Introduction to World Wide Web Architecture

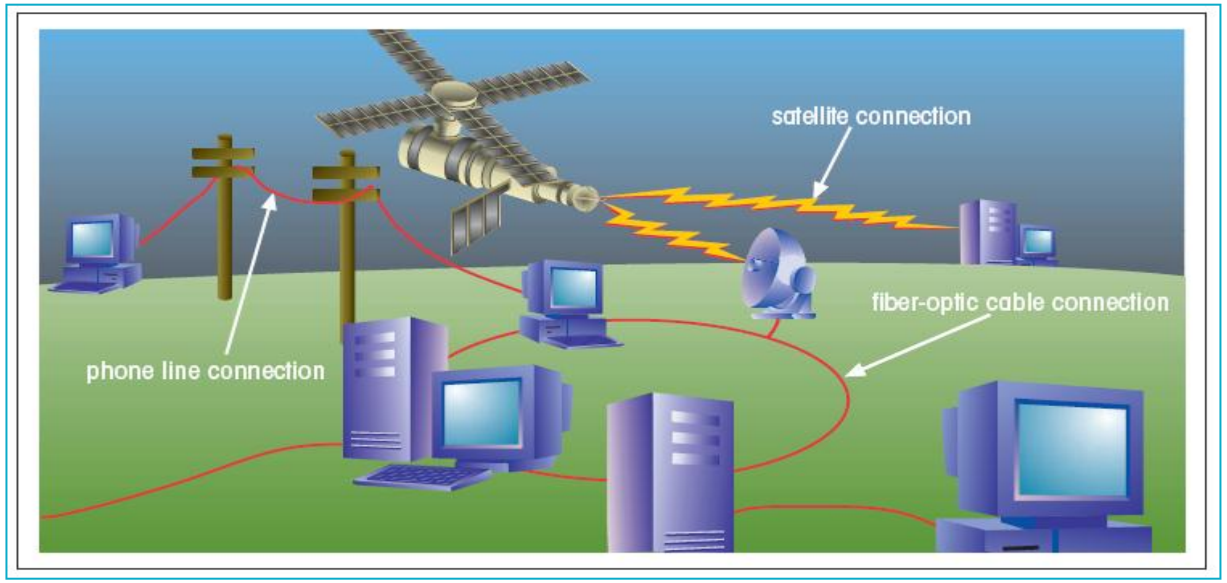
Week 5, Wednesday

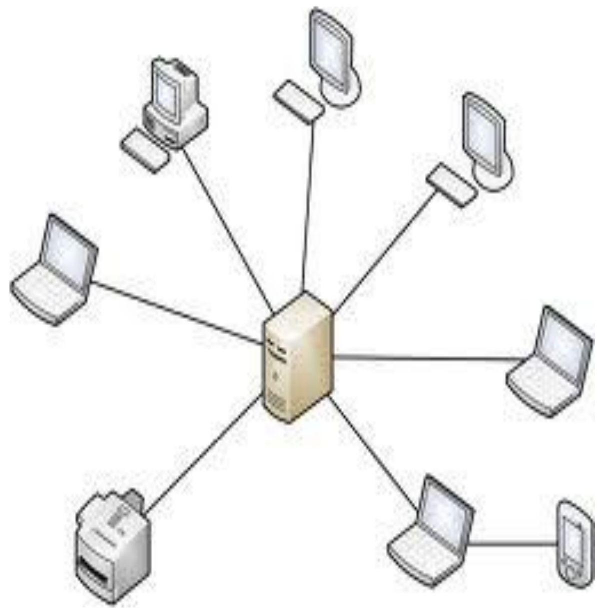
INTERNET - a global network of computers connected (through fiber-optic cables, satellites, phone lines, wireless access points …) with the purpose of sharing information

Users typically access a network through a computer called a host or node

A node that provides information or a service is called a server.

A computer or other device that requests services from a server is called a client.





World Wide Web (WWW)

- The **World Wide Web (WWW)**, or simply **Web**, is one way of accessing information over the medium of the Internet.
- WWW is an information-sharing model that is built on top of the Internet, where information is not accessed in a linear fashion, but allows documents to be connected using **hypertext links** forming a huge “web” of connected information.
- It is based on a protocol called **HTTP** protocol, only one of the languages spoken over the Internet, to transmit data.

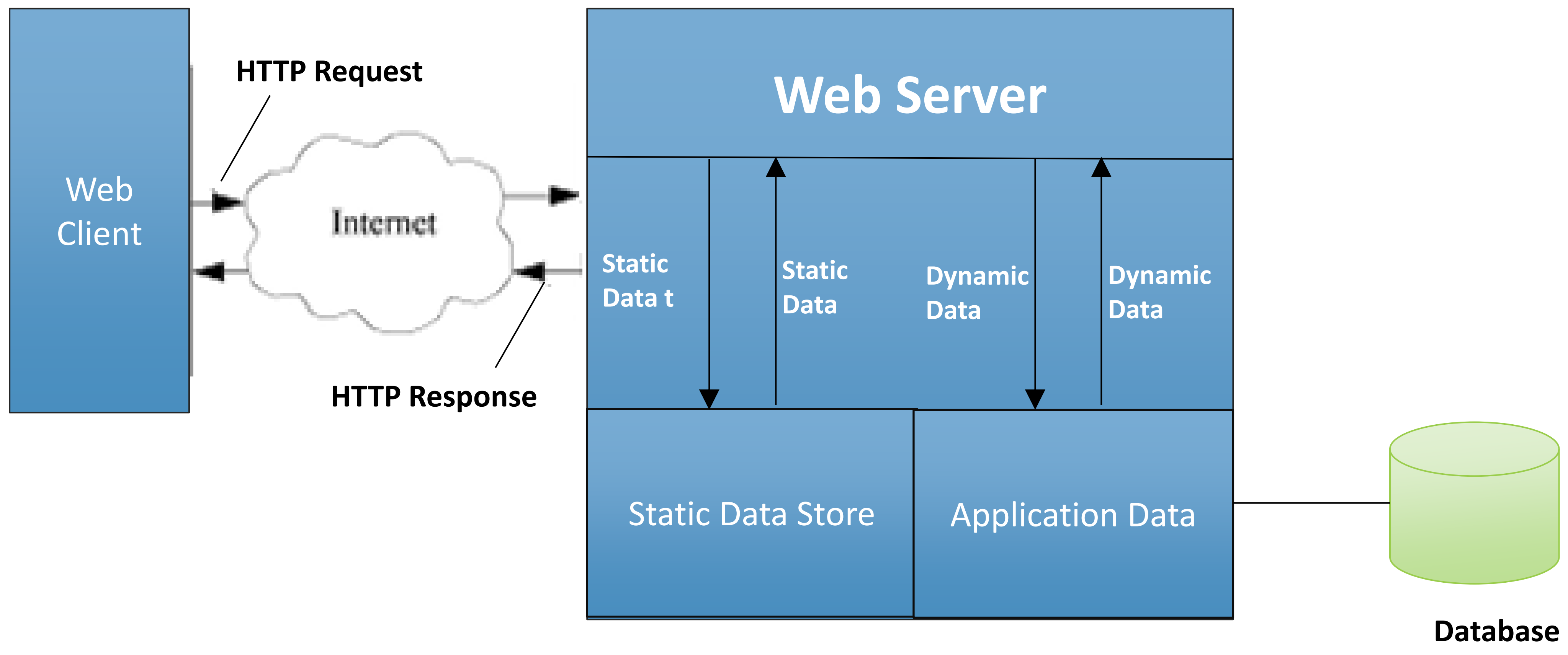
Web Pages And Web Servers

- Each document on the World Wide Web is referred to as a **Web page** and it is basically a text file written in HTML or Hypertext Markup Language
- Web pages are stored on a Web Servers, which are computers that run a special server software that enables communication between computers through the HTTP protocols and these servers make available any web page to any device connected to the Internet e.g., Apache HTTP server, Microsoft IIS server
- Each web page has a special address, **URL (Uniform Resource Locator)**

http://www.cse.unsw.edu.au/~cs2911/outline.html

↑ **protocol** ↑ **domain name of web server** ↑ **absolute path** ↑ **document name**

Web Server Working



How is a web page assembled?

- A client requests a **web page** by specifying the URL or clicking on a hyper link
- Browser sends a HTTP request to the server named in the URL and requests for the specific document
- The **web server** locates the requested file and sends a response.
  - If document is not found, an error message “404, Not found” is returned
  - If the document is found, the server returns the requested file to the browser
- The browser parses the HTML document and assembles the page
  - If the page contains images, the browser requests the server for the image, inserts the image into the document in the position indicated and displays the assembled page

HTML (HYPERTEXT MARKUP LANGUAGE)

- Web pages are text files written in HTML, a platform-independent, markup language that describes the content and structure of the document
- The **textual content** is represented by **plain text** and the **structure** of the document is specified by elements known as HTML tags

<element> textual content </element>

e.g., <p>, </p>

- HTML is **NOT** a **programming language**
- It is also **NOT** a **formatting language** i.e., it does not specify how the content should be rendered. The browser on the user's devices parses the HTML tags and renders the content. Use styles for formatting

Structure Of An HTML File

An HTML document is divided into two main sections: the **head** and the **body.**

- The **root** element <html> ….. </html>
- The **head** element <head> ….. </head>
  - It contains information about the document, for example the document title or the keywords.
  - The **title** element contains the page’s title. A document’s title is usually displayed in the browser’s title bar (or as a title of the tab)
  - The content of the head element is not displayed within the Web page.
- The **body** element <body> ….. </body>
  - It contains all of the content to appear on the Web page.
  - It contain code that tells the browser how to render the content.

Structure Of An HTML File

```
A basic HTML page has the following structure
<!DOCTYPE html>
<html>
     <head>
          <title> My Web Page </title>
          ... head content
     </head>
     <body>
          ... body content
     </body>
</html>
```

Introduction to Flask

Week 5, Wednesday

FLASK

- A **micro** web application framework written in Python, developed by Armin Ronacher
- As a micro-framework, aims to provide a simple, solid core but designed as *extensible* framework e.g., no native support for databases, authenticating users etc., but these key services available through *extensions* that integrate with the core package
- As a developer, you pick the extensions that are relevant to your project or even write your own custom extensions
- Flask relies on two main dependencies
  - Werkzeug which supports routing (request and response), utitlity functions such as debugging and WSGI (Web Server Gateway Interface – a standard interface between web server and we applications
  - Jinja2, a powerful templating language to render dynamic web pages

Python Functions And Decorators

Before we delve into Flask, let’s understand Python Functions and Decorators…

PYTHON FUNCTION DECORATORS

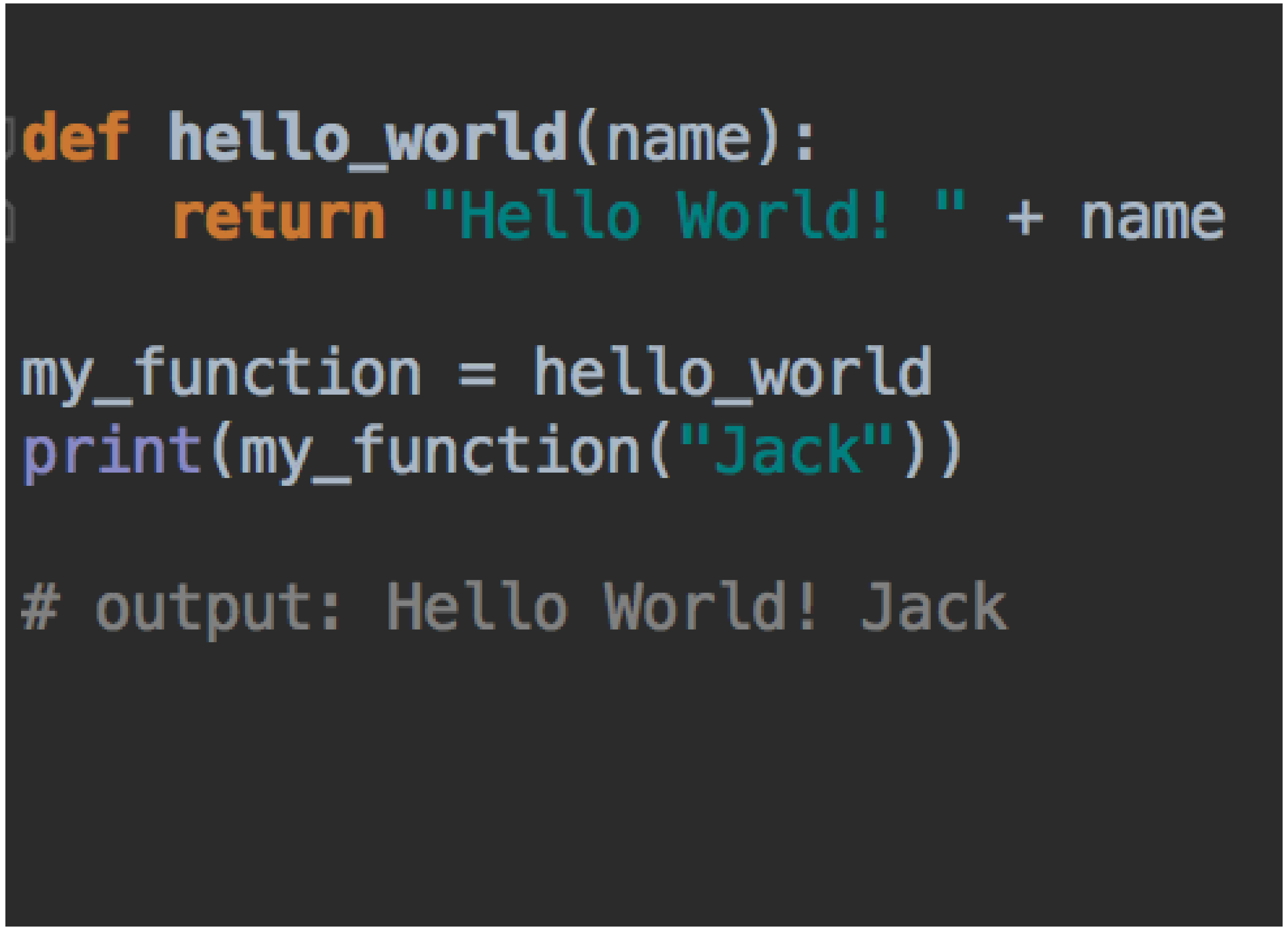
What you need to know about functions.

In python, functions are like other data types (e.g. number, string, list) which means we can do a lot of useful operations on them:

- 1. Assigns function to variables
- 2. Define functions inside another function
- 3. Functions can be passed as parameters to other functions
- 4. Functions can return other functions

Assign Function To Variable

- Define a function which takes a name and returns string “Hello World”+name.
- Assign this function to a variable named my_function.
- Call my_function using print(my_function(“Jack”))
- This is the same as calling hello_world(“Jack”)



```
def hello_world(name):
    return "Hello World! " + name

my_function = hello_world
print(my_function("Jack"))

# output: Hello World! Jack
```

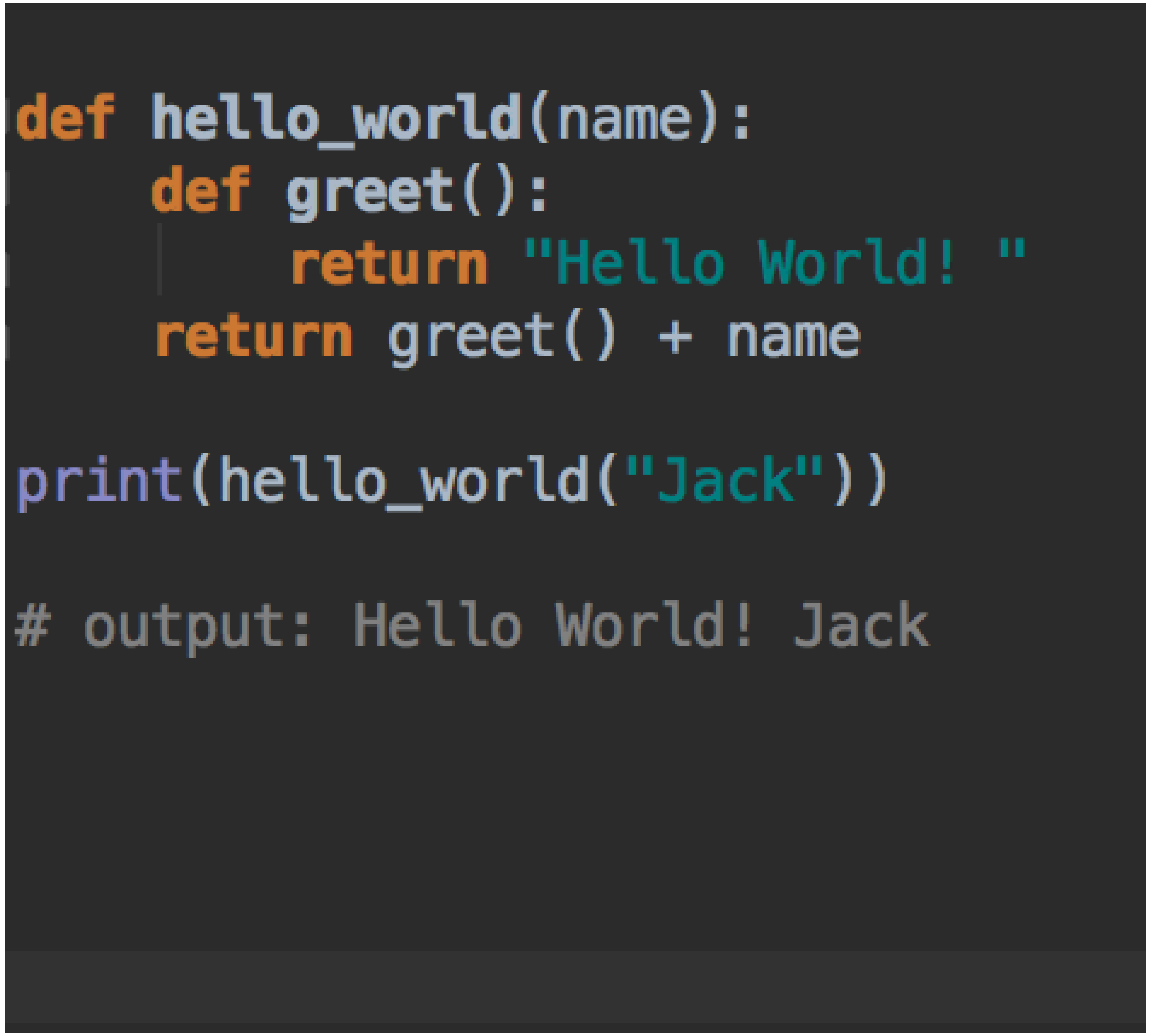
Nesting Functions

Define a function greet() inside the function hello_wold

The nested function always returns a string “Hello World! ”.

Now call hello_world passing in an argument

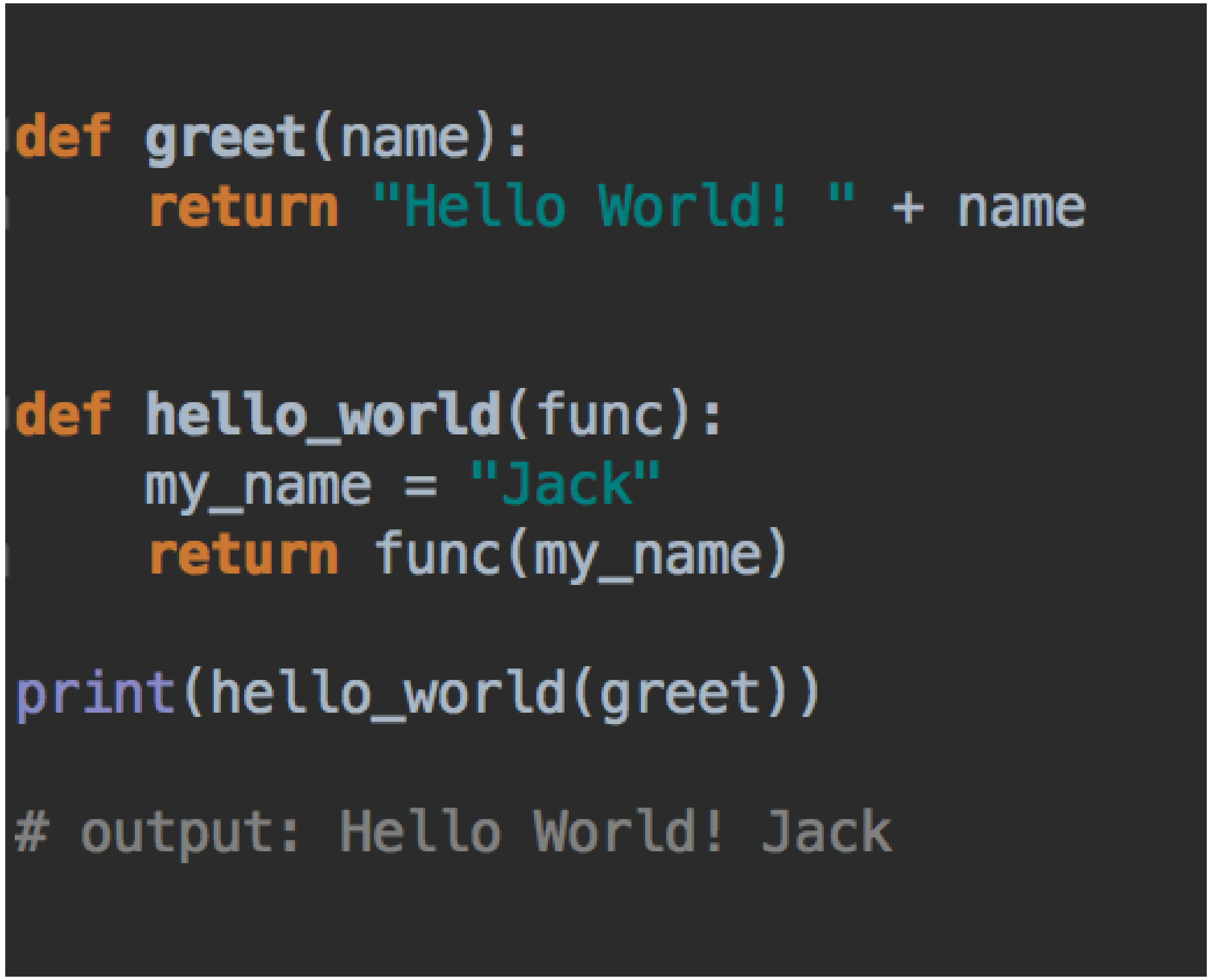
The result is the same as “Hello World! Jack”



```python
def hello_world(name):
    def greet():
        return "Hello World! "
    return greet() + name

print(hello_world("Jack"))

# output: Hello World! Jack
```

Functions Can Be Passed As Parameters To Other Functions

- Define a function greet() which takes in a name and returns a string “Hello World! ” + name
- hello_world(func) function takes in greet() as argument and returns the result of running the greet function with parameter.
- Return the result same as the greet(), which is “Hello World! Jack”



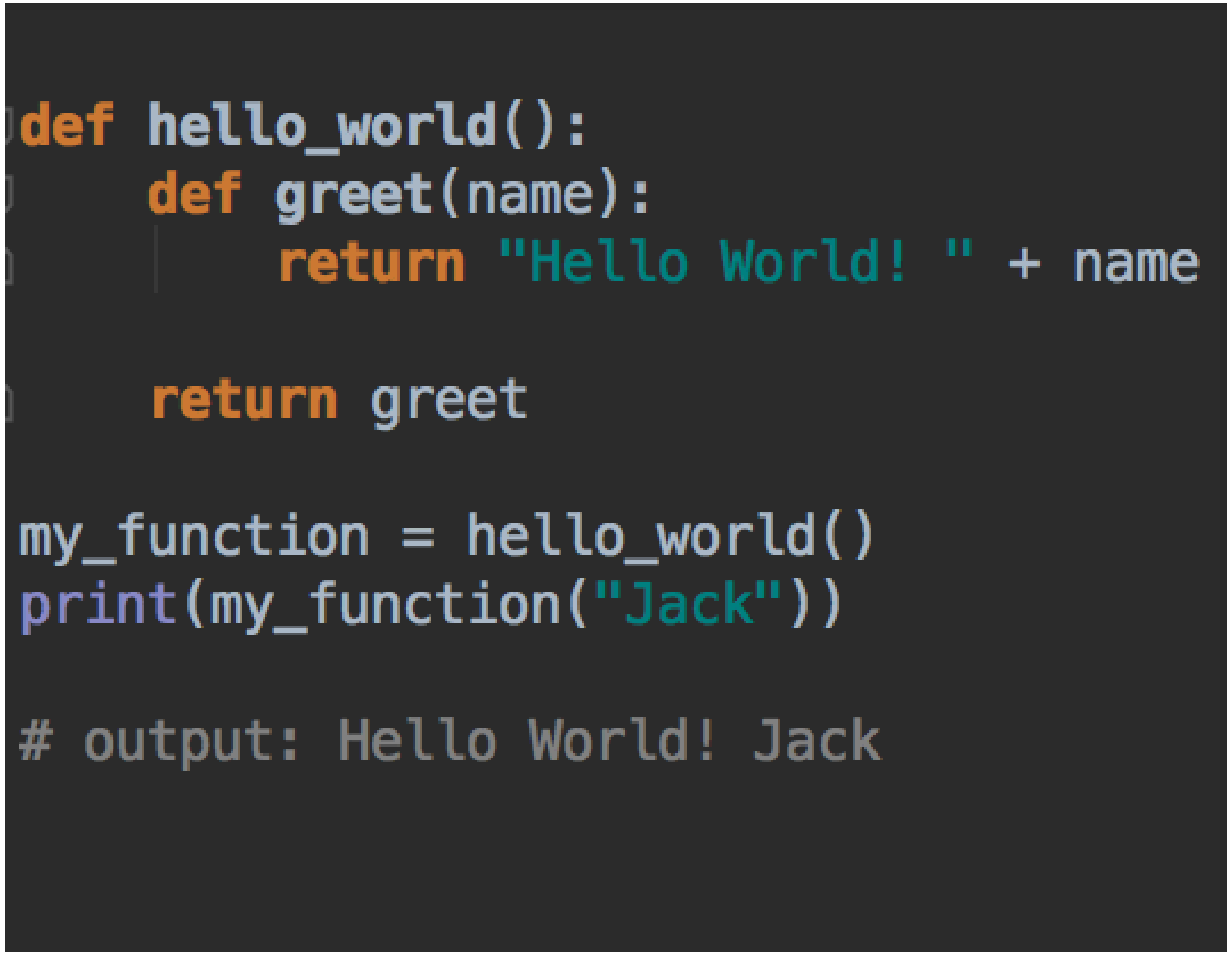
```
def greet(name):
    return "Hello World! " + name


def hello_world(func):
    my_name = "Jack"
    return func(my_name)

print(hello_world(greet))

# output: Hello World! Jack
```

FUNCTIONS CAN RETURN OTHER FUNCTIONS

Define a function greet() which always returns a string “Hello World! ”.

Then make the hello_world(name) function return a string of greet() + name.

The result is the same as “Hello World! Jack”



```python
def hello_world():
    def greet(name):
        return "Hello World! " + name

    return greet

my_function = hello_world()
print(my_function("Jack"))

# output: Hello World! Jack
```

COMPOSITION OF DECORATORS

Function decorators are simply wrappers to existing functions.

Applying what we have learnt so far, we can now build a function decorator.

COMPOSITION OF DECORATORS

Define a decorator function which:

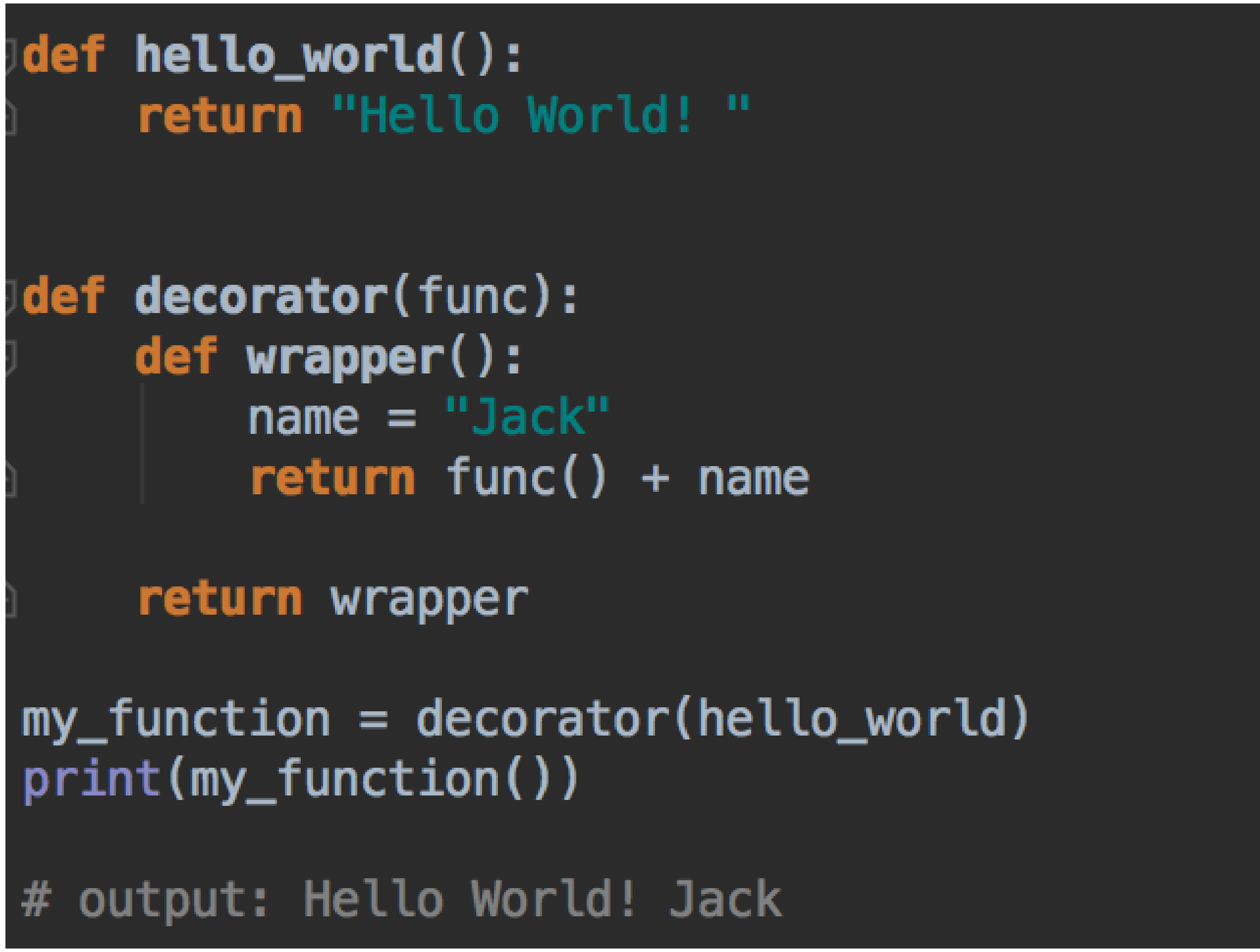
accepts a function as an argument, func

defines a nested function wrapper()

calls wrapper() and returns the result from this function

e.g., here the decorator function takes in hello_world as an argument, invokes wrapper(), which appends the name "jack" to the function hello_world(), returning a modified function that can be used anywhere.

If the function hello_world is to be decorated by the decorator function, we assign a variable, my_function to the result of decorator(hello_world).



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


def decorator(func):
    def wrapper():
        name = "Jack"
        return func() + name

    return wrapper

my_function = decorator(hello_world)
print(my_function())

# output: Hello World! Jack
```

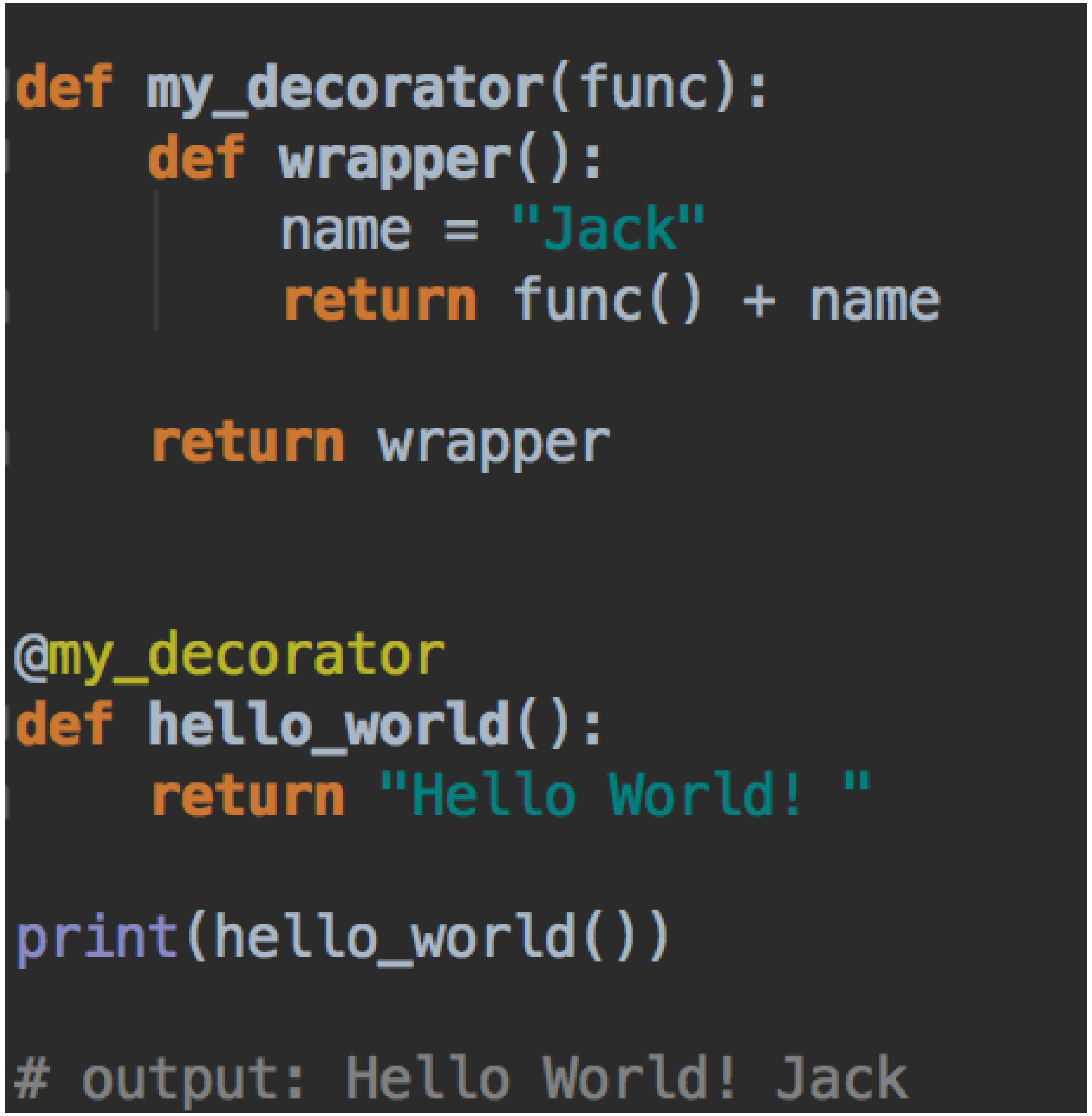
PYTHON'S DECORATOR SYNTAX

Python makes creating and using decorators a bit cleaner and simpler for the programmer

To decorate hello_world , we don't require an assignment to a variable (my_function = my_decorator(hello_world)

A neater shortcut – specify the decorating function before the function to be decorated.

e.g.@ my_decorator



```
def my_decorator(func):
    def wrapper():
        name = "Jack"
        return func() + name

    return wrapper


@my_decorator
def hello_world():
    return "Hello World! "

print(hello_world())

# output: Hello World! Jack
```

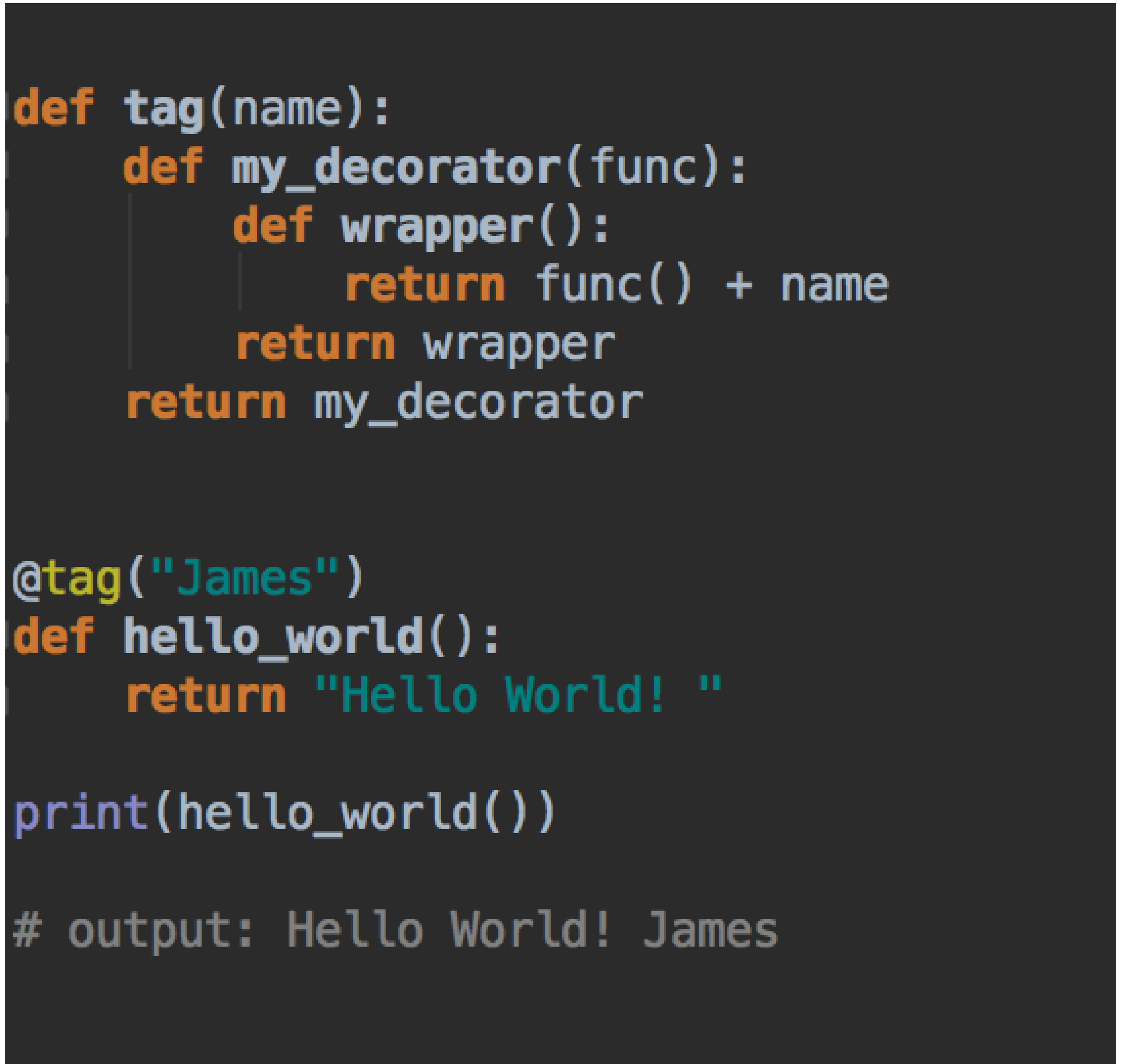
PASSING ARGUMENT TO DECORATOR

In the previous example, name inside the wrapper function was hard-code to “Jack”.

name could actually be passed in as argument to the decorator function so that a decorator function can have several ways for decoration.

Like here, we pass “James” to the decorator using @tag(“James”) and put the decorator function inside the tag function.

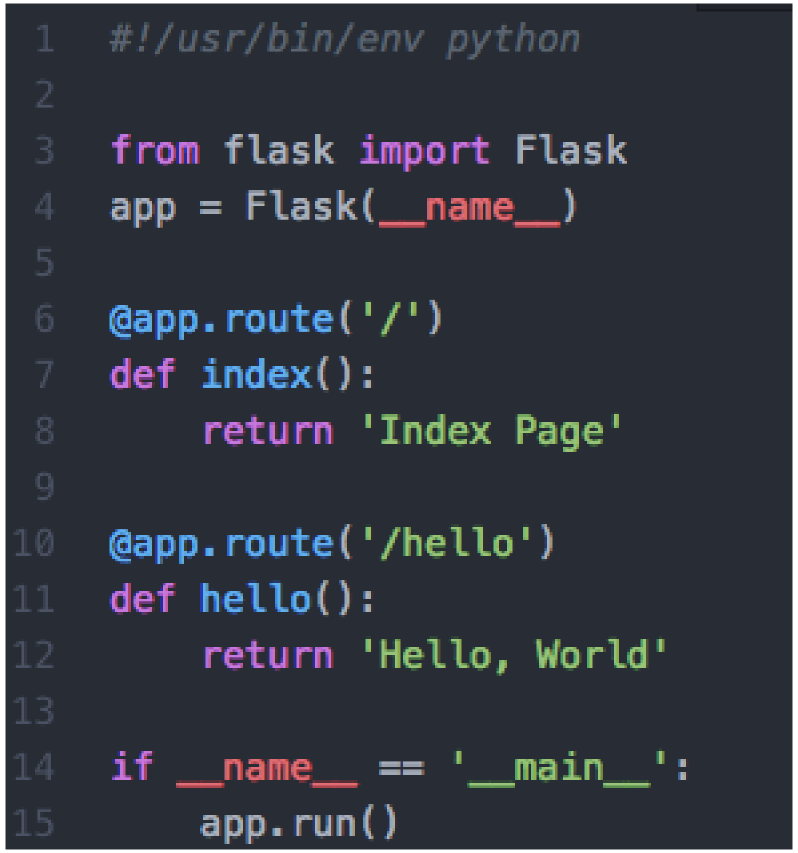
Now the hello_world is decorated with my_decorator with “James” as name which is passed above the hello_world function. So the output is “Hello World! James”



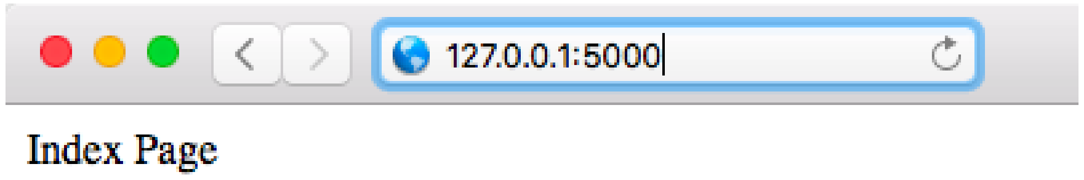
```
def tag(name):
    def my_decorator(func):
        def wrapper():
            return func() + name
        return wrapper
    return my_decorator


@tag("James")
def hello_world():
    return "Hello World! "

print(hello_world())

# output: Hello World! James
```

@app_route Function Decorator

We can use @app_route decorator/annotation to specify functions to be executed at a specific path of the web application, that is, to bind a function to an URL

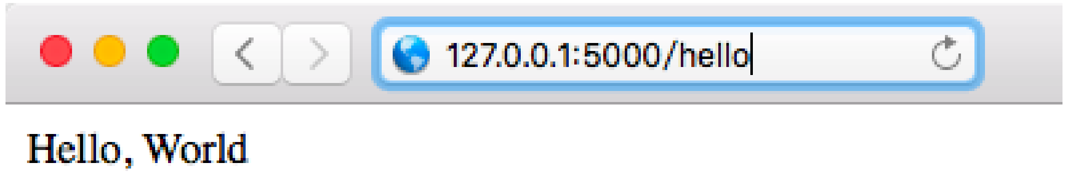


```python
#!/usr/bin/env python

from flask import Flask
app = Flask(__name__)

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

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

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



127.0.0.1:5000

Index Page

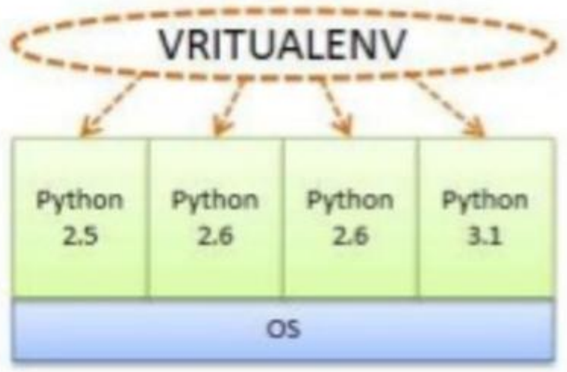


127.0.0.1:5000/hello

Hello, World

USING VIRTUAL ENVIRONMENTS

- A recommended approach to install Flask is to use a virtual environment (*virtualenv*)
  - A "*virtualenv*" is a private copy of the Python interpreter onto which you can install packages privately
  - A "*virtualenv*" enables multiple parallel installations of different versions of Python, one for each project, keeping project environments isolated



- Installing a virtual environment

```
pip3 install virtualenv
virtualenv –version
```

- Once you have *virtualenv* installed, you can launch a shell and create your own virtual environment

```
$ mkdir myproject
$ cd myproject
$ virtualenv myproject
(myproject) $
```

- Activating

```
$ myproject/bin/activate # OSX
> myproject/scripts/activate # win
```

Deactivating

```
$ myproject/bin/deactivate # OSX
> myproject/scripts/deactivate # win
```

A MINIMALIST FLASK APP

- Installing Flask (within your virtual environment or globally)

```
(hello) $ pip3 install flask
```

- Coding

```
# Import Flask Library
from flask import Flask

# create a Flask application instance
app = Flask(__name__)

# define a route through the app.route decorator
@app.route("/")
def index():
    return '<h1> Hello World </h1>'

# launch the integrated development web server
# and run the app on http://localhost:8085

if __name__=='__main__':
   app.run(debug=True,port=8085)
```

Route decorator: binds a function to a URL

View function to render HTML page to browser

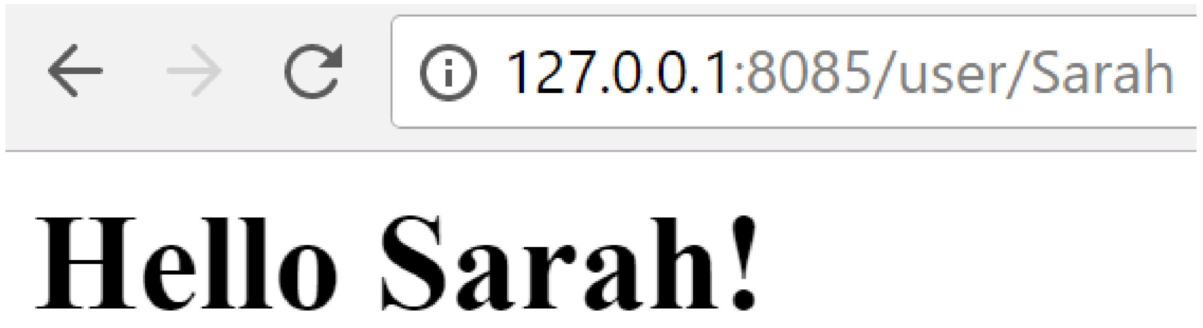
- Running

```
(hello) $ python hello.py
 * Restarting with stat
 * Debugger is active!
 * Debugger PIN: 281-144-438
 * Running on http://127.0.0.1:8085/ (Press CTRL+C to quit)
127.0.0.1 - - [18/Jul/2017 09:10:19] "GET / HTTP/1.1" 200 -
127.0.0.1 - - [18/Jul/2017 09:10:19] "GET /favicon.ico
HTTP/1.1" 404 -
```

A FLASK APPLICATION WITH A DYNAMIC ROUTE

- Add variable parts to a URL my marking as <variable name> and pass this as a keyword argument to your function

```
# Import Flask Library
from flask import Flask

# create a Flask application instance
app = Flask(__name__)

# define a route through the app.route decorator
@app.route("/")
def index():
    return '<h1> Hello World </h1>'

# define a route through the app.route decorator
@app.route("/user/<name>")
def user(name):
    return '<h1> Hello World %s </h1>' %name

# launch the integrated development web server
# and run the app on http://localhost:8085

if __name__=='__main__':
   app.run(debug=True,port=8085)
```


127.0.0.1:8085/user/Sarah

Hello Sarah!

Jinja

- Good application design involves:
  - Writing clean and well-structured code
  - Decoupling business and presentation logic, mixing the two leads to code difficult to understand and maintathis in
- Flask provides decoupling through moving the presentation logic into *templates,* using a powerful templating language, Jinja2
- *Jinja2 templates* are:
  - essentially *.html files with static response text along with placeholder variables and programming logic for the dynamic parts
  - use delimiters such as {%...%} for embedding programming logic and {{...}} for outputting the results of an expression or variable
  - the process of replacing the variables with actual values and returning a final response is called ***rendering***
  - by convention, live in the /templates directory

HELLOWORLD WITH JINJA2

- Split the view function into two templates: index.html and user.html
- By default Flask looks for templates in a *templates* folder

```
templates/index.html:
<h1> Hello World! </h1>

templates/user.html:
<h1> Hello, {{ name}} </h1>
```

- Rendering templates
  - modify the view function to render these templates using function *render_template*
  - the function takes the filename of the template as its first argument and additional arguments as key/value pairs
  - the {{name}} references a placeholder variable, which can be modified with *filters* e.g., Hello, {{ name| capitalize}}

```
from flask import Flask, render_template
#render_template integrates Jinja2 template engine

# define a route through the app.route decorator
@app.route("/”)
def index():
    return render_template(‘index.html’)

@app.route("/user/<name>")
def user(name):
    return render_template(‘user.html’,name=name)
```

Jinja2 Control Structures…

```
{# Jinja2 offers several control structures to alter the flow of the template #}
{# Conditional statements in a template #}

{% if user %}
   Hello, {{ user }}
{% else %}
   Hello, Stranger!
{% endif %}

{# Using a for loop to render a list of contents #}

<ul>
{% for number in numbers %}
   <li> {{ number }} </li>
{% endfor %}
</ul>

{# Portions of template code that need to be repeated in several places
stored in a
   separate file and included from all the templates to avoid repetition #}

{% include ‘common.html’}
```