Functions & Functional Programming

May 15, 2023

Final Project

~~Final Project Proposal~~	~~End of Week 5~~
Meetings with Parth/Tara	**Weeks 7/8**
Presentations	Week 10 (May 31)

Announcements

1. Assignment 1 feedback released tonight
2. Transition to working on the final project — possibly an extension of assignment 2
3. On the horizon: final project pizza party 🥹

Lecture 6: Low-level Python

Lecture 7: Functions & Functional Programming

Lecture 8: Standard & Third-Party Libraries

Lecture 9: Gaming

Lecture 10: Final Project Presentations

Deep dive into Python's implementation, features

Shallow dive into Python's implementation, features

preparing for the final presentations

Learning Objectives

After today's lecture, students will be able to...

- Write generic functions that accept a variable number of arguments
- Handle (possibly infinite) streams of data in Python
- Create functions that modify the behavior of other functions

Agenda

1. Assignment 2 show & tell
2. Variadic arguments
3. Functions as objects and decorators
4. Generators and infinite data

Show & Tell

[1-2 minutes] Introduction

- What are your names?
- What did you build?

[1 minute] Reflections

- Why did you build it?
- What would you do differently?

[1 minute] Q&A

Variadic Arguments

We've seen this...

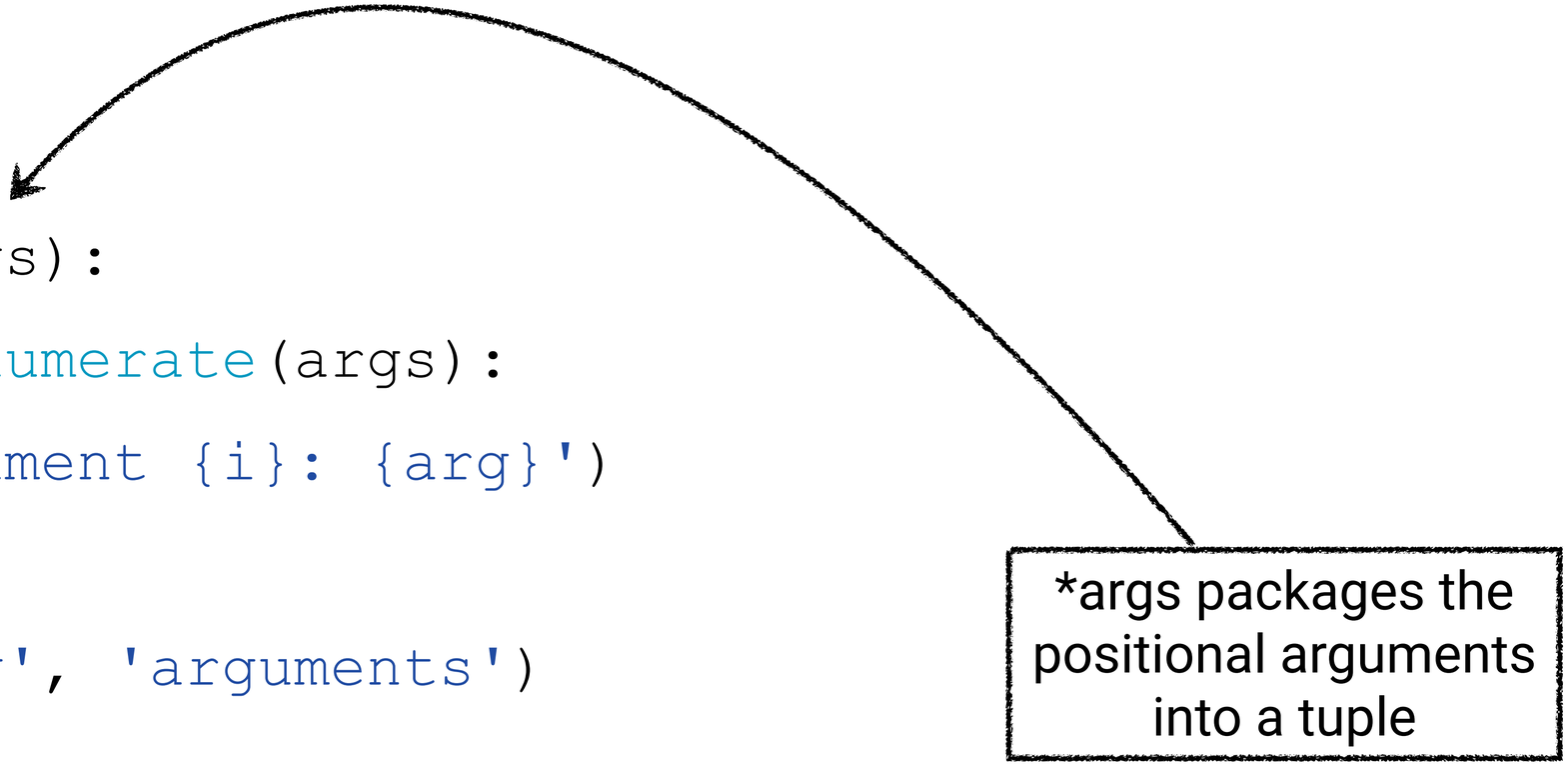
```
def get_image(url, filename, raise_error=False):
    r = requests.get(url)

    if r.ok:
        open(filename, 'wb').write(r.content)
    else:
        if raise_error:
            raise ValueError(f'Calling requests.get("{url}") failed')
```

```
get_image("https://...", "img/tara.png")
get_image("https://...", "img/tara.png", True)
get_image("https://...", filename="img/tara.png", raise_error=True)
```

Case Study: `print`

```
print(1)
# => 1

print(1, 2, 3)
# => 1 2 3

print(1, 2, 'many', 'arguments')
# => 1 2 many arguments
```

print accepts an arbitrary number of arguments, of any type

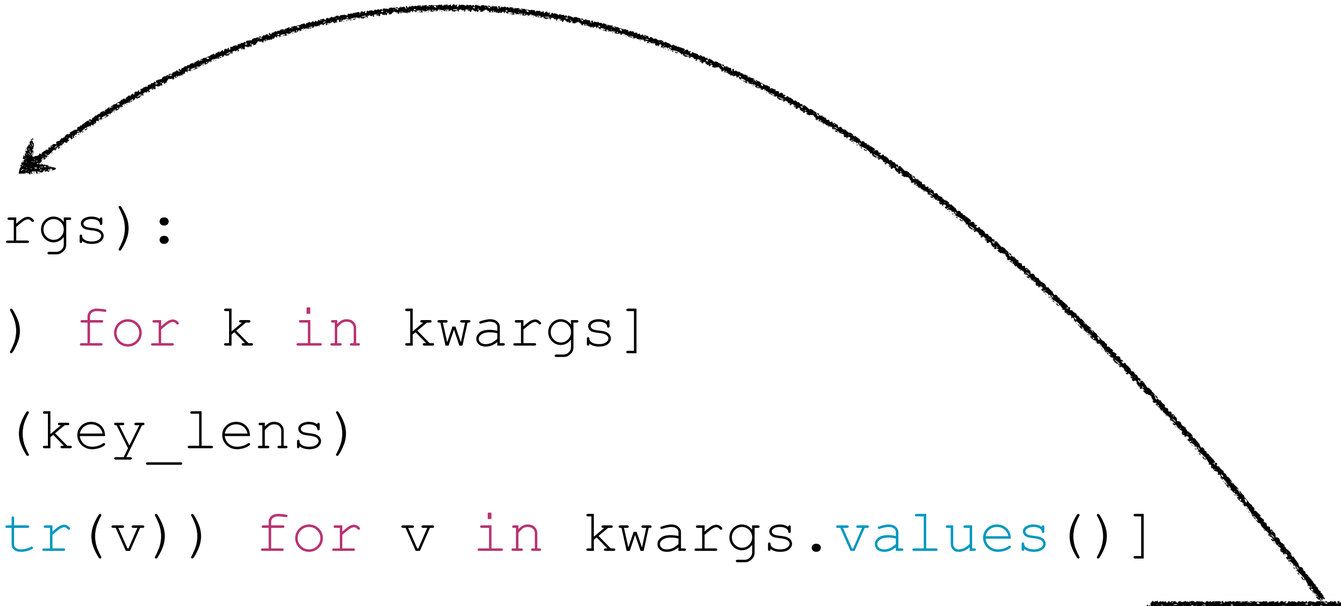



```
def my_function(*args):
    for i, arg in enumerate(args):
        print(f'Argument {i}: {arg}')


my_function(2, 'many', 'arguments')
# Argument 0: 2
# Argument 1: many
# Argument 2: arguments
```

Case Study: `print`

```
print(1, 2, 3, sep=', ')
# 1, 2, 3

print(1, 2, 3, sep=', ', end='\n---\n')
# 1, 2, 3
# ---
```

print accepts an arbitrary number of arguments **and** optional keyword arguments



```
def print_table(**kwargs):
    key_lens = [len(k) for k in kwargs]
    max_key_len = max(key_lens)
    val_lens = [len(str(v)) for v in kwargs.values()]
    max_val_len = max(val_lens) + 1

    print('-' * (max_key_len + max_val_len + 3))
    for left, right in kwargs.items():
        print(f"{left:{max_key_len}} | {right}")
        print('-' * (max_key_len + max_val_len + 3))
```

**kwargs packages the keyword arguments into a dictionary

```
print_table(name='Arpit Ranasaria', title='TA', course='CS 41')
# -------------------------
# name   | Arpit Ranasaria
# -------------------------
# title  | TA
# -------------------------
# course | CS 41
# -------------------------

print_table(artist='The Killers', title='Mr. Brightside',
on_41_playlist=True, added_by_will=True)
# -------------------------------
# artist          | The Killers
# -------------------------------
# title           | Mr. Brightside
# -------------------------------
# on_41_playlist | True
# -------------------------------
# added_by_will  | True
# -------------------------------
```

```
def generic_fn(*args, **kwargs):
    ...
```

*args packages the positional arguments into a tuple

**kwargs packages the keyword arguments into a dictionary

```
generic_fn(1, 2, buckle_my='shoe')
```

```
generic_fn(
  True, 3, '!', arg=False,
  another_arg={'some': 'things'}
)
```

this function can be called with any (valid) collection of parameters

Turn & Talk: Why might this be useful?

Functions as objects and decorators

```
def my_function(a, b, c):
    ...

type(my_function)
# => function

hex(id(my_function))
# => '0x10680f880'

print(my_function)
# => <function my_function at 0x10680f880>
```

functions are objects!

Some things we can do with objects...

Assign variables referencing them...

```
def add(a, b):
    return a + b


combine = add
combine(1, 2) # => 3
```

Pass them as arguments...

```
sorted(['parth', 'tara', 'will', 'arpit'], key=len)
# => ['tara', 'will', 'parth', 'arpit']
```

Some things we can do with objects...

Return them from other functions...

```
def is_multiple_of(n):
    def test(x):
        return n % x == 0

    return test
```

Taking stock...

Now, we can...

- Write generic functions (which accept an arbitrary number of positional or keyword arguments)
- Pass functions as arguments into other functions
- Return a function from a function

Turn & Talk: Why might this be useful?

Decorators!

```
def print_args(func):
    def modified_func(*args, **kwargs):
        print(f'Positional args: {args}')
        print(f'Keyword args: {kwargs}')
        return func(*args, **kwargs)

    return modified_func
```

```
def print_args(func):
    def modified_func(*args, **kwargs):
        print(f'Positional args: {args}')
        print(f'Keyword args: {kwargs}')
        return func(*args, **kwargs)

    return modified_func

def add_mult(a, b, c):
    return (a + b) * c

with_print = print_args(add_mult)
with_print(1, 2, 3)
# Positional args: (1, 2, 3)
# Keyword args: {}
# => 9
```

Decorators!

Decorators!

```
def print_args(func):
    def modified_func(*args, **kwargs):
        print(f'Positional args: {args}')
        print(f'Keyword args: {kwargs}')
        return func(*args, **kwargs)

    return modified_func
```

Decorators!

take a function as an argument...

```
def print_args(func):
    def modified_func(*args, **kwargs):
        print(f'Positional args: {args}')
        print(f'Keyword args: {kwargs}')
        return func(*args, **kwargs)

    return modified_func
```

Decorators!

```
def print_args(func):
    def modified_func(*args, **kwargs):
        print(f'Positional args: {args}')
        print(f'Keyword args: {kwargs}')
        return func(*args, **kwargs)

    return modified_func
```

take a function as an argument...

...define a generic function...

Decorators!

```
def print_args(func):
    def modified_func(*args, **kwargs):
        print(f'Positional args: {args}')
        print(f'Keyword args: {kwargs}')
        return func(*args, **kwargs)

    return modified_func
```

take a function as an argument...

...define a generic function...

...which does something, e.g., prints out the arguments...

Decorators!

```
def print_args(func):
    def modified_func(*args, **kwargs):
        print(f'Positional args: {args}')
        print(f'Keyword args: {kwargs}')
        return func(*args, **kwargs)

    return modified_func
```

take a function as an argument...

...define a generic function...

...which does something, e.g., prints out the arguments...

...and then replicates the original function's behavior...

Decorators!

```
def print_args(func):
    def modified_func(*args, **kwargs):
        print(f'Positional args: {args}')
        print(f'Keyword args: {kwargs}')
        return func(*args, **kwargs)

    return modified_func
```

take a function as an argument...

...define a generic function...

...which does something, e.g., prints out the arguments...

...and then replicates the original function's behavior...

...then return that function.

Decorators!

```
def print_args(func):
    def modified_func(*args, **kwargs):
        print(f'Positional args: {args}')
        print(f'Keyword args: {kwargs}')
        return func(*args, **kwargs)

    return modified_func
```

take a function as an argument...

...define a generic function...

...which does something, e.g., prints out the arguments...

...and then replicates the original function's behavior...

...then return that function.

```
def add_mult(a, b, c):
    return (a + b) * c

with_print = print_args(add_mult)
```

modify (or "decorate") the add_mult function and store the new function as with_print

```
def add_mult(a, b, c):
    return (a + b) * c

add_mult = print_args(add_mult)
```

overwrite add_mult with the decorated version

```
@print_args
def add_mult(a, b, c):
    return (a + b) * c
```

👀 syntactic sugar for
add_mult = print_args(add_mult)

Demo: Our own `timeit`

Diagram: How does this program work?

```
1  from flask import Flask
2
3  app = Flask(__name__)
4
5  @app.route('/', methods=['GET'])
6  def greet():
7      return 'Hello world!'
8
```

```
class Flask:
    ...
    def route(self, rule: str, **options):
        # app.route(...) returns a decorator
        def decorator(f):
            # the decorator records the route...
            endpoint = options.pop("endpoint", None)
            self.add_url_rule(rule, endpoint, f, **options)
            # ... and then returns the function without modification
            return f

        return decorator
```

see the code on GitHub

Generators and infinite data

Demo: The Fibonacci sequence

Generators are "resumable functions"

```
def fibbi():
    a = 0
    b = 1

    while True:
        temp = a + b
        a = b
        b = temp

        yield a
```

```
f = fibbi()

print(f)
# => <generator object fibbi at 0x103275310>

next(f) # => 1
next(f) # => 1
next(f) # => 2
next(f) # => 3
next(f) # => 5
```

Generators are "resumable functions"

```
def fibbi():
    a = 0
    b = 1

    while True:
        temp = a + b
        a = b
        b = temp

        yield a
```

```
next(f) # => 8
f.gi_frame.f_locals
# => {'a': 8, 'b': 13}


next(f) # => 13
f.gi_frame.f_locals
# => {'a': 13, 'b': 21}
```

Turn & Talk: Why might this be useful?

Looping over a generator

Under the hood, for loops use next...

```
for i, x in enumerate(f):
    print(f'The {i + 1}th Fibonacci number is: {x}')

# The 1th Fibonacci number is: 1
# The 2th Fibonacci number is: 1
# The 3th Fibonacci number is: 2
# The 4th Fibonacci number is: 3
# The 5th Fibonacci number is: 5
# The 6th Fibonacci number is: 8
# The 7th Fibonacci number is: 13
# The 8th Fibonacci number is: 21
# The 9th Fibonacci number is: 34
# The 10th Fibonacci number is: 55
# ...
```

Generators can reduce memory usage

```
sum([
    x ** 2
    for x in range(100_000)
])
```

this loads the entire list into memory, and then takes the sum

```
sum((
    x ** 2
    for x in range(100_000)
))
```

this creates a generator, and can clean up each element once it's done

Generator comprehensions

```
for elem in collection:
    if condition(elem):
        yield fn(elem)
```

```
(
    fn(elem)
    for elem in collection
    if condition(elem)
)
```

Generator comprehensions

```
g = (
    x ** 2
    for x in range(100)
    if x % 2 != 0
)
```

```
next(g)  # => 1
next(g)  # => 9
next(g)  # => 25
next(g)  # => 49
next(g)  # => 81
next(g)  # => 121
```