COMP 1531

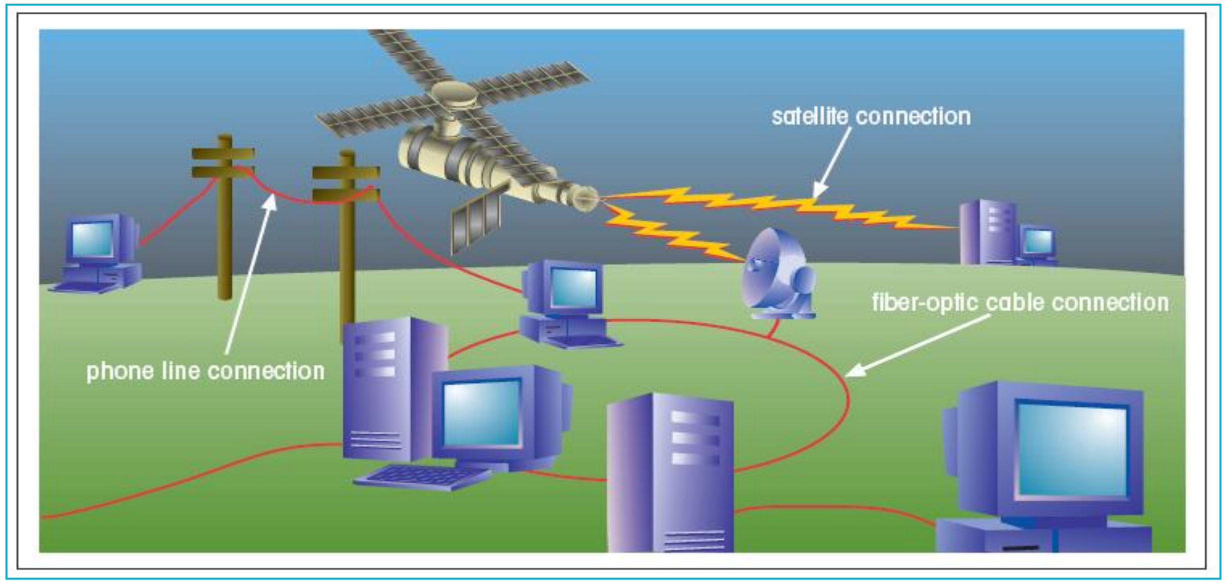
Software Engineering Fundamentals

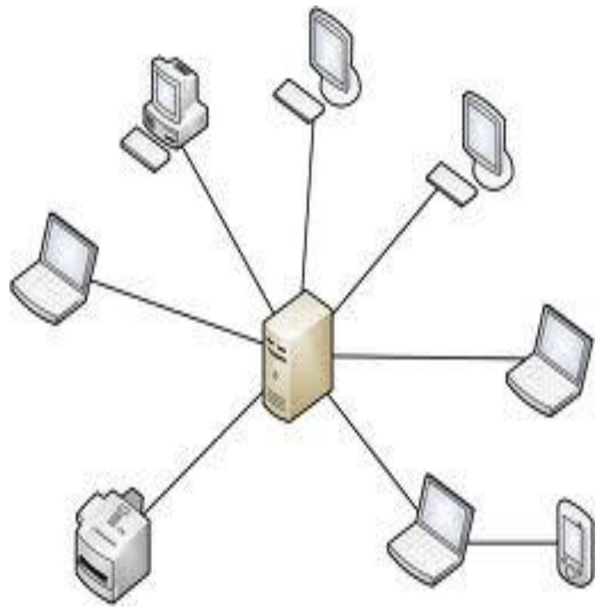
Week 04, Wednesday

Introduction to WWW and Web Architecture

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.

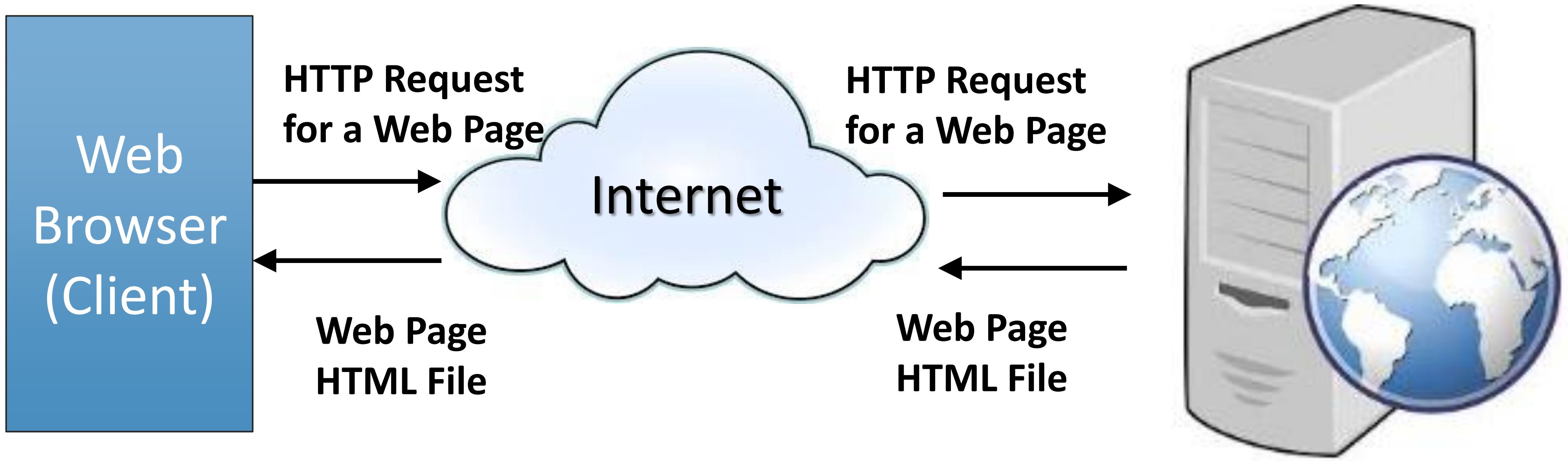




World Wide Web (WWW)

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

Web Architecture



Web Pages

- 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 and stored on a Web 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**

- HTML files are viewed using a web browser

Web Servers and Web Browsers

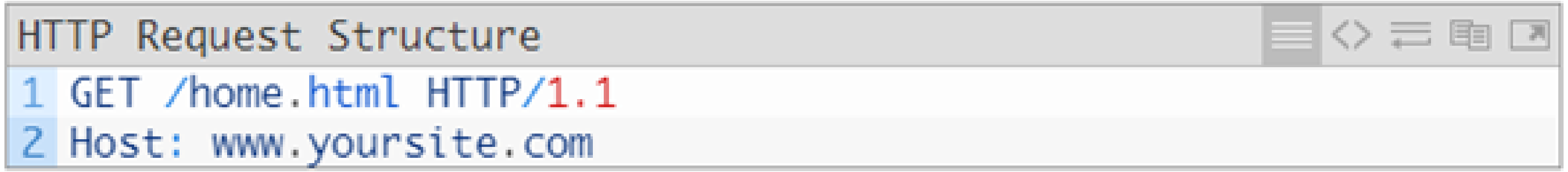
- A Web Server is a computer that runs a special server software that enables communication between computers through the HTTP protocols and these servers make available any web page to any device connected to the Internet e.g., Apache HTTP server, Microsoft IIS server

- A Web Browser is a software application running on the client for retrieving and rendering a web page to an end-user e.g., Internet Explorer, Chrome

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 web server named in the URL and requests for the specific document
- The **web server** locates the requested file and sends a HTTP 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

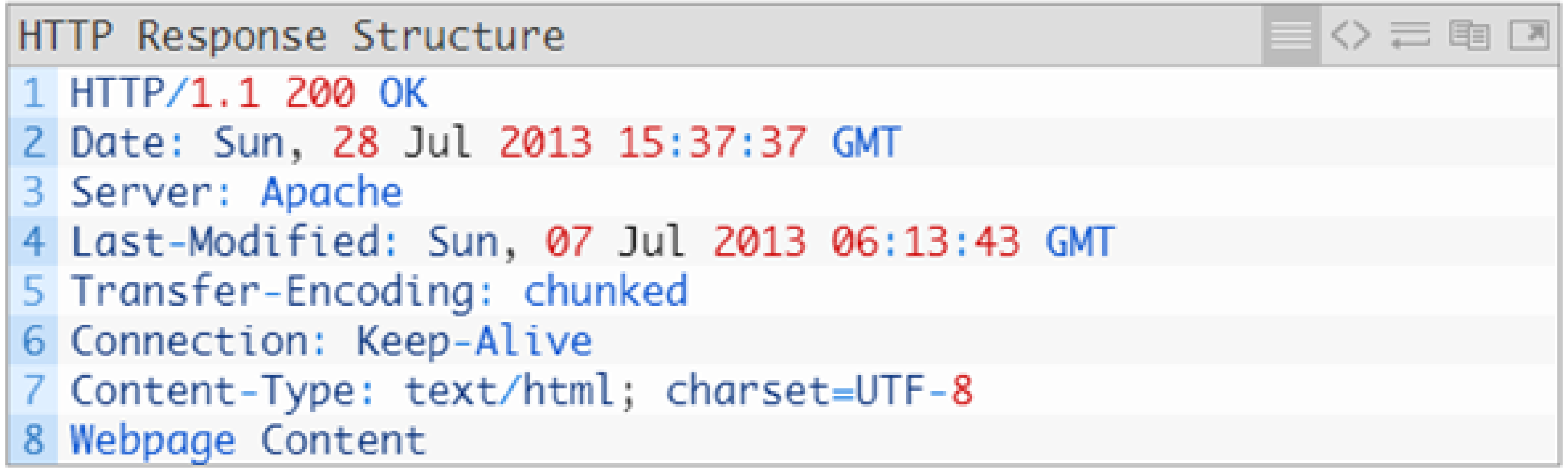
A HTTP Request


HTTP Request Structure

```
GET /home.html HTTP/1.1
Host: www.yoursite.com
```

- A Request message consists of:
  - Request Line (GET /home.html HTTP/1.1)
  - Headers
  - An optional message body
- Common HTTP request methods:
  - HEAD, GET, POST, DELETE

A HTTP Response



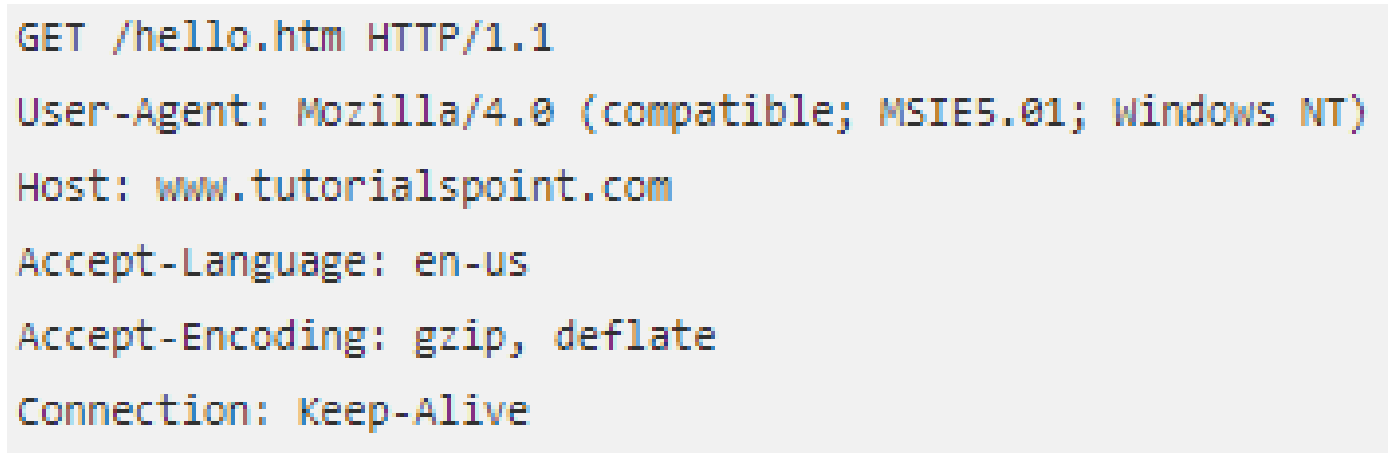
HTTP Response Structure

```
HTTP/1.1 200 OK
Date: Sun, 28 Jul 2013 15:37:37 GMT
Server: Apache
Last-Modified: Sun, 07 Jul 2013 06:13:43 GMT
Transfer-Encoding: chunked
Connection: Keep-Alive
Content-Type: text/html; charset=UTF-8
Webpage Content
```

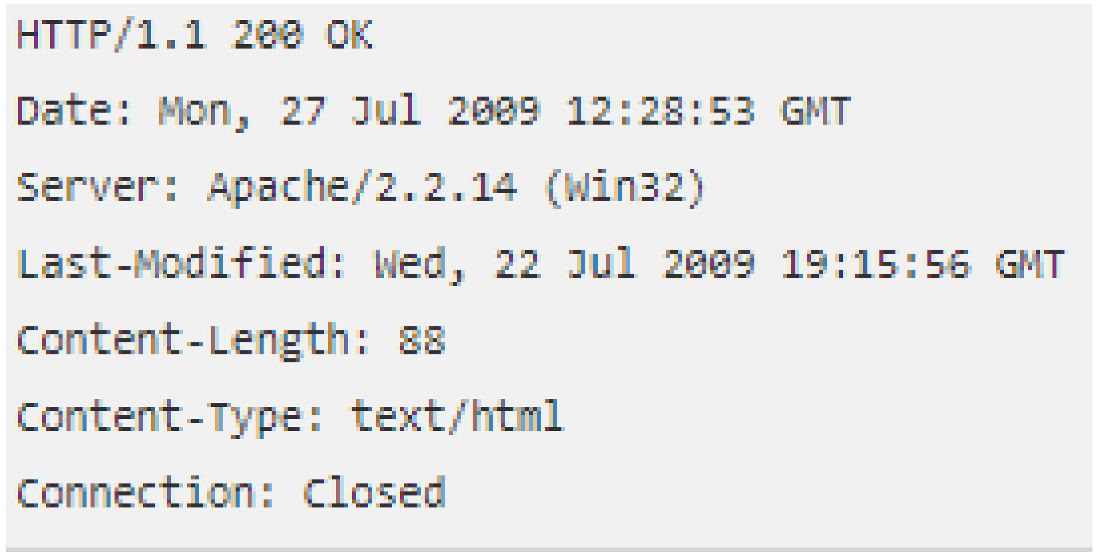
A HTTP response consists of:

- A Status Line that includes a status code
  - Success: 2xx
  - Redirection: 3xx
  - Client-Error: 4xx
  - Server-Error: 5xx
- Headers
- An optional message body

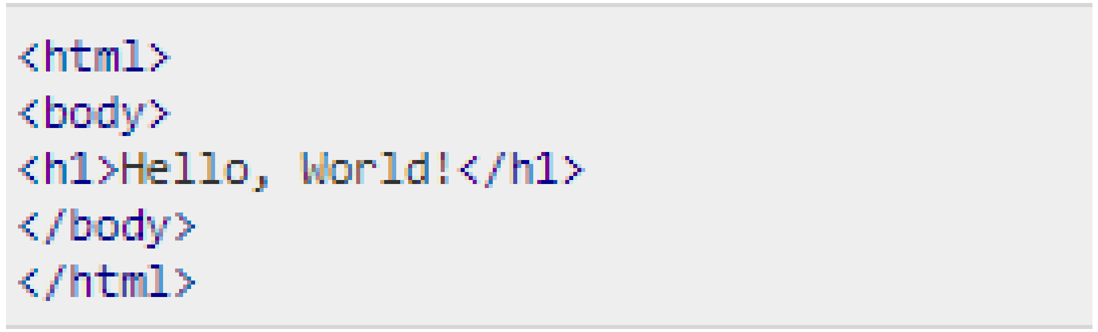
A sample HTTP Request and Response



```
GET /hello.htm HTTP/1.1
User-Agent: Mozilla/4.0 (compatible; MSIE5.01; Windows NT)
Host: www.tutorialspoint.com
Accept-Language: en-us
Accept-Encoding: gzip, deflate
Connection: Keep-Alive
```



```
HTTP/1.1 200 OK
Date: Mon, 27 Jul 2009 12:28:53 GMT
Server: Apache/2.2.14 (Win32)
Last-Modified: Wed, 22 Jul 2009 19:15:56 GMT
Content-Length: 88
Content-Type: text/html
Connection: Closed
```



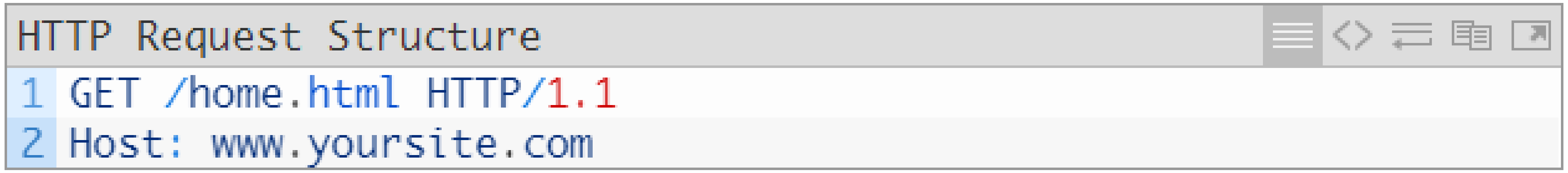
```
<html>
<body>
<h1>Hello, World!</h1>
</body>
</html>
```

HTTP session state

- HTTP is a stateless protocol
  - Server and client are only aware of each other during a current request
  - Neither client nor server can retain information between different requests across web pages
- How is it possible the customise the content of a website for a user e.g., a shopping
  - Cookies (a small piece of text stored on user’s computer)
  - Sessions
  - Hidden variables (when the current page is a form)

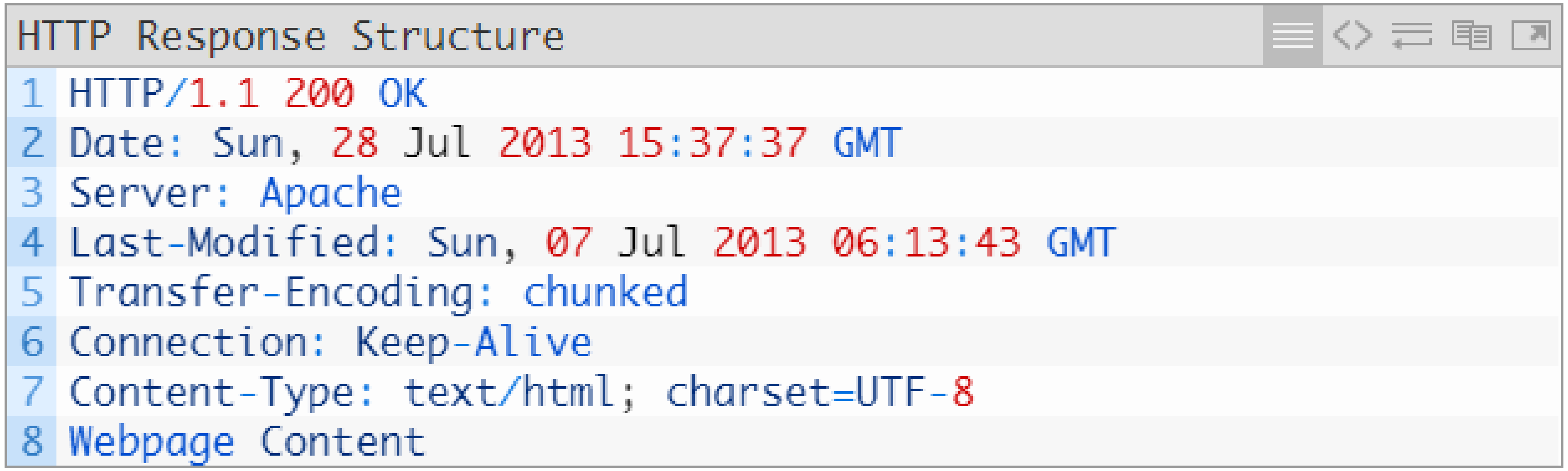
A HTTP Request and Response

HTTP defines eight possible Request methods: HEAD, GET, POST, PUT, DELETE, TRACE, OPTIONS,CONNECT


HTTP Request Structure

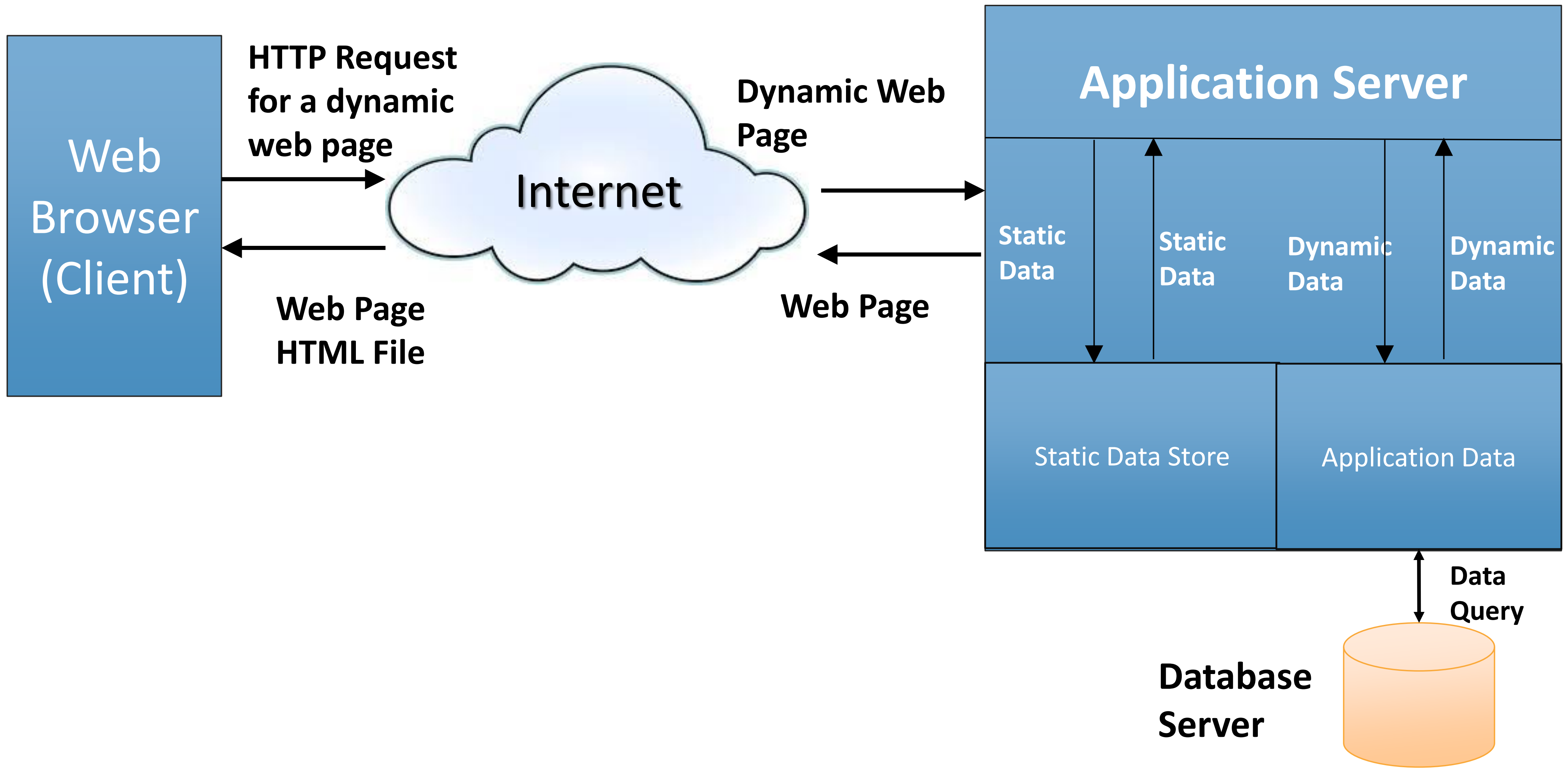
```
GET /home.html HTTP/1.1
Host: www.yoursite.com
```

The response from the web server should look like below:


HTTP Response Structure

```
HTTP/1.1 200 OK
Date: Sun, 28 Jul 2013 15:37:37 GMT
Server: Apache
Last-Modified: Sun, 07 Jul 2013 06:13:43 GMT
Transfer-Encoding: chunked
Connection: Keep-Alive
Content-Type: text/html; charset=UTF-8
Webpage Content
```

Extended Web Architecture



Server-Side Processing

- Server-Side processing (e.g., PHP, Perl, J2EE, FLASK, Django, Ruby on Rails)
  - Receives dynamic web page request
  - Server processes user input, renders the dynamic web page to be returned to the client for display on browser
- Client-Side Processing (e.g., JavaScript)
  - Processing needs to be “executed” by the browser to:
    - Complete the request for the dynamic page (e.g. valid user-input)
    - Create the dynamic web page
  - More responsive UI and lowers the bandwidth cost

Next....

HTML and CSS

Issac Carr

COMP 1531

Software Engineering Fundamentals

Week 04, Thursday

Introduction to Flask

Python Functions And Decorators

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

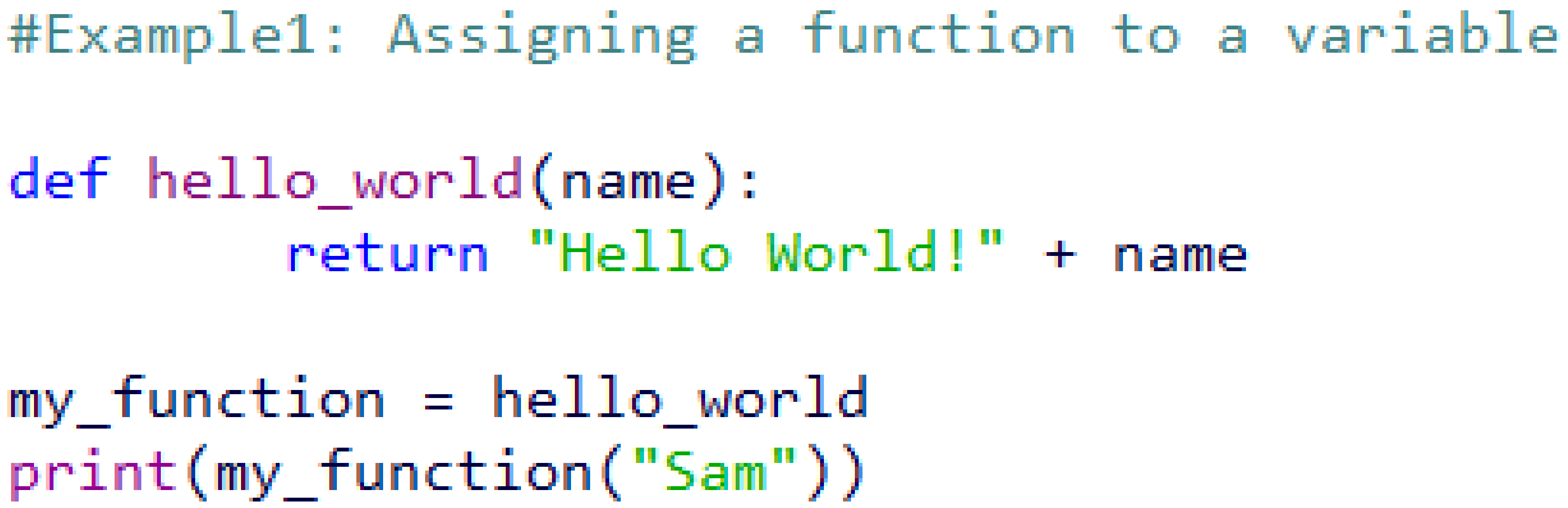
Python Function Decorators

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:

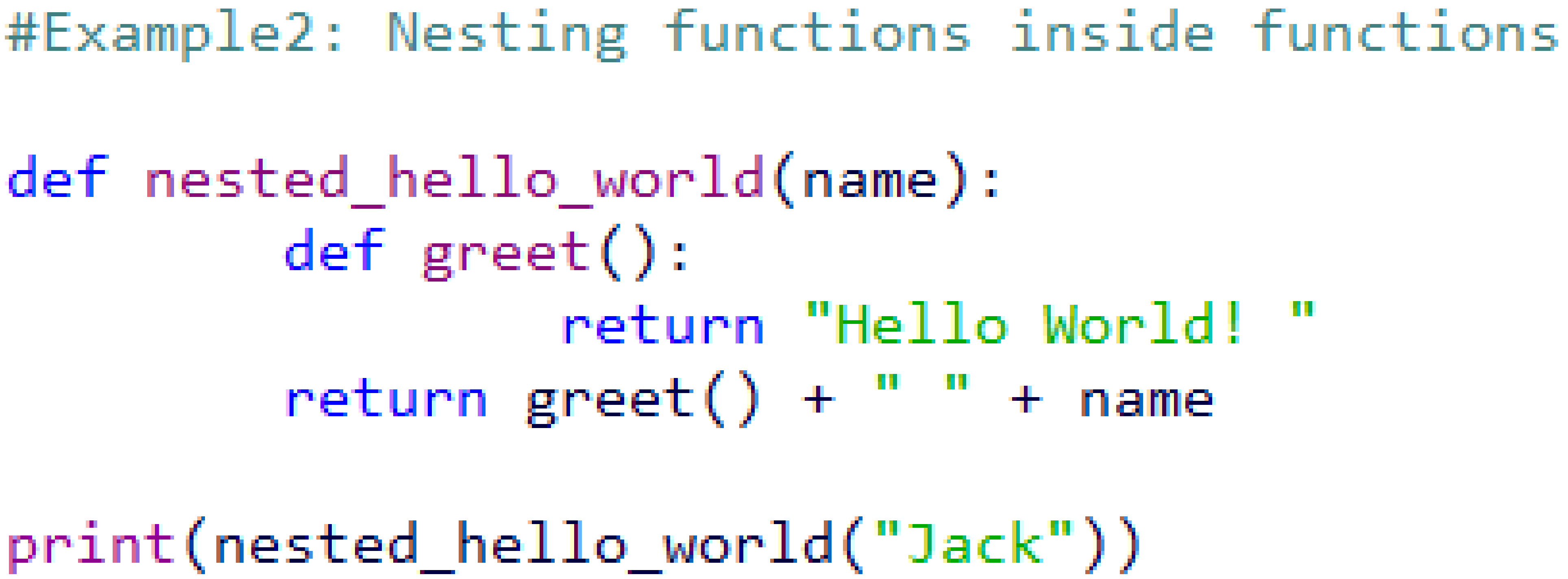
1. Assigns function to variables
2. Define functions inside another function
3. Functions can be passed as parameters to other functions
4. Functions can return other functions

Assign Function To Variable



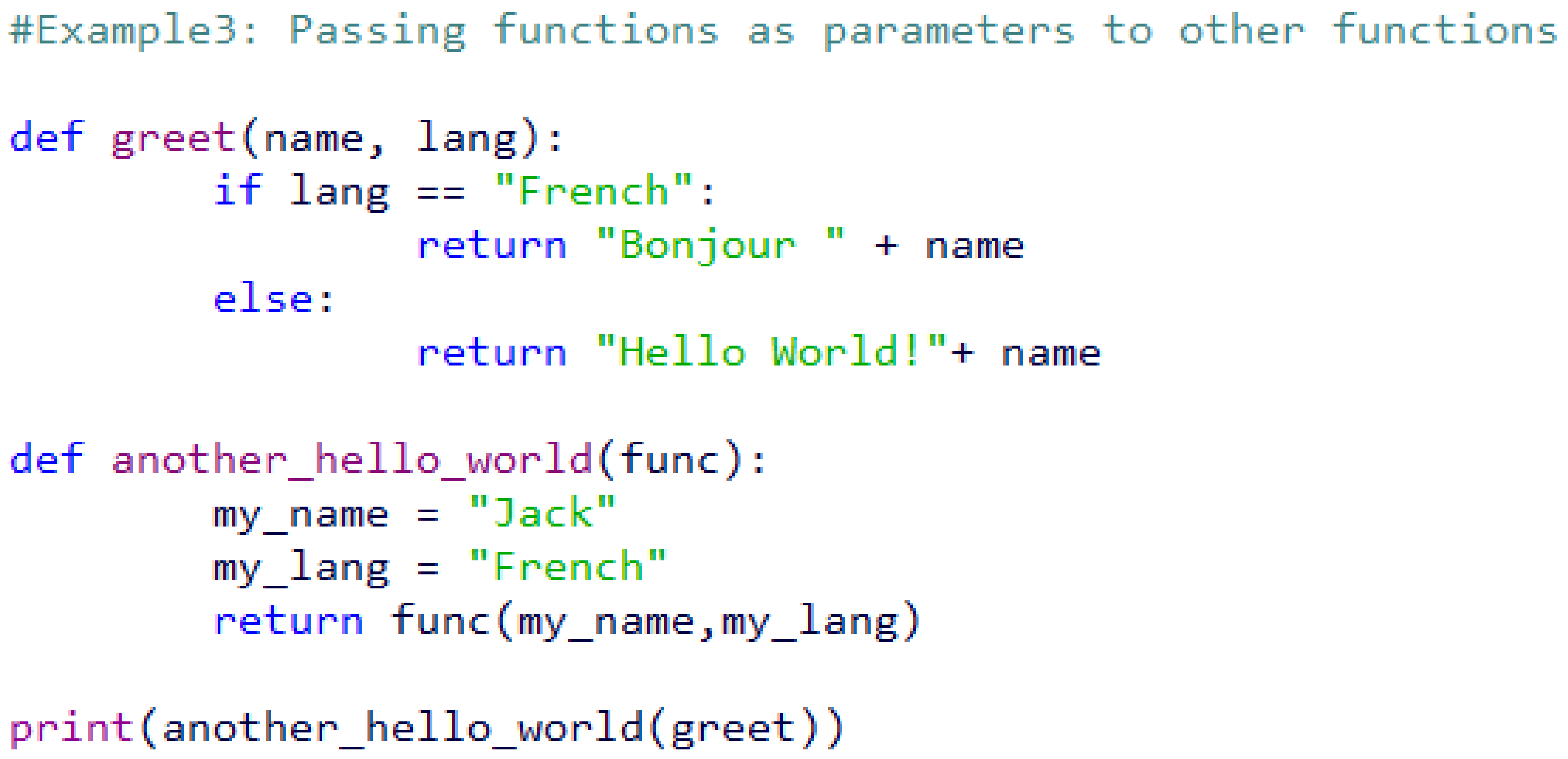
```
#Example1: Assigning a function to a variable

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

my_function = hello_world
print(my_function("Sam"))
```

Nesting Functions



```python
#Example2: Nesting functions inside functions

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

print(nested_hello_world("Jack"))
```

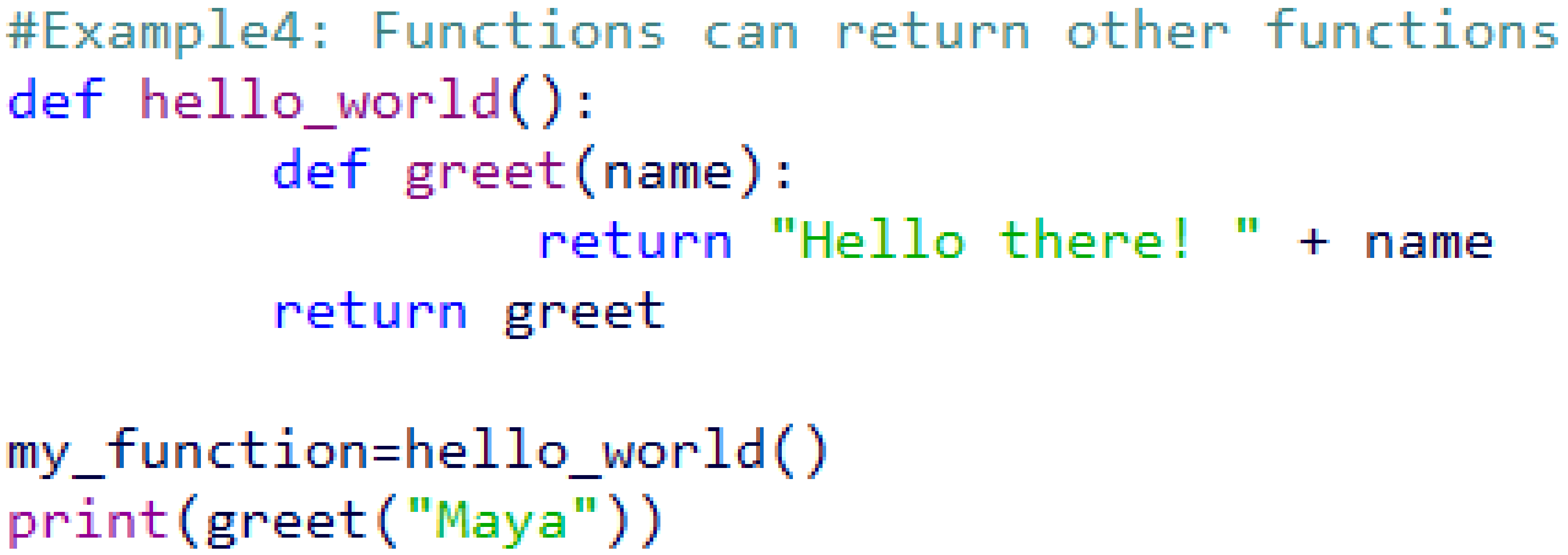
Functions Can Be Passed As Parameters To Other Functions



```python
#Example3: Passing functions as parameters to other functions

def greet(name, lang):
        if lang == "French":
                return "Bonjour " + name
        else:
                return "Hello World!"+ name

def another_hello_world(func):
        my_name = "Jack"
        my_lang = "French"
        return func(my_name,my_lang)

print(another_hello_world(greet))
```

This example returns the same result as invoking the function greet() directly

Functions can return other Functions



```
#Example4: Functions can return other functions
def hello_world():
        def greet(name):
                return "Hello there! " + name
        return greet

my_function=hello_world()
print(greet("Maya"))
```

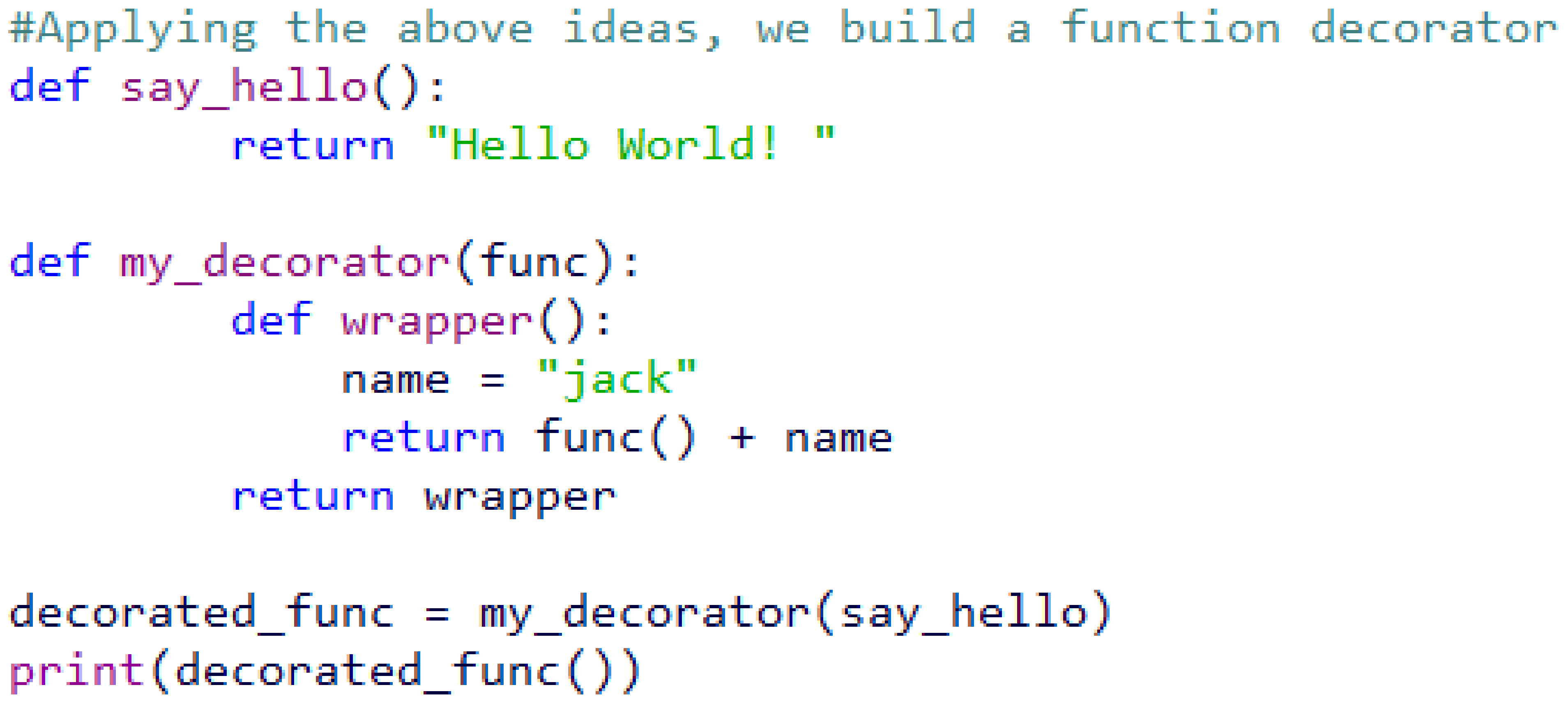
Composition Of Decorators

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

- Function decorators are simply wrappers to existing functions.
- They are useful for extending the behavior of functions without having to actually modify them

Composition Of Decorators

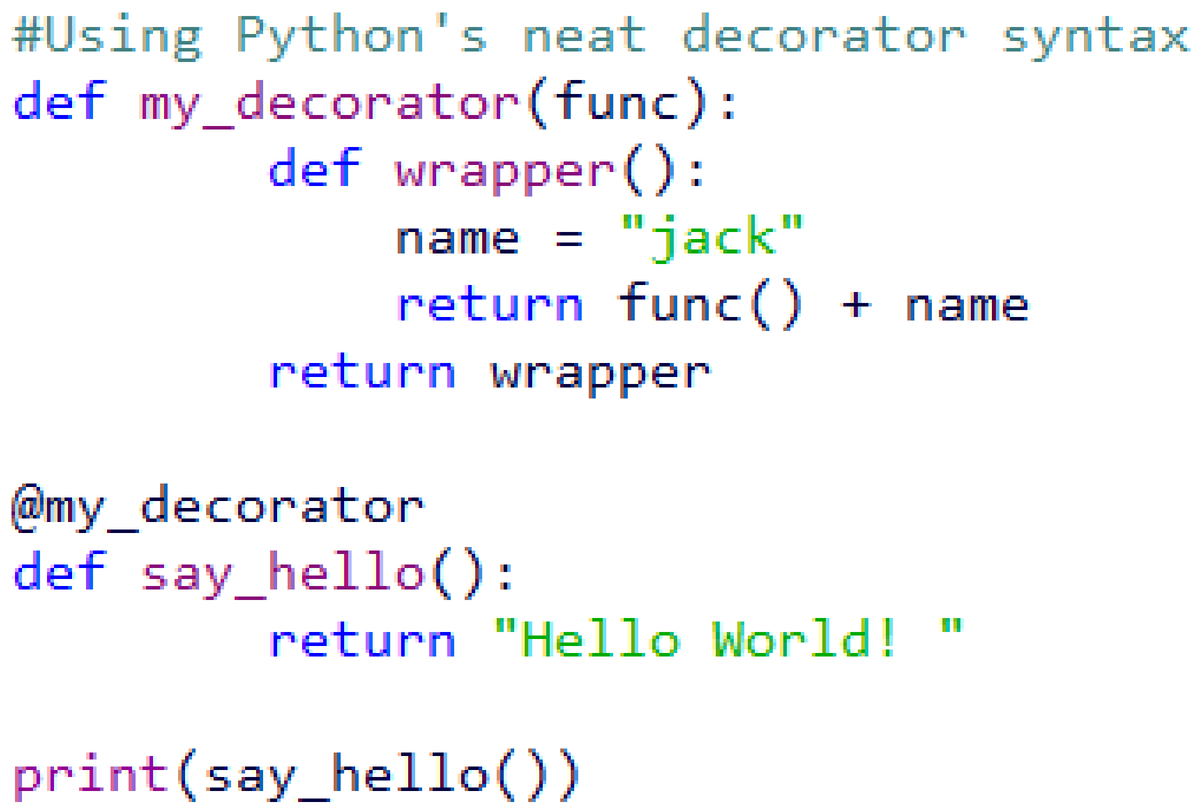
A decorator function basically takes a function as an argument, generates a new function that augments the work of the original function and returns the newly generated function



```python
#Applying the above ideas, we build a function decorator
def say_hello():
        return "Hello World! "

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

decorated_func = my_decorator(say_hello)
print(decorated_func())
```

Python's Decorator Syntax

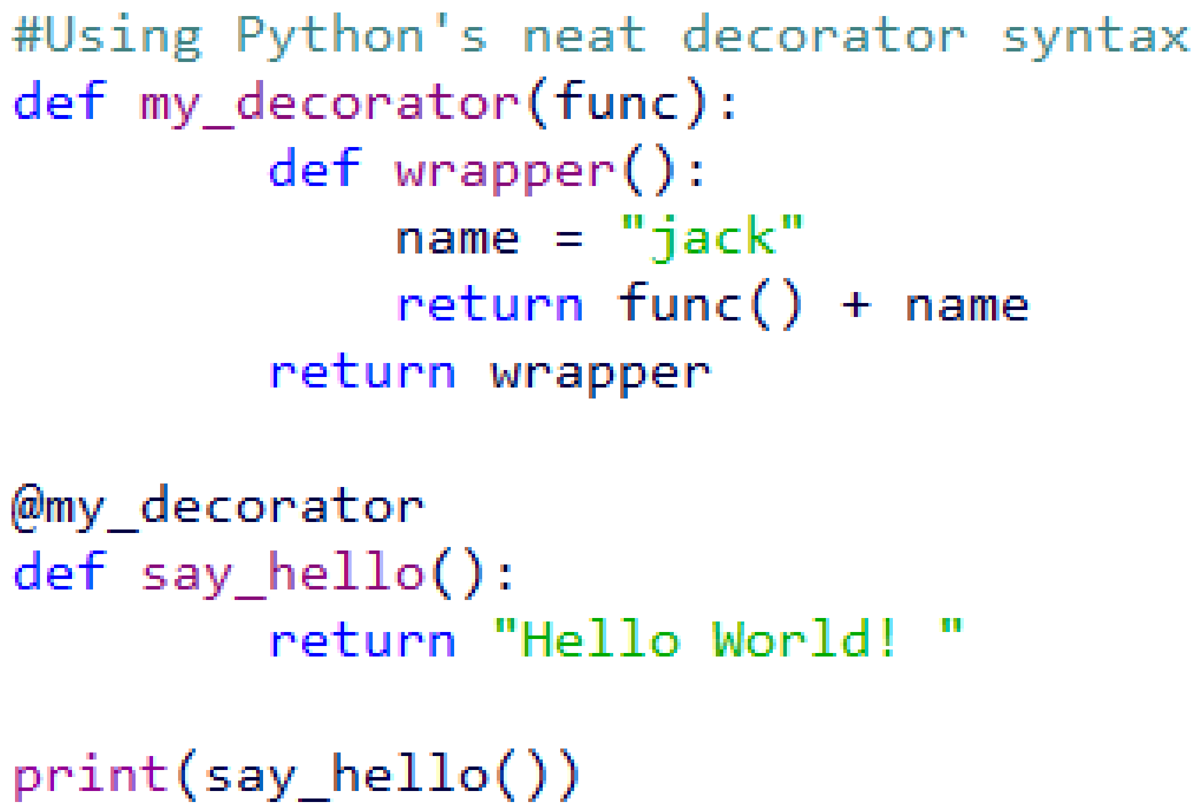
Provides a neater shortcut, by specifying the decorating function before the function to be decorated



```python
#Using Python's neat decorator syntax
def my_decorator(func):
        def wrapper():
            name = "jack"
            return func() + name
        return wrapper

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

print(say_hello())
```

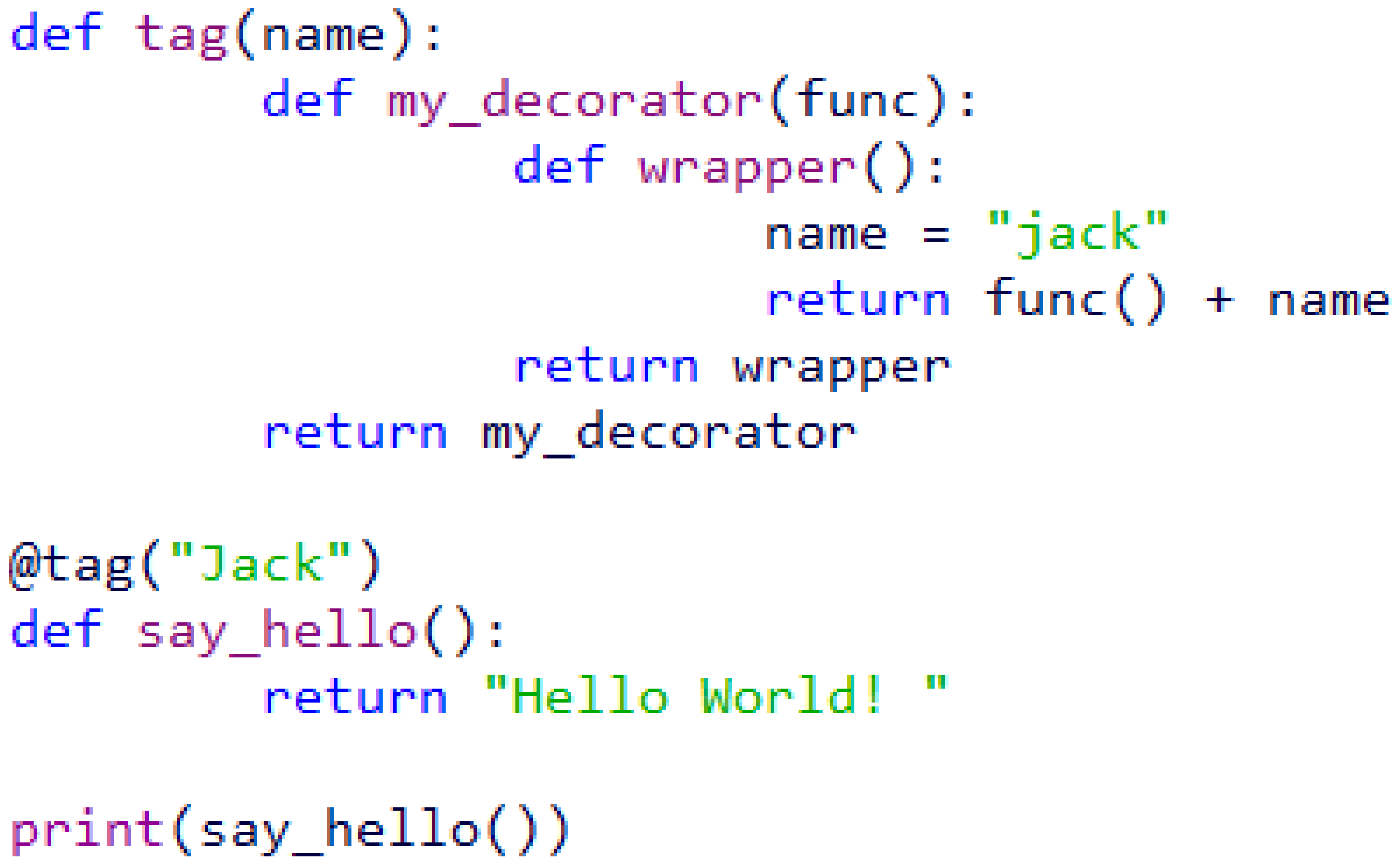
Passing arguments to Python’s decorator

Provides a neater shortcut, by specifying the decorating function before the function to be decorated


```
#Using Python's neat decorator syntax
def my_decorator(func):
        def wrapper():
            name = "jack"
            return func() + name
        return wrapper

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

print(say_hello())
```

Passing Argument To Decorator

Instead of hard-coding the variable **name**, it could actually be passed in as an argument to the decorator function e.g., the function **my_decorator** can be wrapped inside another function **tag**, which could take in the name argument



```python
def tag(name):
        def my_decorator(func):
                def wrapper():
                        name = "jack"
                        return func() + name
                return wrapper
        return my_decorator

@tag("Jack")
def say_hello():
        return "Hello World! "

print(say_hello())
```

FLASK

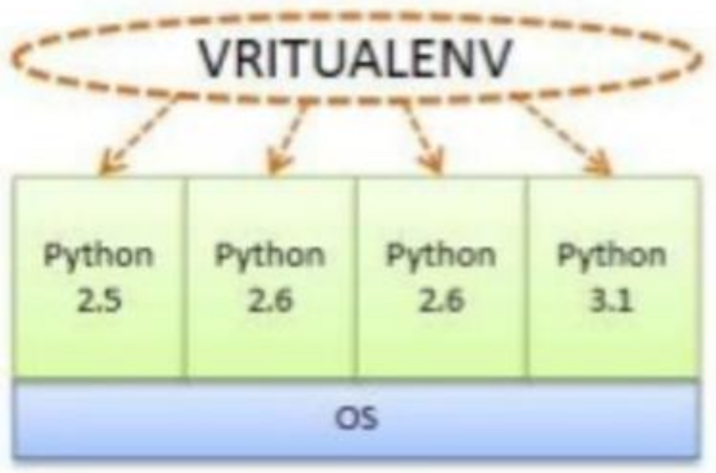
FLASK

- 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

Install flask using virtual environments

```
pip3 install virtualenv
virtualenv --version
```

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



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

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

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

A Simple Flask Application

```
# 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

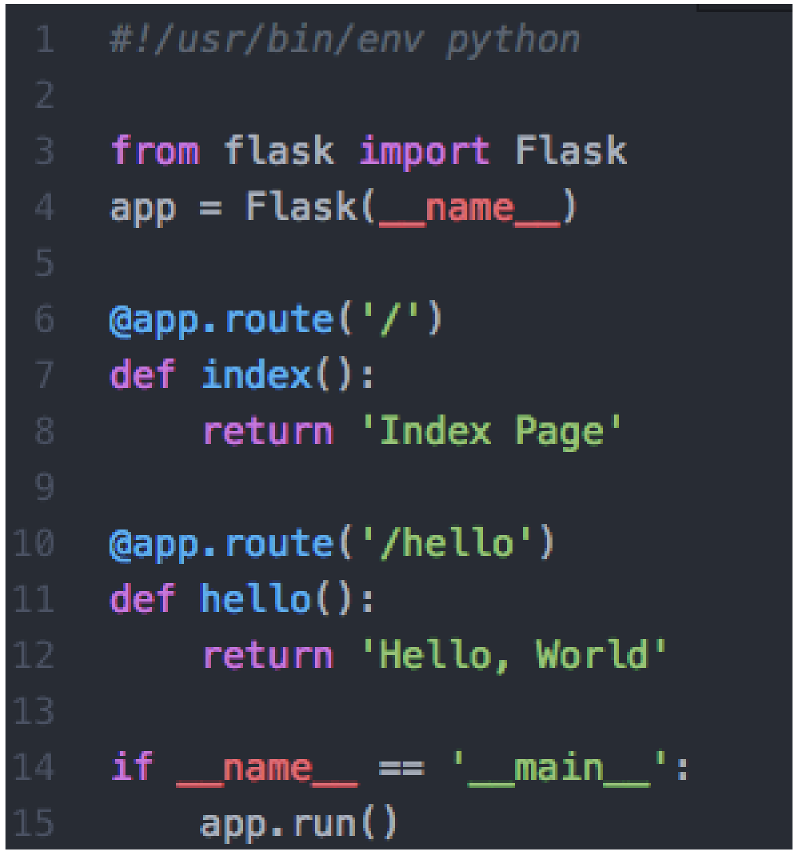
View function to render HTML page to browser

Running Flask Application

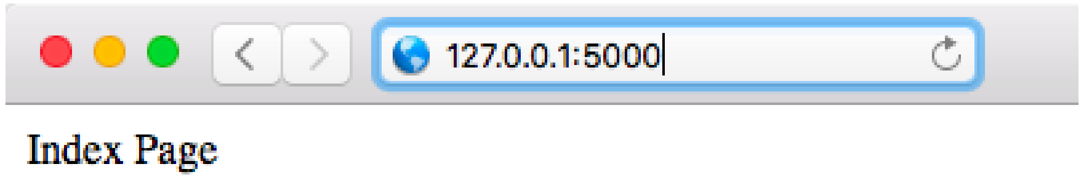
```
(hello) $ python hello.py
 * Restarting with stat
 * Debugger is active!
 * Debugger PIN: 281-144-438
 * Running on http://127.0.0.1:8085/ (Press CTRL+C to
quit)
127.0.0.1 - - [18/Jul/2017 09:10:19] "GET / HTTP/1.1"
200 -
127.0.0.1 - - [18/Jul/2017 09:10:19] "GET /favicon.ico
HTTP/1.1" 404 -
```

@app_route Function Decorator

The @app_route decorator or annotation is used to specify python functions to be executed for a specific URL path of the web application, i.e., to bind a function to an URL

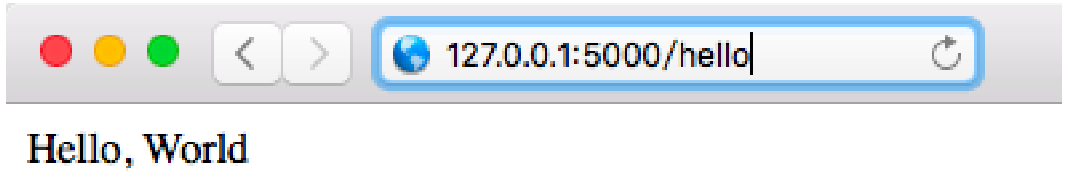


```python
#!/usr/bin/env python

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



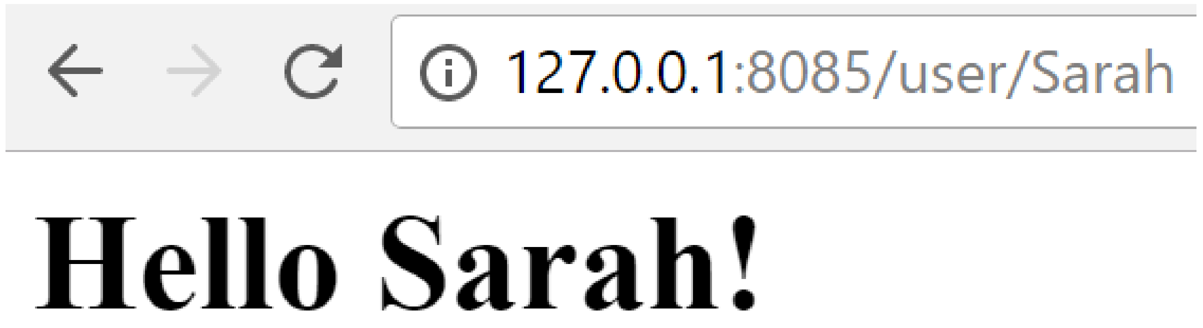
127.0.0.1:5000/hello

Hello, World

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!

- 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 maintain
- 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’}
```