GENERATORS AND DECORATORS

ITERATORS

AND

GENERATORS

Objectives

- Define Iterator and Iterable
- Understand the iter() and next() methods
- Build our own for loop
- Define what generators are and how they can be used
- Compare generator functions and generator expressions
- Use generators to pause execution of expensive functions
- Define what decorators are and how they can be used
- Create decorators to enhance the behavior of a function
- Explain what the wraps function is, and why it's used when writing decorators
- Create decorators that accept arguments

Iterators? Iterables??

Iterable - An object which will return an Iterator when *iter()* is called on it.

Iterator - an object that can be iterated upon. An object which returns data, one element at a time when next() is called on it

Ok But what does that really mean

"HELLO" is an iterable, but
it is not an iterator.

iter("HELLO") returns
an iterator

NEXT

When next() is called on an iterator, the iterator returns the next item. It keeps doing so until it raises a StopIteration error.

CUSTOM FOR LOOP

```
def for_loop(iterable, func):
    iterator = iter(iterable)
    while True:
        try:
            thing = next(iterator)
        except StopIteration:
            break
        else:
            func(thing)
```

Generators

- Generators are iterators
- Generators can be created with generator functions
- Generator functions use the yield keyword
- Generators can be created with generator expressions

Functions vs Generator Functions

Functions	**Generator Functions**
uses `return`	uses `yield`
returns once	can yield multiple times
When invoked, returns the return value	When invoked, returns a generator

Our First Generator

```
def count_up_to(max):
    count = 1
    while count <= max:
        yield count
        count += 1
```

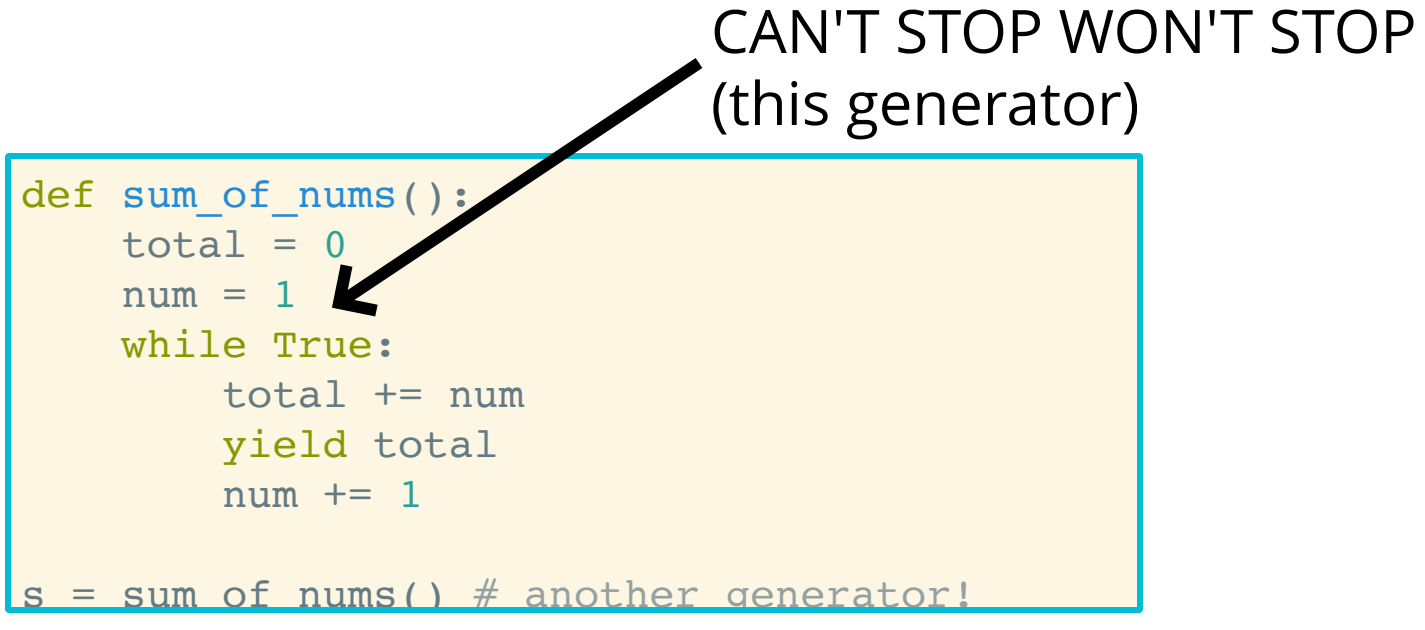
Exhausting a Generator

- Calling `next` on a generator with nothing left to yield will throw a `StopIteration` error
- When we loop over a generator, the loop will stop before the `StopIteration` error gets thrown

Generator Expressions

- You can also create generators from *generator expressions*
- Generator expressions look a lot like list comprehensions
- Generator expressions use () instead of []

Another Example




```
def sum_of_nums():
    total = 0
    num = 1
    while True:
        total += num
        yield total
        num += 1

s = sum_of_nums() # another generator!
```

Why Generators?

Lazy Evaluation

- Also called calculation on demand
- Only compute values as needed
- Can help improve performance of your code

An Example

Some Number Theory!

A number is called *abundant* if the sum of all of its proper divisors exceeds the number.

Examples:

- 12 (1 + 2 + 3 + 4 + 6 > 12)
- 18 (1 + 2 + 3 + 6 + 9 > 18)
- 20 (1 + 2 + 4 + 5 + 10 > 20)

Non-Examples:

- 4 (1 + 2 < 4)
- 6 (1 + 2 + 3 = 6)
- 15 (1 + 3 + 5 < 15)

Generating Abundant Numbers

```
def is_abundant(n):
    total = 0
    for d in range(1,n):
        if n % d == 0:
            total += d
    return total > n

is_abundant(12) # True
is_abundant(4) # False
```

Lists vs. Generators

```
def list_first_abundants(n):
    abundant_nums = []
    num = 1
    while len(abundant_nums) < n:
        if is_abundant(num):
            abundant_nums.append(num)
        num += 1
    return abundant_nums
```

```
def gen_first_abundants(n):
    count = 0
    num = 1
    while count < n:
        if is_abundant(num):
            yield num
            count += 1
        num +=1
```

Recap

- Generators are iterators
- Generators can be created with generator functions using the yield keyword
- Generators can be created with generator expressions
- Generators may or may not have terminating conditions
- Generators can provide memory savings
- Generators calculate values as they are needed

YOUR TURN

Introduction to Decorators

What's a Decorator??

- Decorators are functions
- Decorators wrap other functions and enhance their behavior
- Decorators are examples of higher order functions
- Decorators have their own syntax, using "@" (*syntactic sugar*)

Decorators as Functions

```
def be_polite(fn):
    def wrapper():
        print("What a pleasure to meet you!")
        fn()
        print("Have a great day!")
    return wrapper

def greet():
    print("My name is Colt.")

greet = be_polite(greet)
# we are decorating our function
# with politeness!
```

Decorator Syntax

```
def be_polite(fn):
    def wrapper():
        print("What a pleasure to meet you!")
        fn()
        print("Have a great day!")
    return wrapper

@be_polite
def greet():
    print("My name is Matt.")

# we don't need to set
# greet = be_polite(greet)
```

Functions with Different Signatures

```
def shout(fn):
    def wrapper(name):
        return fn(name).upper()
    return wrapper

@shout
def greet(name):
    return f"Hi, I'm {name}."

@shout
def order(main, side):
    return f"Hi, I'd like the {main}, with a side of {side}, please."
```

Decorator Pattern

```
def my_decorator(fn):
    def wrapper(*args, **kwargs):
        # do some stuff with fn(*args, **kwargs)
        pass
    return wrapper
```

Preserving Metadata

```
def log_function_data(fn):
    def wrapper(*args, **kwargs):
        print(f"you are about to call {fn.__name__}")
        print(f"Here's the documentation: {fn.__doc__}")
        return fn(*args, **kwargs)
    return wrapper

@log_function_data
def add(x,y):
    '''Adds two numbers together.'''
    return x + y;
```

Decorator Pattern

```
from functools import wraps
# wraps preserves a function's metadata
# when it is decorated

def my_decorator(fn):
    @wraps(fn)
    def wrapper(*args, **kwargs):
        # do some stuff with fn(*args, **kwargs)
        pass
    return wrapper
```

Using Decorators

Why Use Decorators?

- Removing code duplication across functions
- More easily perform function analytics/logging
- Exit out of a function early if certain conditions aren't met

Decorators Example

```
from functools import wraps
from time import time

def speed_test(fn):
    @wraps(fn)
    def wrapper(*args, **kwargs):
        t1 = time()
        result = fn(*args, **kwargs)
        t2 = time()
        print(f"Time Elapsed: {t2 - t1} seconds.")
        return result
    return wrapper
```

Another Example

```
from functools import wraps

def ensure_no_kwargs(fn):
    @wraps(fn)
    def wrapper(*args, **kwargs):
        if kwargs:
            return "No keyword arguments allowed!"
        return fn(*args)
    return wrapper
```

Decorators with Arguments

How can we write this decorator?

```
@ensure_first_arg_is("burrito")
def fav_foods(*foods):
    print(foods)

fav_foods("burrito", "ice cream")
  # ('burrito', 'ice cream')
fav_foods("ice cream", "burrito")
  # 'Invalid! First argument must be burrito'

@ensure_first_arg_is(10)
def add_to_ten(num1, num2):
    return num1 + num2

add_to_ten(10, 12) # 12
add_to_ten(1, 2)
  # 'Invalid! First argument must be 10'
```

Decorators with Arguments

```
def ensure_first_arg_is(val):
    def inner(fn):
        @wraps(fn)
        def wrapper(*args, **kwargs):
            if args and args[0] != val:
                return f"Invalid! First argument must be {val}"
            return fn(*args, **kwargs)
        return wrapper
    return inner
```

Recap

- Decorators are functions that enhance other functions
- Decorators use "@" as *syntactic sugar*
- In general, the functions that decorators return accept an unlimited number of positional and keyword arguments
- To preserve information about the decorated function, use *wraps*
- To write a decorator that accepts an argument, use another level of function nesting
- Decorators are useful for minimizing code duplication, analyzing functions, returning early from a function, and more!

YOUR
TURN