPI

Digital Career Institute



Publishing to PyPI: Steps 1-3

1. Write your python module and save it under a directory.
2. Create the setup.py file with necessary information.
3. Choose a LICENSE and provide a `README.md` file for your project.

```
(my-env) [artur@artur-PC task2]$ tree -L 4 -I "bin|include|lib*|pyvenv.cfg"
.
├── area-calculator-package
│   ├── area-calculator-dci
│   │   └── src
│   │       └── area_calculator_dci.py
│   ├── LICENSE
│   ├── README.md
│   └── setup.py
└── my-env

4 directories, 4 files
```

Publishing to PyPI: Step #4

4. Generate the distribution archives on local machine.

```
(my-env) [artur@artur-PC task2]$ tree -L 5 -I "bin|include|lib*|pyvenv.cfg|pictures"
.
└── area-calculator-package
    ├── area-calculator-dci
    │   └── src
    │       ├── area_calculator_dci.egg-info
    │       │   ├── dependency_links.txt
    │       │   ├── PKG-INFO
    │       │   ├── SOURCES.txt
    │       │   └── top_level.txt
    │       └── area_calculator_dci.py
    ├── build
    │   └── bdist.linux-x86_64
    ├── dist
    │   ├── area_calculator_dci-0.0.1-py3-none-any.whl
    │   └── area-calculator-dci-0.0.1.tar.gz
    ├── LICENSE
    ├── my-env
    ├── README.md
    └── setup.py
```

Publishing to PyPI: Step #5

Digital Career Institute

5. Try installing the package on your local machine.

```
(my-env) $ pip install -e .

Obtaining file:///home/.../area-calculator-package
Installing collected packages: area-calculator-dci
  Running setup.py develop for area-calculator-dci
Successfully installed area-calculator-dci-0.0.1
```

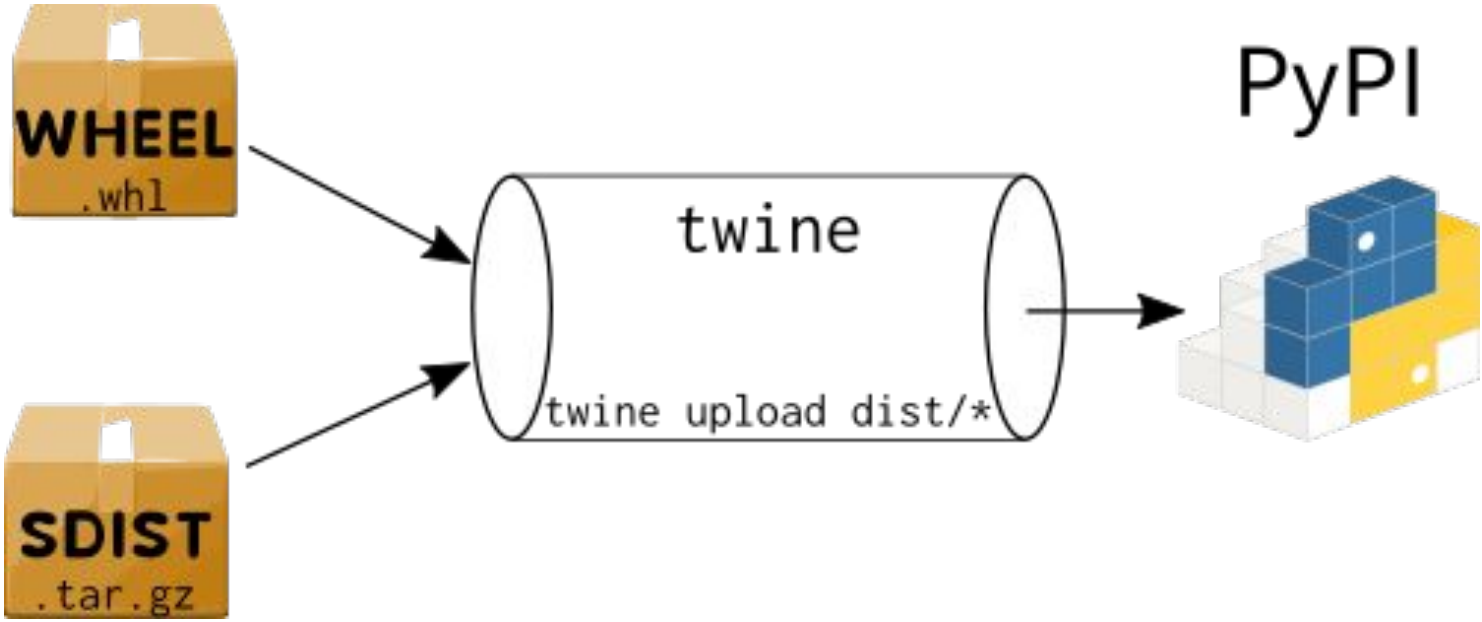
Publishing to PyPI: Step #5

Digital Career Institute

```
(my-env) $ python
>>> import area_calculator_dci
>>> area_calculator_dci.welcome()
Hello, welcome to area calculator package!
>>> area_calculator_dci.square(5)
25
```

Publishing to PyPI: Step #6

6. Publish the package to the TestPyPi repository to check if all works well.



Publishing to PyPI: Step #6

Digital Career Institute

- You can encounter errors, for example like this:

```
HTTPError: 400 Bad Request from https://test.pypi.org/legacy/

This filename has already been used, use a different version. See
https://test.pypi.org/help/#file-name-reuse for more information.
```

- Read carefully error information and make a correction.

Publishing to PyPI: Step #6

Digital Career Institute

Uploading package to TestPyPI:

```
(my-env) [artur@artur-PC area-calculator-package]$ twine upload --repository-url https://test.pypi.org/legacy/ dist/*
Uploading distributions to https://test.pypi.org/legacy/
Enter your username: zacniewski
Enter your password:
Uploading area_calculator_dci-0.0.2-py3-none-any.whl
100%|                                                                                | 5.99k/5.99k [00:01<00:00, 3.71kB/s]
Uploading area-calculator-dci-0.0.2.tar.gz
100%|                                                                                | 5.77k/5.77k [00:01<00:00, 4.27kB/s]

View at:
https://test.pypi.org/project/area-calculator-dci/0.0.2/
```

Publishing to PyPI: Step #6

Digital Career Institute

```
(my-env)$ pip uninstall area-calculator-dci
Found existing installation: area-calculator-dci 0.0.2
Uninstalling area-calculator-dci-0.0.2:
  Would remove:
    /home/.../site-packages/area-calculator-dci.egg-link
Proceed (Y/n)? y
  Successfully uninstalled area-calculator-dci-0.0.2
```

Publishing to PyPI: Step #6

Digital Career Institute

```
(my-env) pip install -i https://test.pypi.org/simple/ area-calculator-dci

Looking in indexes: https://test.pypi.org/simple/

Collecting area-calculator-dci

 Downloading
https://test-files.pythonhosted.org/packages/.../area_calculator_dci-0.0.2-py
3-none-any.whl (2.7 kB)

Installing collected packages: area-calculator-dci

Successfully installed area-calculator-dci-0.0.2
```

Publishing to PyPI: Step #7

Digital Career Institute

Finally, publish the package to the PyPi repository.

```
(my-env) [artur@artur-PC area-calculator-package]$ twine upload dist/*
Uploading distributions to https://upload.pypi.org/legacy/
Enter your username: zacniewski
Enter your password:
Uploading area_calculator_dci-0.0.2-py3-none-any.whl
100%|                                                                                | 5.99k/5
Uploading area-calculator-dci-0.0.2.tar.gz
100%|                                                                                | 5.77k/5

View at:
https://pypi.org/project/area-calculator-dci/0.0.2/
```

Publishing to PyPI: Step #7

Digital Career Institute

```
(my-env) $ pip uninstall area-calculator-dci
Found existing installation: area-calculator-dci 0.0.2
Uninstalling area-calculator-dci-0.0.2:
  Would remove:
    /home/.../site-packages/area_calculator_dci-0.0.2.dist-info/*
    /home/.../site-packages/area_calculator_dci.py
Proceed (Y/n)? y
  Successfully uninstalled area-calculator-dci-0.0.2
```

Publishing to PyPI: Step #7

Digital Career Institute

```
(my-env) $ pip install area-calculator-dci
Collecting area-calculator-dci
  Downloading area_calculator_dci-0.0.2-py3-none-any.whl (2.7 kB)
Installing collected packages: area-calculator-dci
Successfully installed area-calculator-dci-0.0.2
```

Now it's your turn 😃

At the core of the lesson

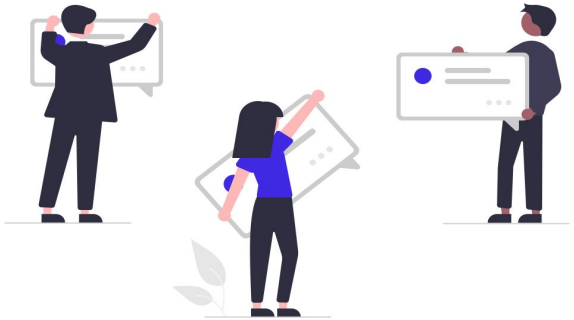
Digital Career Institute

Lesson learned:

- We know the idea of publishing packages on PyPI.
- We know how to prepare our project to publish on PyPI.

Self Study

Digital Career Institute



Topics

Expert Round

Digital Career Institute

At the core of the lesson

Digital Career Institute

Documentation

Documentation

1. Namespaces and scope in Python.
2. Python scope & the LEGB Rule.
3. Itertools module.
4. Glossary of Python.
5. Built-in function dir().
6. Regular and namespace packages.
7. Environment variables.
8. Glossary of Python packaging.
9. Garbage collection in Python.

Contact Details
DCI Digital Career Institute gGmbH

Digital Career Institute