Skipped in Fall 2020

[320] Web 2: Advanced Functions for Web Frameworks and Tracing

Tyler Caraza-Harter

Review Web

Differents servers on the same computer generally listen on different _____s.

If a process is listening for external traffic on port N, but clients cannot communicate, it's possible that a _____ is blocking port N.

Is it dangerous to run python3 -m http.server --bind=127.0.0.1 in a directory full of private data?

Why might a web browser need to fetch multiple resouces to load a page?

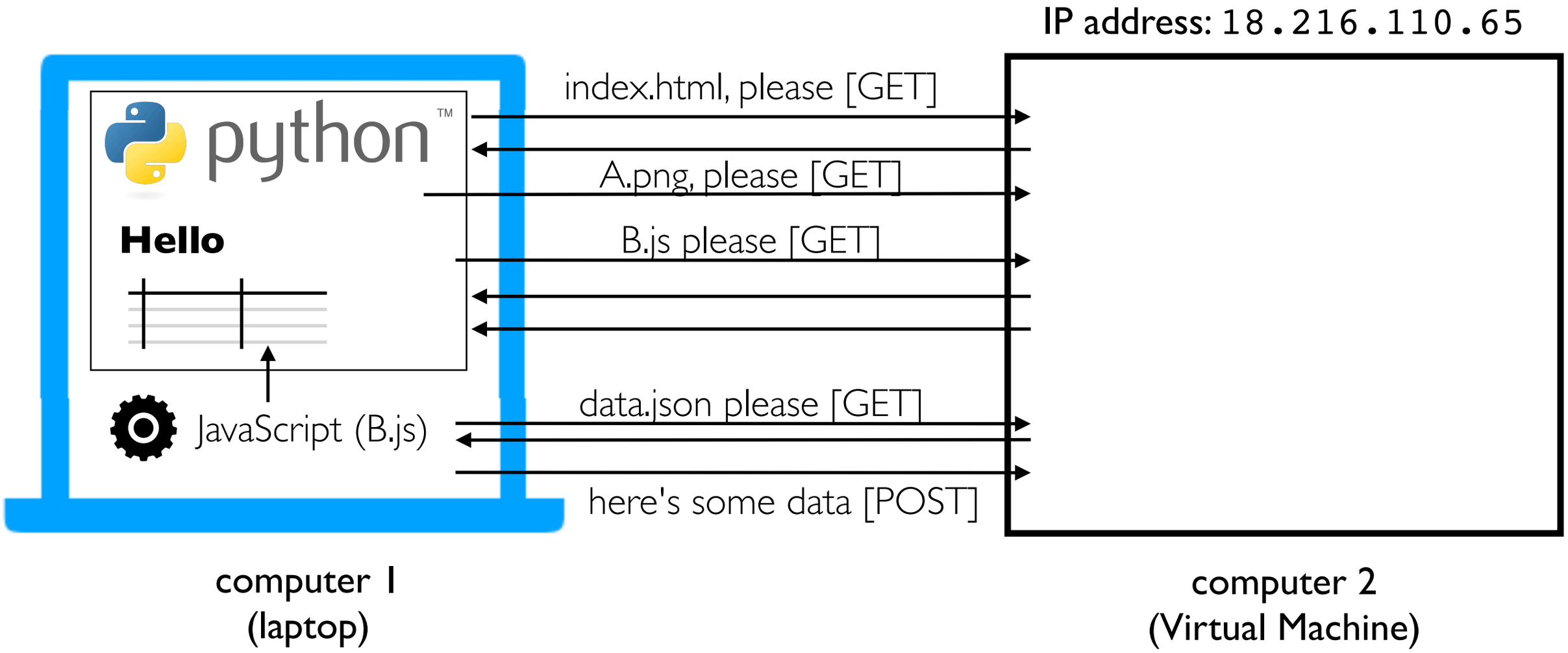
For the URL **http://123.123.123.123/about.html**, the resource **/about.html** refers to a **page**. For the URL **http://123.123.123.123/img/logo.png**, what is the resource?

A _____ page corresponds to the contents of a file.

A domain-name system (DNS) is like a dictionary, where you give it a domain name as a key, and you get back a _____ as a value.

Data may be uploaded with an HTTP _____ request.

Page Load, the Big Picture



It's hard to scrape this kind of table: requests.get("index.html") wouldn't work...

Sending Arguments to Server

HTTP GET:

resource

query string

```
GET /email.json?user=alice HTTP/1.1
Host: example.com
User-Agent: ...
Accept: */*
```

```
from flask import request

@app.route('/email.json')
def handler():
    name=request.args.get("user")
```

HTTP POST:

resource

```
POST /email.json HTTP/1.1
Host: example.com
User-Agent: ...
Content-Type: application/json
Content-Length: 14
Accept: */*

alice@wisc.edu
```

request body

```
@app.route('/email.json',
           methods=["POST"])
def handler():
    email=str(request.data, "utf-8")
```

Function References and Decorators

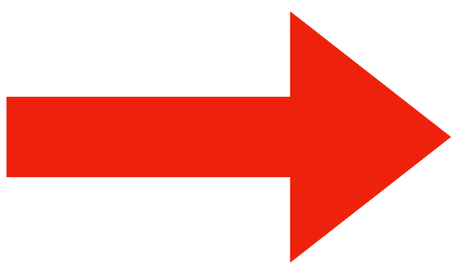
What does "@____" mean???

```python
import pandas as pd
from flask import Flask, request, jsonify

app = Flask(__name__)
# df = pd.read_csv("main.csv")

@app.route('/')
def home():
    with open("index.html") as f:
        html = f.read()

    return html

if __name__ == '__main__':
    app.run(host="0.0.0.0") # don't change this line!
```

decorator

https://github.com/tylerharter/cs320/tree/master/s20/p3

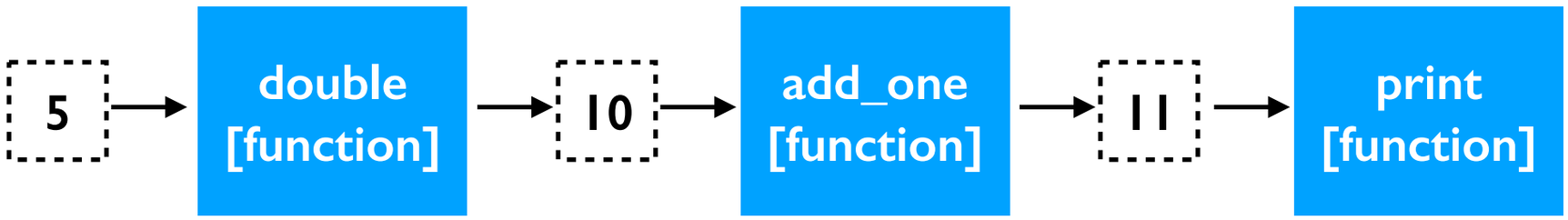
```
@f
def g():
  ...
```



f is a decorator, meaning:
it is a function that takes a reference to another function and returns a reference to a third function

Combine simple functions to achieve complex goals

Composition:



Calls:

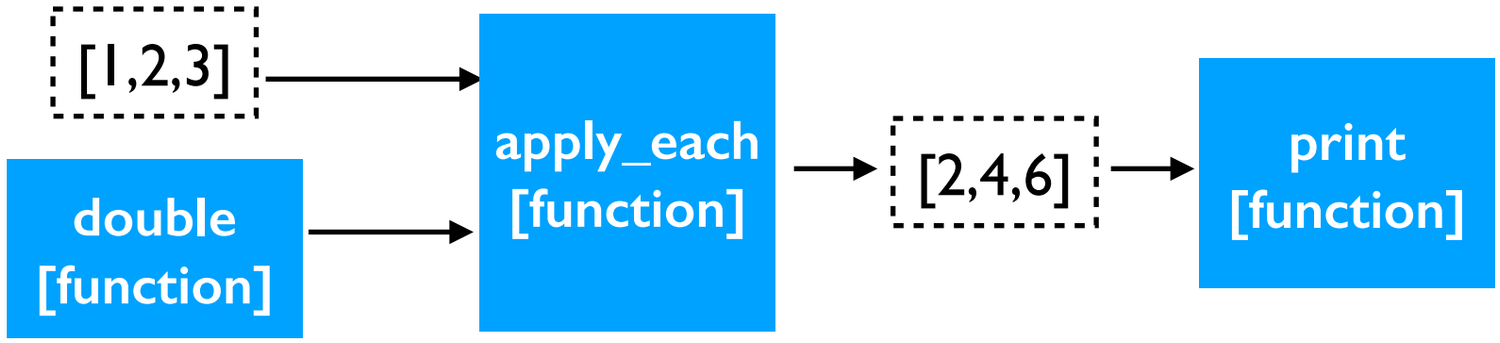
```
print(add_one(double(5)))
```

Definitions:

```
def double(x):
    return x * 2

def add_one(x):
    return x + 1
```

Combine simple functions to achieve complex goals

Passing Function Reference:



Calls:

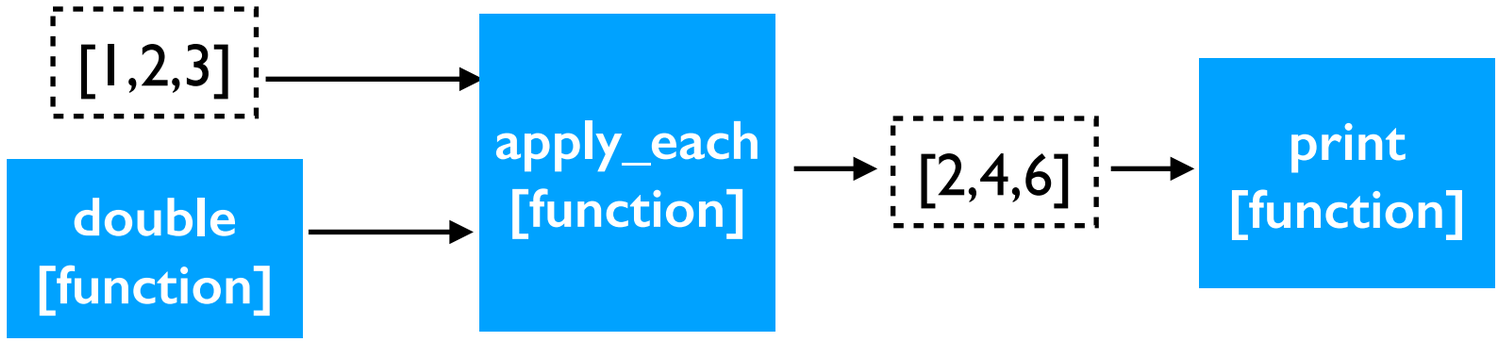
```
print(apply_each([1,2,3], double))
```

Definitions:

```
def double(x):
    return x * 2

def apply_each(nums, fn):
    return [fn(x) for x in nums]
```

Combine simple functions to achieve complex goals

Passing Function Reference:



Calls:

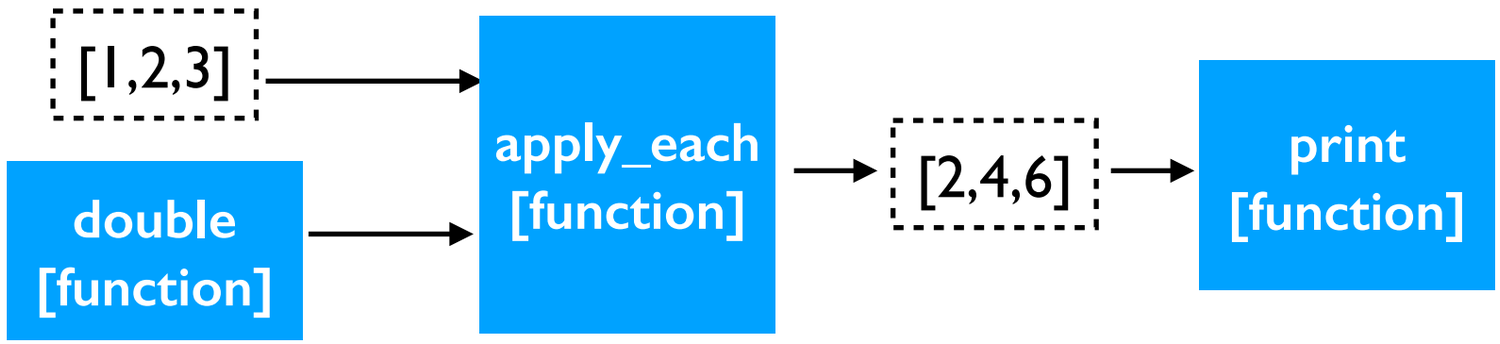
```
print(apply_each([1,2,3], double))
```

Definitions:

```
def double(x):
    return x * 2
double = lambda x: x * 2   # same as def double(x)...

def apply_each(nums, fn):
    return [fn(x) for x in nums]
```

Combine simple functions to achieve complex goals

Passing Function Reference:



Calls:

```
print(apply_each([1,2,3], double))
```

Definitions:

```
def double(x):
    return x * 2
```

```
double = lambda x: x * 2
```

reference to the function

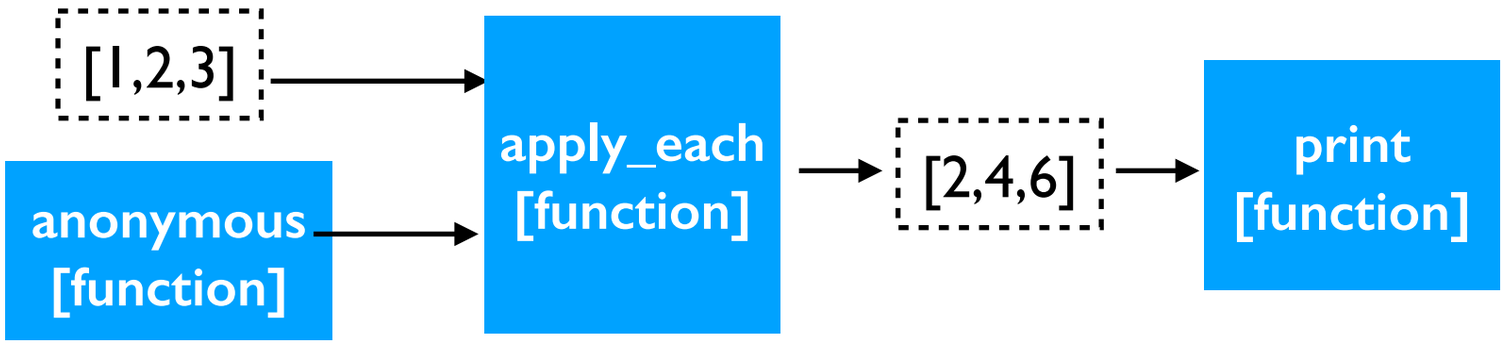
parameter

return value

```
def apply_each(nums, fn):
    return [fn(x) for x in nums]
```

Combine simple functions to achieve complex goals

Passing Function Reference:



Calls:

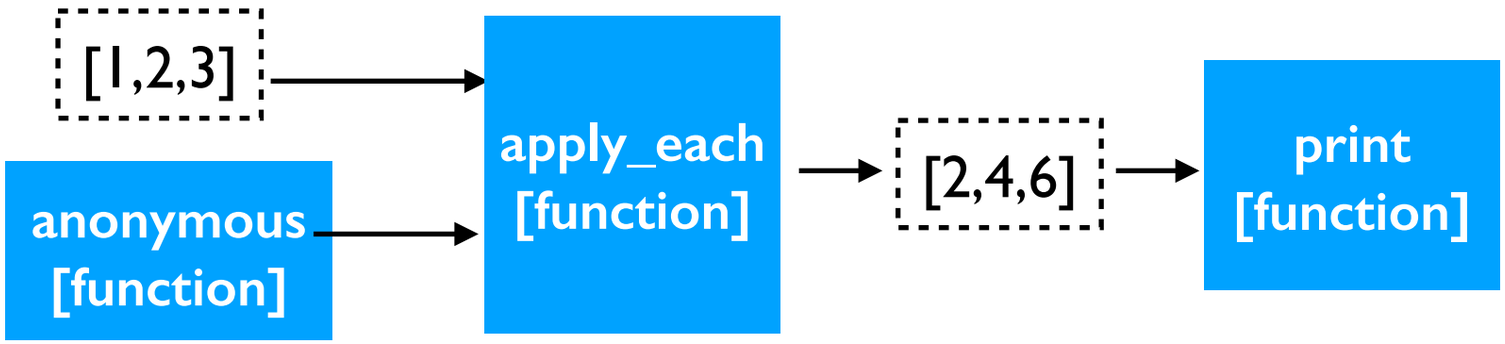
```
print(apply_each([1,2,3], double))
```

Definitions:

```
def double(x):
    return x * 2
double = lambda x: x * 2

def apply_each(nums, fn):
    return [fn(x) for x in nums]
```

Combine simple functions to achieve complex goals

Passing Function Reference:



Calls:

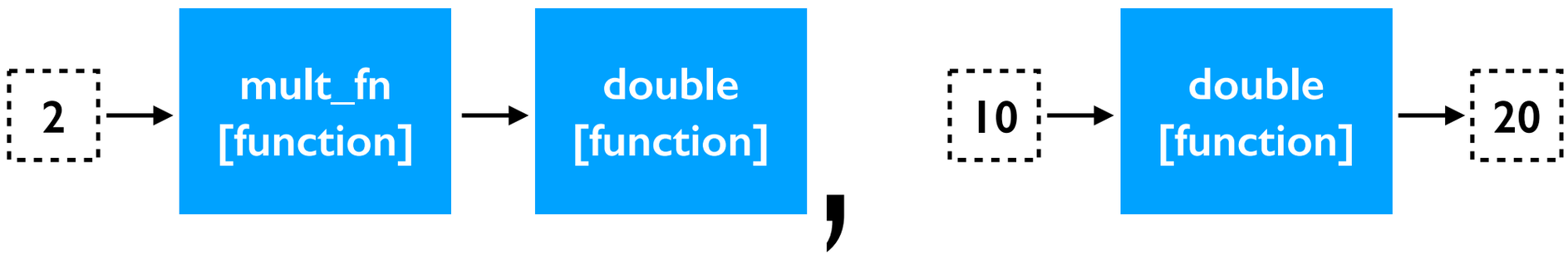
```
print(apply_each([1,2,3], lambda x: x * 2))
```

Definitions:

```
def double(x):
    return x * 2
double = lambda x: x * 2

def apply_each(nums, fn):
    return [fn(x) for x in nums]
```

Combine simple functions to achieve complex goals

Return Function Reference:



Calls:

```
double = mult_fn(2)
triple = mult_fn(3)
y = double(10)
```

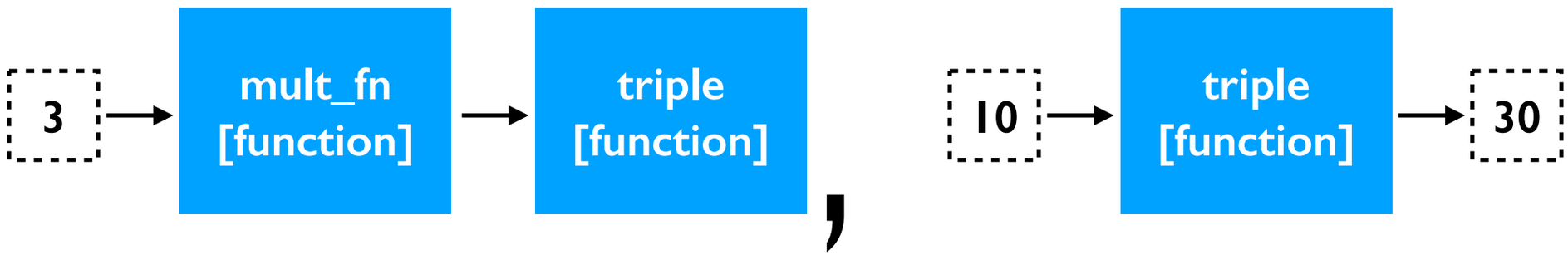
Definitions:

```
def mult_fn(num):
    def multiplier(x):
        return x * num
    return multiplier
```

PythonTutor

Combine simple functions to achieve complex goals

Return Function Reference:



Calls:

```
double = mult_fn(2)
triple = mult_fn(3)
y = triple(10)
```

Definitions:

```
def mult_fn(num):
    def multiplier(x):
        return x * num
    return multiplier
```

PythonTutor

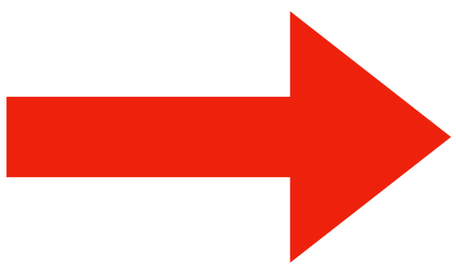
What does "@____" mean???

```python
import pandas as pd
from flask import Flask, request, jsonify

app = Flask(__name__)
# df = pd.read_csv("main.csv")

@app.route('/')
def home():
    with open("index.html") as f:
        html = f.read()

    return html

if __name__ == '__main__':
    app.run(host="0.0.0.0") # don't change this line!
```

decorator

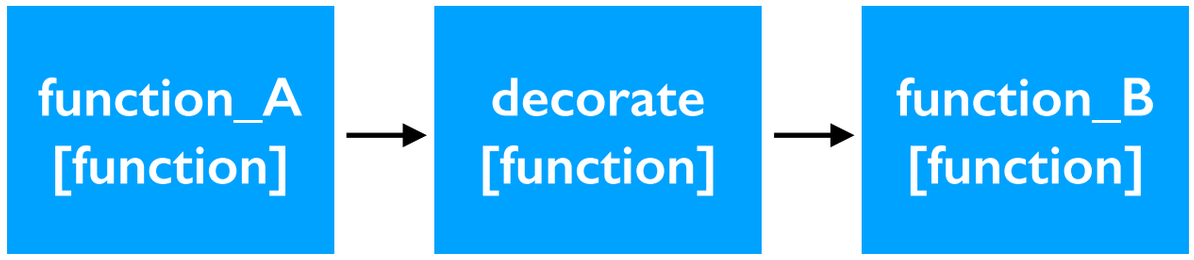
https://github.com/tylerharter/cs320/tree/master/s20/p3

```
@f
def g():
  ...
```



f is a decorator, meaning:
it is a function that takes a reference to another function and returns a reference to a third function

Decorator: Mechanics

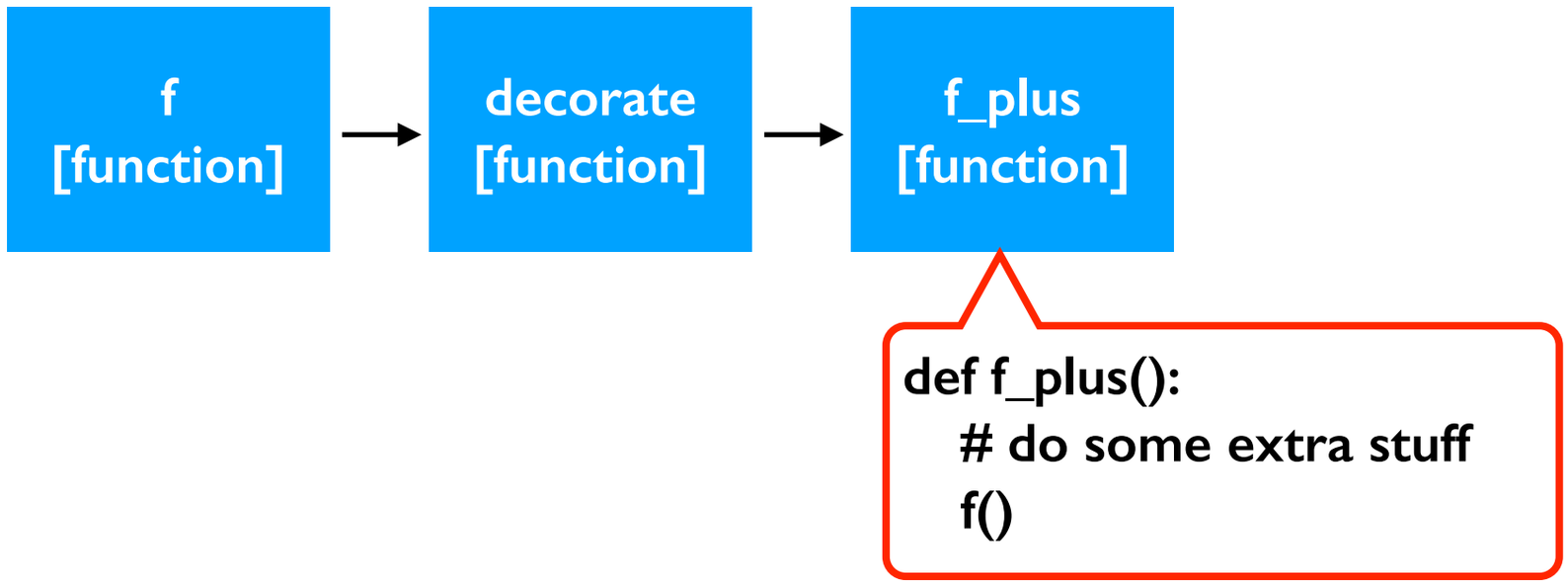



```
def function_A():
    print("A")

def decorate(fn):
    print("decorating!")
    return function_A

def function_B():
    print("B")
function_B = decorate(function_B)

function_B() # prints "A"!
```

```
def function_A():
    print("A")

def decorate(fn):
    print("decorating!")
    return function_A

@decorate
def function_B():
    print("B")

function_B() # prints "A"!
```

PythonTutor

Decorator Pattern 1: wrapper



Decorator Pattern 1: wrapper

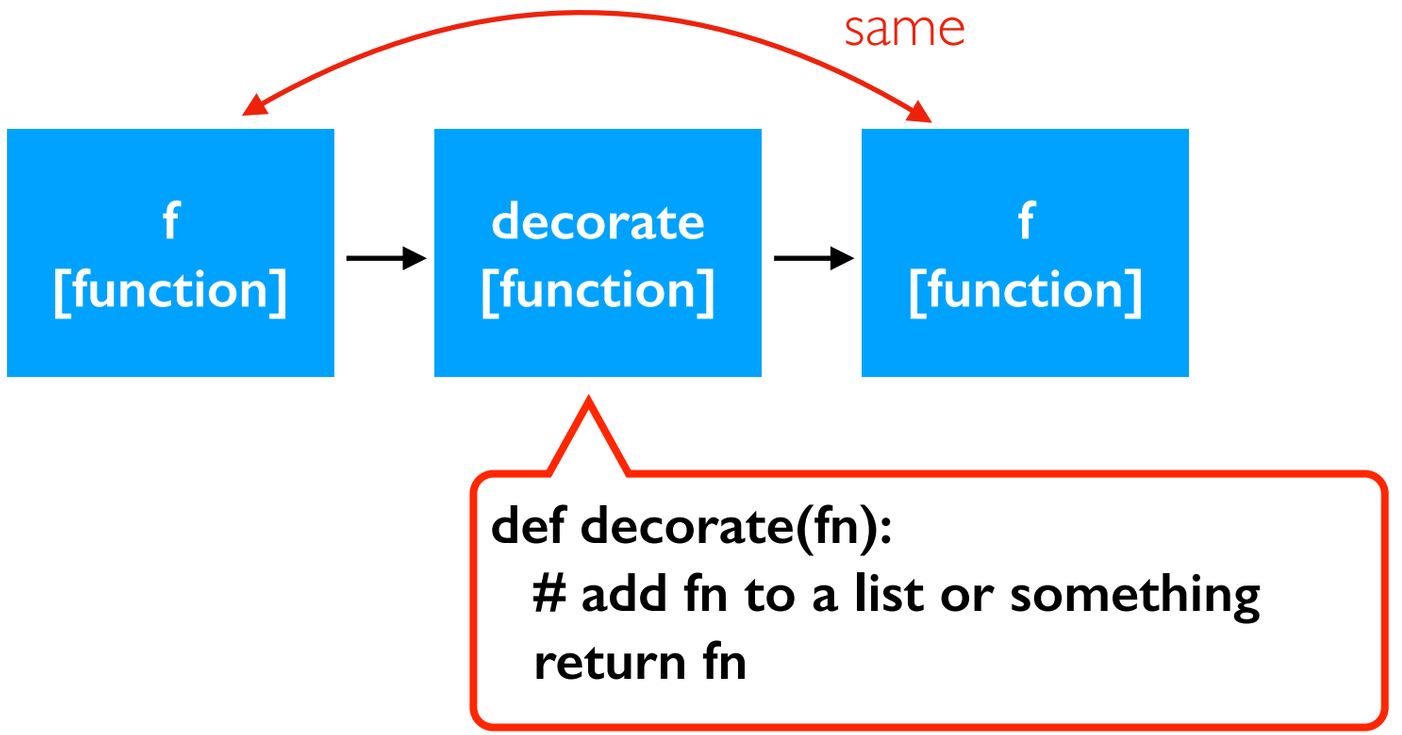

```
counts = {}

def count_me(fn):
    counts[fn.__name__] = 0
    def wrapper():
        counts[fn.__name__] += 1
        fn()
    return wrapper

@count_me
def f():
    print("f")

@count_me
def g():
    print("g")
```

```
def f_plus():
    # do some extra stuff
    f()
```

example: track how often each function is called

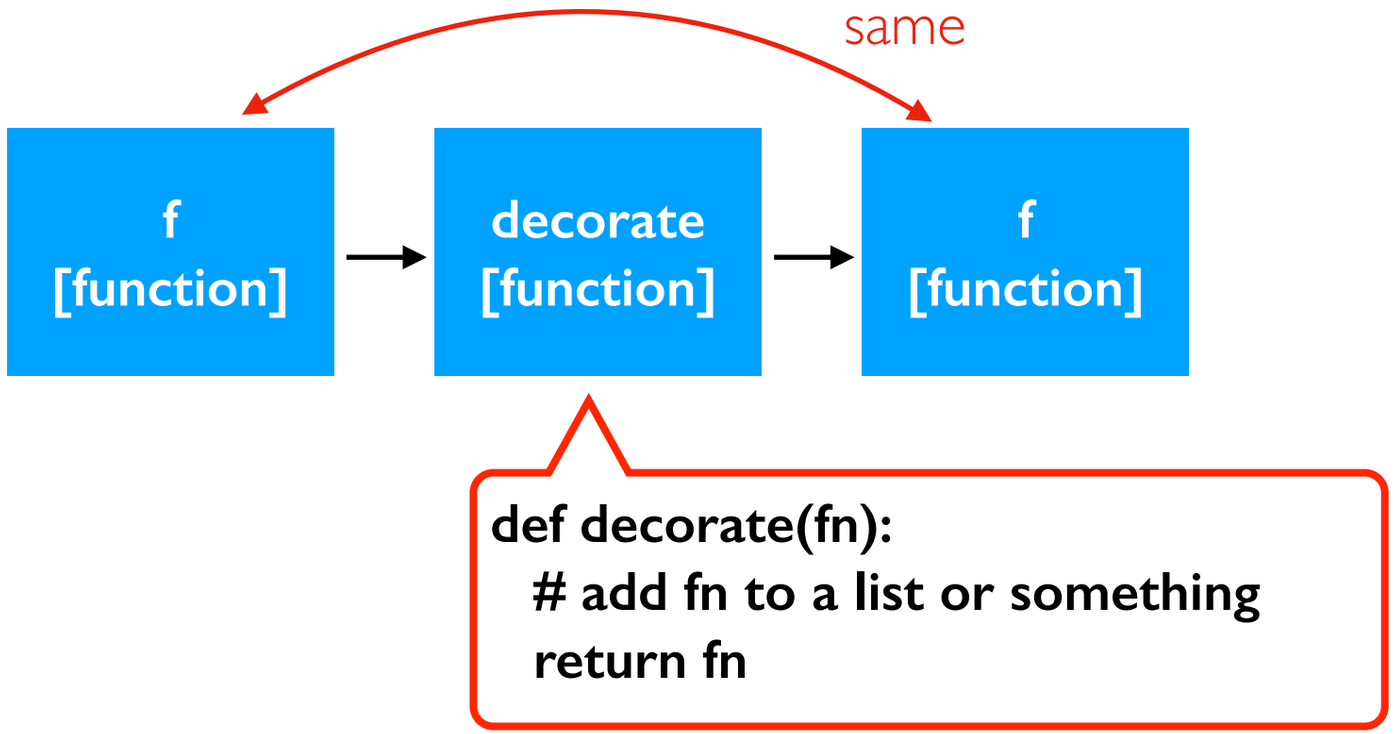
PythonTutor

Decorator Pattern 2: register



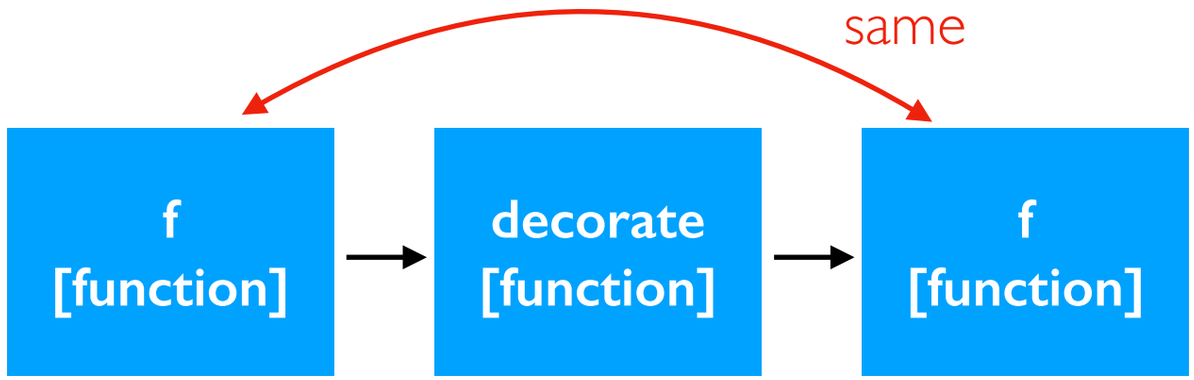
Decorator Pattern 2: register

```
def abs(x):
    if x < 0:
        return -x
    elif x > 0:
        return x

tests = []
def test(fn):
    tests.append(fn)
    return fn

@test
def test_neg():
    assert abs(-1) == 1
    assert abs(-3) == 3

@test
def test_pos():
    assert abs(1) == 1
    assert abs(3) == 3

@test
def test_zero():
    assert abs(0) == 0

passing = 0
failing = 0
for test_fn in tests:
    try:
        test_fn()
        passing += 1
    except Exception:
        failing += 1
print("PASS", passing, "FAIL", failing)
```



PythonTutor

Decorator Pattern 2: register




```
routes = {}

def route(url):
    def wrap(fn):
        routes[url] = fn
        return fn
    return wrap

@route("/")
def home():
    print("home")

@route("/donate.html")
def page2():
    print("donate")

for resource in ["/", "/donate.html", "missing.html"]:
    fn = routes.get(resource)
    if fn == None:
        print("404!")
        continue
    fn()
```

def decorate(fn):
 # add fn to a list or something
return fn

PythonTutor

Register home function to handle "/" requests

```
import pandas as pd
from flask import Flask, request, jsonify

app = Flask(__name__)
# df = pd.read_csv("main.csv")

@app.route('/')
def home():
    with open("index.html") as f:
        html = f.read()

    return html

if __name__ == '__main__':
    app.run(host="0.0.0.0") # don't change this line!
```

decorator

https://github.com/tylerharter/cs320/tree/master/s20/p3

Variable Length Arguments

*args

```
s = "Dear {}, you are invited to {}."

print(s.format(????))
```

how many arguments should go here?

*args

```
s = "Dear {}, you are invited to {}."

print(s.format("Student", "hackathon"))
```

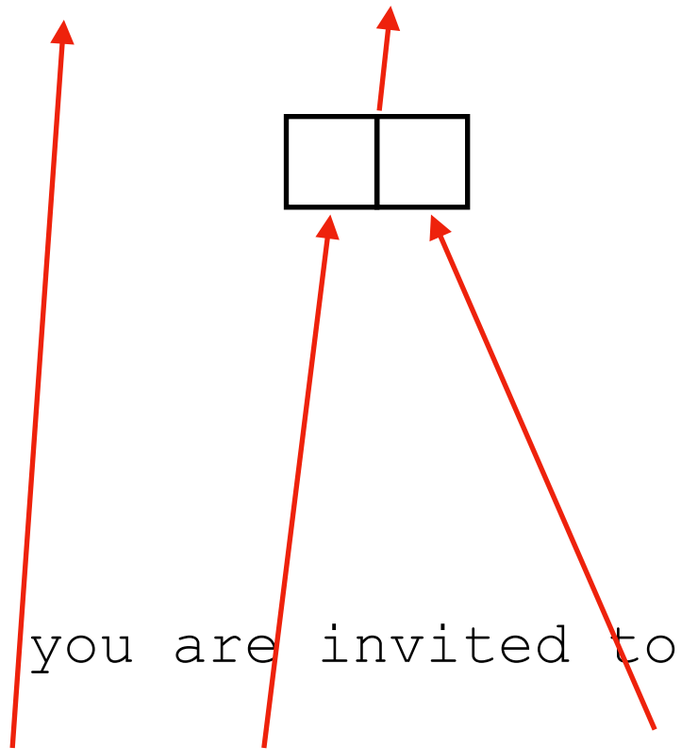
*args

```
def format(template, *args):
    ...
```

```
s = "Dear {}, you are invited to {}."

print(format(s, "Student", "hackathon"))
```

*args

```
def format(template, *args):
    ...

s = "Dear {}, you are invited to {}."

print(format(s, "Student", "hackathon"))
```

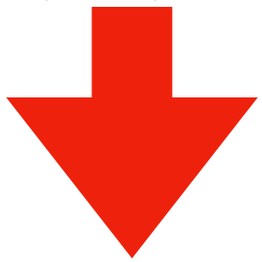


*args

```
def format(template, *args):
    parts = template.split("{}")
    assert(len(parts) == len(args) + 1)
    result = []
    for i in range(len(args)):
        result.append(parts[i])
        result.append(args[i])
    result.append(parts[-1])
    return "".join(result)

s = "Dear {}, you are invited to {}."

print(format(s, "Student", "hackathon"))
```

Star (*) can be used on both parameter and argument sides

```
print(1, 2, 3)
```



```
print(*[1, 2, 3])
```

Double star (**) can be for keyword arguments

```
def f(*args, **kwargs):
    print("ARGS", args)
    print("KWARGS", kwargs)

f(1, 2, x=3, y=4, z=5)
```


```
ARGS (1, 2)
KWARGS {'x': 3, 'y': 4, 'z': 5}
```

Tracing

Tracing

What if we want a record/log/trace of every function invocation, and the arguments?

Use decorators to wrap the function of interest.

Use *args and **kwargs to capture any inputs.

```
def trace(fn):
    def wrap(*args, **kwargs):
        print("CALL {}(*{}, **{})".format(fn.__name__, args, kwargs))
        return fn(*args, **kwargs)
    return wrap

@trace
def add(x, y):
    return x+y

@trace
def mult(x, y):
    return x*y

print(add(1, 2))
print(add(x=1, y=2))
print(mult(2, y=3))
```

Query Strings and Post Bodies

[code examples]