[301] Advanced Functions

Tyler Caraza-Harter

Learning Objectives Today

Iterators

- what is an iteratable?
- how to read files, with sequences or iterators
- advantages of laziness
- writing your own generators

References to functions

- ways to get a reference
- callbacks
- sort/map

Learning Objectives Today

Iterators

- what is an iteratable?
- how to read files, with sequences or iterators
- advantages of laziness
- writing your own generators

References to functions

- ways to get a reference
- callbacks
- sort/map

Revisiting the For Loop

for loops can iterate over **sequences**

- list values
- string characters
- other sequences

```
for letter in “hello”:
    print(letter)
```

```
for num in [1,2,3]:
    print(num)
```

Revisiting the For Loop

for loops can iterate over **sequences**

- list values
- string characters
- other sequences

More precisely…

```
for letter in “hello”:
    print(letter)
```

```
for num in [1,2,3]:
    print(num)
```

Revisiting the For Loop

for loops can iterate over **sequences**

- list values
- string characters
- other sequences

More precisely…

for loops can iterate over **iterables**

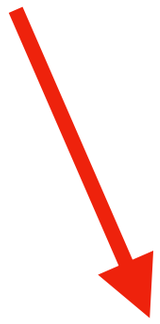
- **sequences** are **iterable**
- **other things** (like dict values) are also **iterable**

Example: Dictionary Values

```
d = {1:"one", 2:"two", 3:"three"}
d.values() # type is <class 'dict_values'>
```

Example: Dictionary Values

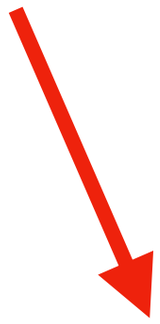
```
d = {1:"one", 2:"two", 3:"three"}
d.values() # type is <class 'dict_values'>
```



```
vals = list(d.values())
```

Example: Dictionary Values

```
d = {1:"one", 2:"two", 3:"three"}
d.values() # type is <class 'dict_values'>
```

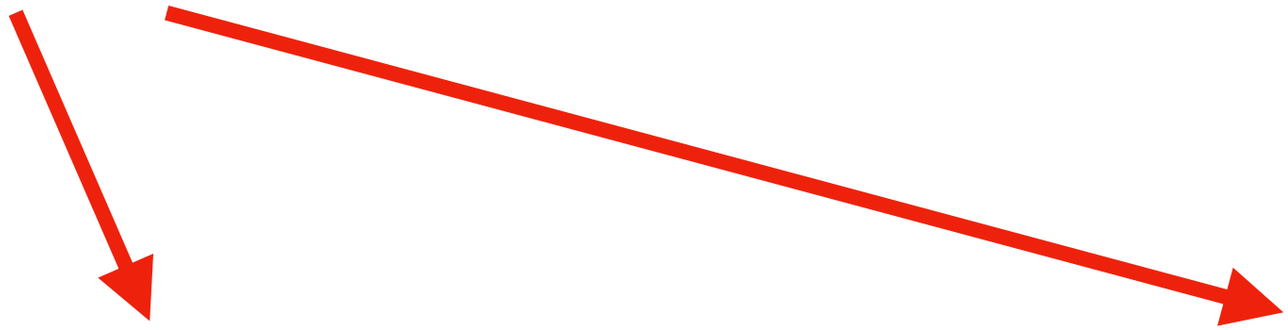


```
vals = list(d.values())

for v in vals:
    print(v)
```

Prints (or other order):
two
one
three

Example: Dictionary Values

```
d = {1:"one", 2:"two", 3:"three"}
d.values() # type is <class 'dict_values'>
```



```
vals = list(d.values())
```

```
it = iter(d.values())
```

```
for v in vals:
    print(v)
```

Example: Dictionary Values

```
d = {1:"one", 2:"two", 3:"three"}
d.values() # type is <class 'dict_values'>
```

```
vals = list(d.values())

for v in vals:
    print(v)
```

```
it = iter(d.values())
```

if you can call **iter(x)**,
then **x** is *iterable*,
by definition

Example: Dictionary Values

```
d = {1:"one", 2:"two", 3:"three"}
d.values() # type is <class 'dict_values'>
```

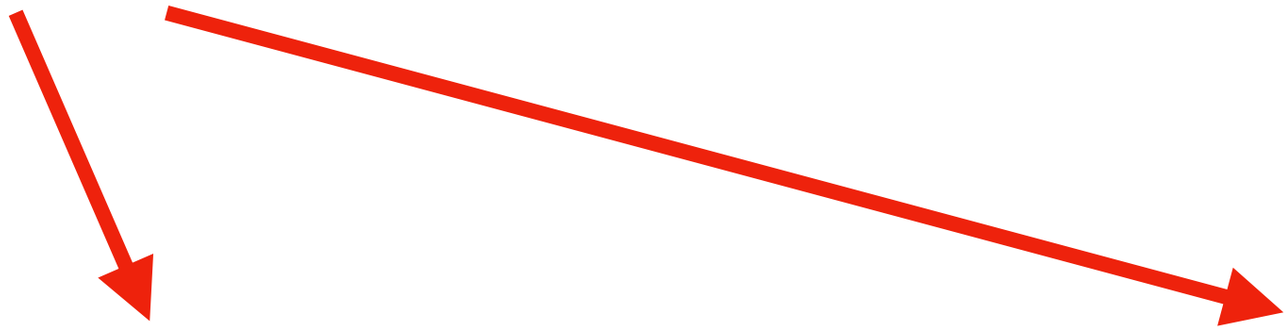
```
vals = list(d.values())

for v in vals:
    print(v)
```

```
it = iter(d.values())
```

d.values() is iterable, and **it** is an iterator

Example: Dictionary Values

```
d = {1:"one", 2:"two", 3:"three"}
d.values() # type is <class 'dict_values'>
```



```
vals = list(d.values())
for v in vals:
    print(v)
```

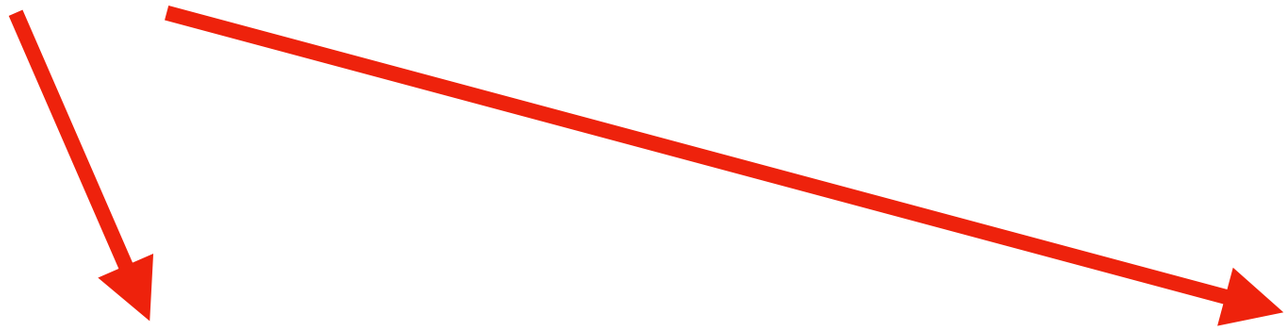
```
it = iter(d.values())
for v in it:
    print(v)
```

Both print the same:

two
one
three

Example: Dictionary Values

```
d = {1:"one", 2:"two", 3:"three"}
d.values() # type is <class 'dict_values'>
```



```
vals = list(d.values())
for v in vals:
    print(v)
```

```
it = iter(d.values())
for v in it:
    print(v)
```

Both print the same:

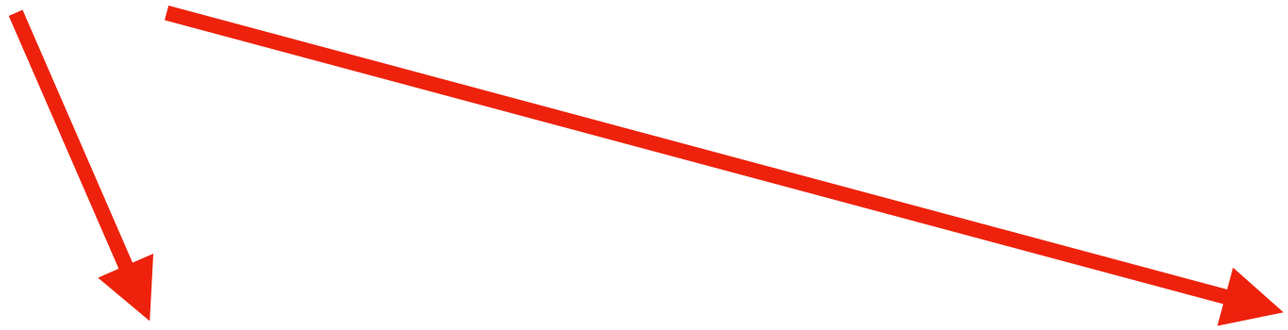
two
one
three

NOTE: the for loop automatically calls iter if necessary, so we could have written this instead:

```
for v in d.values():
    print(v)
```

Example: Dictionary Values

```
d = {1:"one", 2:"two", 3:"three"}
d.values() # type is <class 'dict_values'>
```



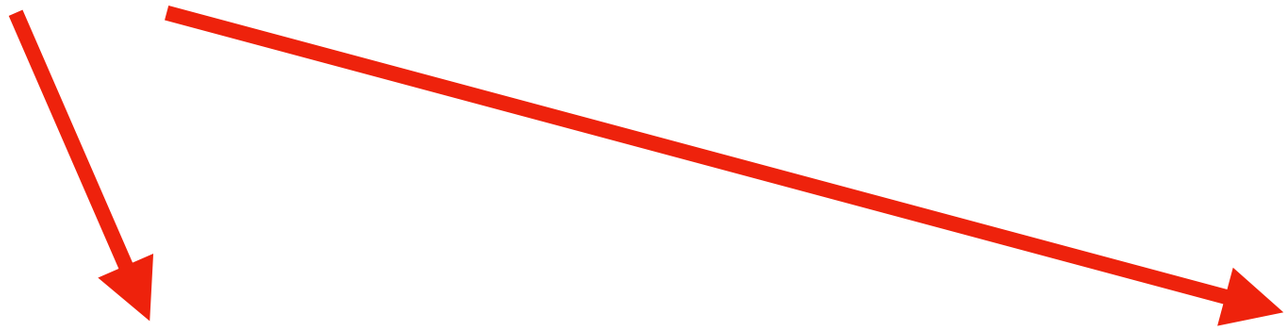
```
vals = list(d.values())

for v in vals:
    print(v)

print(vals[2])
```

```
it = iter(d.values())

for v in it:
    print(v)
```

We can index over a sequence.
Example prints:
three

Example: Dictionary Values

```
d = {1:"one", 2:"two", 3:"three"}
d.values() # type is <class 'dict_values'>
```

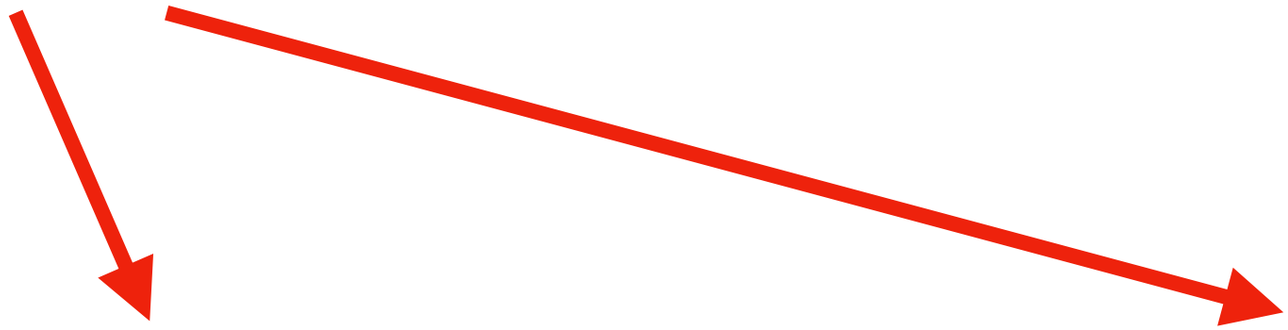


```
vals = list(d.values())

for v in vals:
    print(v)

print(vals[2])
```

```
it = iter(d.values())

for v in it:
    print(v)

print(it[2]) # BAD!
```

Example: Dictionary Values

```
d = {1:"one", 2:"two", 3:"three"}
d.values() # type is <class 'dict_values'>
```



```
vals = list(d.values())

for v in vals:
    print(v)

print(vals[2])
```

```
it = iter(d.values())

for v in it:
    print(v)

print(it[2]) # BAD!
```

```
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'dict_valueiterator' object is not subscriptable
```

You can only loop over iterators, not index with them

Comparison

	sequence	iterator
can use for loop	✓	✓
can do indexing	✓	✗

Comparison

	sequence	iterator
can use for loop	✓	✓
can do indexing	✓	✗

why ever use the less-capable iterator?

Comparison

	sequence	iterator
can use for loop	✓	✓
can do indexing	✓	✗

why ever use the less-capable iterator?

it’s often faster (as we’ll see later)

Learning Objectives Today

Iterators

- what is an iteratable?
- how to read files, with sequences or iterators
- advantages of laziness
- writing your own generators

References to functions

- ways to get a reference
- callbacks
- sort/map

Reading Files

```
path = “file.txt”
f = open(path)
```

Reading Files

```
path = “file.txt”
f = open(path)
```

open(…) function is built in

Reading Files

```
path = “file.txt”
f = open(path)
```

it takes a string argument,
which contains path to a file

file.txt

```
This is a test!
3
2
1
Go!
```

c:\users\tyler\my-doc.txt

/var/log/events.log

../data/input.csv

Reading Files

```
path = “file.txt”
f = open(path)
```

it returns a file object

file.txt

```
This is a test!
3
2
1
Go!
```

Reading Files

```
path = “file.txt”
f = open(path)
```

it returns a file object

file.txt

This is a test!
3
2
1
Go!

file objects are iterable!

Reading Files

```
path = “file.txt”
f = open(path)

for line in f:
    print(line)
```

Output

This is a test!

3

2

1

Go!

file.txt

This is a test!
3
2
1
Go!

Reading Files

```
path = “file.txt”
f = open(path)

for line in f:
    print(line.strip())
```

file.txt

```
This is a test!
3
2
1
Go!
```

Output

```
This is a test!
3
2
1
Go!
```

Reading Files

```
path = “file.txt”
f = open(path)

for line in f:
    print(line.strip())
```

file.txt

```
This is a test!
3
2
1
Go!
```

Another option: use the iterable file object to create a list

Reading Files

```
path = “file.txt”
f = open(path)
lines = list(f) # create list from iterable

for line in f:
    print(line.strip())
```

file.txt

```
This is a test!
3
2
1
Go!
```

Another option: use the iterable file object to create a list

Reading Files

```
path = “file.txt”
f = open(path)
lines = list(f) # create list from iterable

for line in f:
    print(line.strip())
```

file.txt

```
This is a test!
3
2
1
Go!
```

Another option: use the iterable file object to create a list

lines is a list:

```
[“This is a test\n”, “3\n”, “2\n”, “1\n”, “Go!\n”]
```

Reading Files

```
path = “file.txt”
f = open(path)
lines = list(f) # create list from iterable

for line in £ lines:
    print(line.strip())
```

file.txt

```
This is a test!
3
2
1
Go!
```

Another option: use the iterable file object to create a list

Reading Files

```
path = “file.txt”
f = open(path)
lines = list(f) # create list from iterable

for line in lines:
    print(line.strip())
```

file.txt

```
This is a test!
3
2
1
Go!
```

Another option: use the iterable file object to create a list

Learning Objectives Today

Iterators

- what is an iteratable?
- how to read files, with sequences or iterators
- advantages of laziness
- writing your own generators

References to functions

- ways to get a reference
- callbacks
- sort/map

Demo 1: Add numbers in a file

Goal: read all lines from a file as integers and add them

Input:

- file containing **50 million numbers** between 0 and 100

Output:

- The sum of the numbers

Example:

```
prompt> python sum.py
2499463617
```

Demo 1: Add numbers in a file

Goal: read all lines from a file as integers and add them

Input:

- file containing **50 million numbers** between 0 and 100

Output:

- The sum of the numbers

Example:

```
prompt> python sum.py
2499463617
```

Two ways:

- Put all lines in a list first
- Directly use iterable file

Learning Objectives Today

Iterators

- what is an iteratable?
- how to read files, with sequences or iterators
- advantages of laziness
- writing your own generators

References to functions

- ways to get a reference
- callbacks
- sort/map

Reviewing Return

```
def f():
  return “A”
  return “B”
  return “C”

print(“Got”, f())
```

What is printed?

Reviewing Return

```
def f():
  return “A”
  return “B”
  return “C”

print(“Got”, f())
```

What is printed?

Got A

Reviewing Return

```
def f():
  return “A”
  return “B”
  return “C”

print(“Got”, f())
```

What is printed?

Got A

Let’s say we want to return 3 values

Reviewing Return

```
def f():
  items = []
  items.append(“A”)
  items.append(“B”)
  items.append(“C”)
  return items

for item in f():
  print(“Got”, item)
```

What is printed?

Reviewing Return

```
def f():
  items = []
  items.append(“A”)
  items.append(“B”)
  items.append(“C”)
  return items

for item in f():
  print(“Got”, item)
```

What is printed?

Got A
Got B
Got C

Reviewing Return

```
def f():
  items = []
  print("Produce A")
  items.append("A")
  print("Produce B")
  items.append("B")
  print("Produce C")
  items.append("C")
  return items

for item in f():
  print("Got", item)
```

What is printed?

Reviewing Return

```
def f():
    items = []
    print("Produce A")
    items.append("A")
    print("Produce B")
    items.append("B")
    print("Produce C")
    items.append("C")
    return items

for item in f():
    print("Got", item)
```

What is printed?

Produce A
Produce B
Produce C

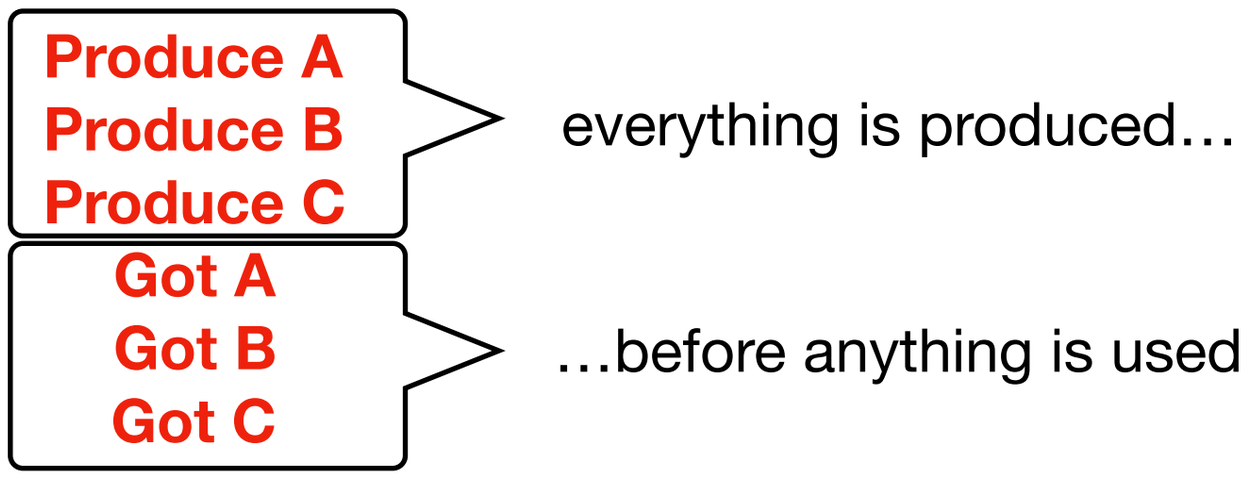
everything is produced…

Got A
Got B
Got C

Reviewing Return

```
def f():
    items = []
    print("Produce A")
    items.append("A")
    print("Produce B")
    items.append("B")
    print("Produce C")
    items.append("C")
    return items

for item in f():
    print("Got", item)
```

What is printed?



Reviewing Return

```
def f():
    items = []
    print("Produce A")
    items.append("A")
    print("Produce B")
    items.append("B")
    print("Produce C")
    items.append("C")
    return items

for item in f():
    print("Got", item)
```

What is printed?

Sometimes we want to be "lazy" and only produce values right before they're needed

Introducing Yield

```
def f():
  print(“Produce A”)
  yield “A”
  print(“Produce B”)
  yield “B”
  print(“Produce C”)
  yield “C”

items = f()
```

Introducing Yield

```
def f():
  print(“Produce A”)
  yield “A”
  print(“Produce B”)
  yield “B”
  print(“Produce C”)
  yield “C”

items = f()
```

what is yield?

- produce results, like return
- can yield multiple values, unlike return

Introducing Yield

```
def f():
  print(“Produce A”)
  yield “A”
  print(“Produce B”)
  yield “B”
  print(“Produce C”)
  yield “C”

items = f()
```

what is yield?

- produce results, like return
- can yield multiple values, unlike return

What is printed?

Introducing Yield

```
def f():
  print(“Produce A”)
  yield “A”
  print(“Produce B”)
  yield “B”
  print(“Produce C”)
  yield “C”

items = f()
```

what is yield?

- produce results, like return
- can yield multiple values, unlike return

What is printed?

nothing

Introducing Yield

```
def f():
  print(“Produce A”)
  yield “A”
  print(“Produce B”)
  yield “B”
  print(“Produce C”)
  yield “C”

items = f()
```

what is yield?

- produce results, like return
- can yield multiple values, unlike return
- functions with yield are lazy
  (don’t run until result is needed)

What is printed?

nothing

Introducing Yield

```
def f():
  print(“Produce A”)
  yield “A”
  print(“Produce B”)
  yield “B”
  print(“Produce C”)
  yield “C”

items = f()
```

type of items is:
<class 'generator'>

what is yield?

- produce results, like return
- can yield multiple values, unlike return
- functions with yield are lazy
  (don’t run until result is needed)

What is printed?

nothing

Introducing Yield

```
def f():
  print(“Produce A”)
  yield “A”
  print(“Produce B”)
  yield “B”
  print(“Produce C”)
  yield “C”

items = f()
```

what is yield?

- produce results, like return
- can yield multiple values, unlike return
- functions with yield are lazy
  (don’t run until result is needed)

What is printed?

nothing

type of items is:
<class 'generator'>

weird, no?
we don’t return anything

Introducing Yield

```
def f():
  print("Produce A")
  yield "A"
  print("Produce B")
  yield "B"
  print("Produce C")
  yield "C"

items = f()
```

what is yield?

- produce results, like return
- can yield multiple values, unlike return
- functions with yield are lazy
  (don't run until result is needed)
- functions with yield automatically return a *generator*, a type of iterator

What is printed?

nothing

type of items is:
<class 'generator'>

weird, no?
we don't return anything

Introducing Yield

```
def f():
  print("Produce A")
  yield "A"
  print("Produce B")
  yield "B"
  print("Produce C")
  yield "C"
```

what is yield?

- produce results, like return
- can yield multiple values, unlike return
- functions with yield are lazy
  (don't run until result is needed)
- functions with yield automatically return a *generator*, a type of iterator

```
items = f()
for item in items:
  print(item)
```

What is printed?

Introducing Yield

```
def f():
  print(“Produce A”)
  yield “A”
  print(“Produce B”)
  yield “B”
  print(“Produce C”)
  yield “C”

items = f()
for item in items:
  print(item)
```

what is yield?

- produce results, like return
- can yield multiple values, unlike return
- functions with yield are lazy (don’t run until result is needed)
- functions with yield automatically return a *generator*, a type of iterator

What is printed?

Produce A

Introducing Yield

```
def f():
  print("Produce A")
  yield "A"
  print("Produce B")
  yield "B"
  print("Produce C")
  yield "C"

items = f()
for item in items:
  print(item)
```

what is yield?

- produce results, like return
- can yield multiple values, unlike return
- functions with yield are lazy (don't run until result is needed)
- functions with yield automatically return a *generator*, a type of iterator

What is printed?

Produce A

Introducing Yield

```
def f():
  print(“Produce A”)
  yield “A”
  print(“Produce B”)
  yield “B”
  print(“Produce C”)
  yield “C”

items = f()
for item in items:
  print(item)
```

what is yield?

- produce results, like return
- can yield multiple values, unlike return
- functions with yield are lazy (don’t run until result is needed)
- functions with yield automatically return a *generator*, a type of iterator

What is printed?

Produce A

A

Introducing Yield

```
def f():
  print("Produce A")
  yield "A"
  print("Produce B")
  yield "B"
  print("Produce C")
  yield "C"
```

```
items = f()
for item in items:
  print(item)
```

what is yield?

- produce results, like return
- can yield multiple values, unlike return
- functions with yield are lazy (don't run until result is needed)
- functions with yield automatically return a *generator*, a type of iterator

What is printed?

Produce A

A

Introducing Yield

```
def f():
  print(“Produce A”)
  yield “A”
  print(“Produce B”)
  yield “B”
  print(“Produce C”)
  yield “C”

items = f()
for item in items:
  print(item)
```

what is yield?

- produce results, like return
- can yield multiple values, unlike return
- functions with yield are lazy (don’t run until result is needed)
- functions with yield automatically return a *generator*, a type of iterator

What is printed?

Produce A

A

Produce B

Introducing Yield

```
def f():
  print(“Produce A”)
  yield “A”
  print(“Produce B”)
  yield “B”
  print(“Produce C”)
  yield “C”

items = f()
for item in items:
  print(item)
```

what is yield?

- produce results, like return
- can yield multiple values, unlike return
- functions with yield are lazy (don’t run until result is needed)
- functions with yield automatically return a *generator*, a type of iterator

What is printed?

Produce A

A

Produce B

Introducing Yield

```
def f():
  print("Produce A")
  yield "A"
  print("Produce B")
  yield "B"
  print("Produce C")
  yield "C"

items = f()
for item in items:
  print(item)
```

what is yield?

- produce results, like return
- can yield multiple values, unlike return
- functions with yield are lazy
  (don't run until result is needed)
- functions with yield automatically return a *generator*, a type of iterator

What is printed?

Produce A
A
Produce B
B

Introducing Yield

```
def f():
  print(“Produce A”)
  yield “A”
  print(“Produce B”)
  yield “B”
  print(“Produce C”)
  yield “C”

items = f()
for item in items:
  print(item)
```

what is yield?

- produce results, like return
- can yield multiple values, unlike return
- functions with yield are lazy (don’t run until result is needed)
- functions with yield automatically return a *generator*, a type of iterator

What is printed?

Produce A
A
Produce B
B

Introducing Yield

```
def f():
  print(“Produce A”)
  yield “A”
  print(“Produce B”)
  yield “B”
  print(“Produce C”)
  yield “C”

items = f()
for item in items:
  print(item)
```

what is yield?

- produce results, like return
- can yield multiple values, unlike return
- functions with yield are lazy (don’t run until result is needed)
- functions with yield automatically return a *generator*, a type of iterator

What is printed?

Produce A
A
Produce B
B
Produce C

Introducing Yield

```
def f():
  print(“Produce A”)
  yield “A”
  print(“Produce B”)
  yield “B”
  print(“Produce C”)
  yield “C”

items = f()
for item in items:
  print(item)
```

what is yield?

- produce results, like return
- can yield multiple values, unlike return
- functions with yield are lazy (don’t run until result is needed)
- functions with yield automatically return a *generator*, a type of iterator

What is printed?

Produce A
A
Produce B
B
Produce C

Introducing Yield

```
def f():
  print(“Produce A”)
  yield “A”
  print(“Produce B”)
  yield “B”
  print(“Produce C”)
  yield “C”

items = f()
for item in items:
  print(item)
```

what is yield?

- produce results, like return
- can yield multiple values, unlike return
- functions with yield are lazy (don’t run until result is needed)
- functions with yield automatically return a *generator*, a type of iterator

What is printed?

Produce A
A
Produce B
B
Produce C
C

Introducing Yield

```
def f():
  print("Produce A")
  yield "A"
  print("Produce B")
  yield "B"
  print("Produce C")
  yield "C"

items = f()
for item in items:
  print(item)
```

what is yield?

- produce results, like return
- can yield multiple values, unlike return
- functions with yield are lazy
  (don't run until result is needed)
- functions with yield automatically
  return a *generator*, a type of iterator

observations

- we bounce in and out of
  a generator function
- the function starts producing
  values even before it finishes

What is printed?

Produce A
A
Produce B
B
Produce C
C

Demo 2: Squares

Goal: generate sequence of squares

Input:

- none

Output:

- Squares

Example:

```
prompt> python squares.py
1
4
9
16
25
...
```

Iterator/Generator Vocabulary Recap

Sequence: object we can loop over (with for) **AND** index into

Iterator: object we can loop over (with for)

Iterable: object **x** that can give us an iterator if we call **iter(x)**

Generator: simple iterator returned by a function that **yields**

Generator function: function that returns a generator

Learning Objectives Today

Iterators

- what is an iteratable?
- how to read files, with sequences or iterators
- advantages of laziness
- writing your own generators

References to functions

- ways to get a reference
- callbacks
- sort/map

Learning Objectives Today

Iterators

- what is an iteratable?
- how to read files, with sequences or iterators
- advantages of laziness
- writing your own generators

References to functions

- ways to get a reference
- callbacks
- sort/map

Learning Objectives Today

Iterators

- what is an iteratable?
- how to read files, with sequences or iterators
- advantages of laziness
- writing your own generators

References to functions

- ways to get a reference
- callbacks
- sort/map