

[301] Advanced Functions

Tyler Caraza-Harter

1 Functions as Objects

2 Iterators/Generators

Radical Claim:

Functions are Objects

Radical Claim:

Functions are Objects

implications:

- variables can reference functions
- lists/dicts can reference functions
- we can pass function references to other functions
- we can pass lists of function references to other functions
- ...

Function References (Part I)

Outline

- functions as objects
- sort

```
x = [1,2,3]
```

```
y = x
```

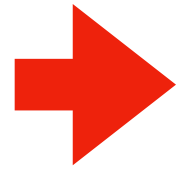
```
def f():  
    return "hi"
```

```
g = f
```

```
z = f()
```

your notes should probably include this example, with an explanation of what each of the 5 steps do!

which line of code is most novel for us?



```
x = [1, 2, 3]
```

```
y = x
```

```
def f():
```

```
    return "hi"
```

```
g = f
```

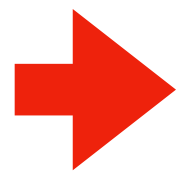
```
z = f()
```

State:

references

objects





```
x = [1, 2, 3]  
y = x
```

Explanation: x should reference a new list object

```
def f():  
    return "hi"
```

```
g = f
```

```
z = f()
```

State:

references

x 

objects

1	2	3
---	---	---

➔ `x = [1, 2, 3]`
`y = x`

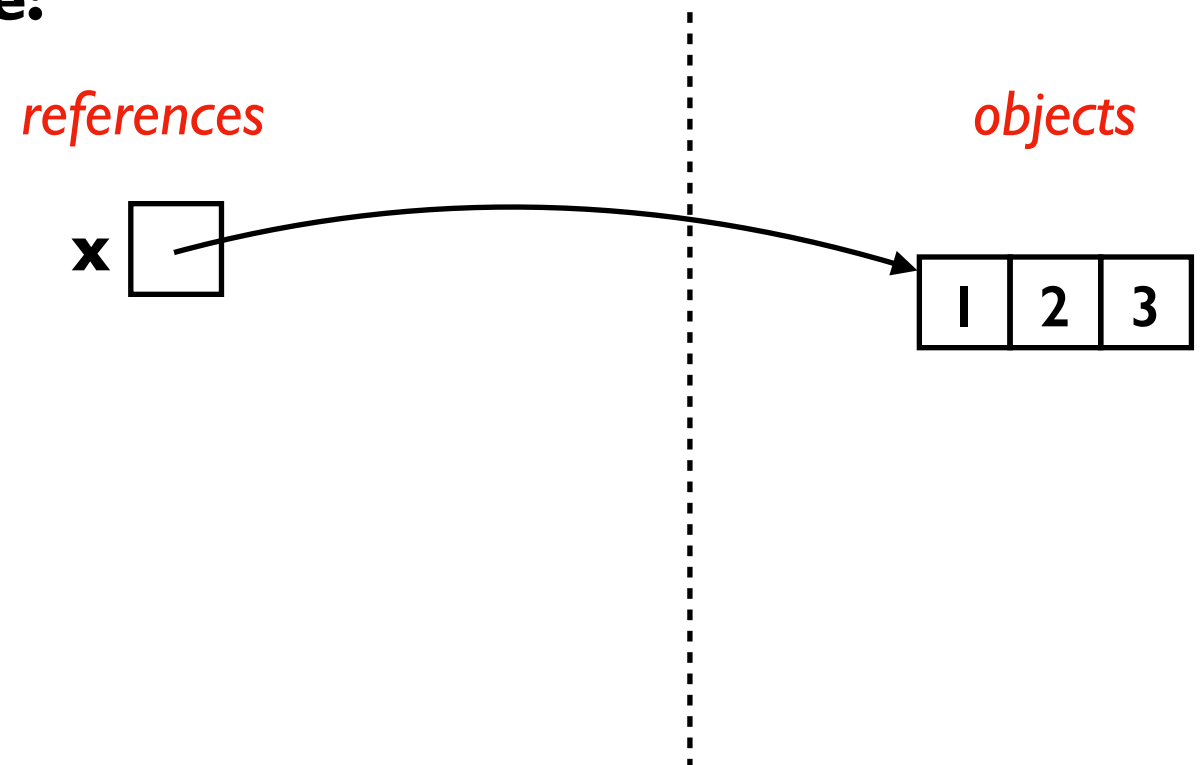
Explanation: x should reference a new list object

```
def f():  
    return "hi"
```

`g = f`

`z = f()`

State:





```
x = [1, 2, 3]
```

```
y = x
```

```
def f():  
    return "hi"
```

```
g = f
```

```
z = f()
```

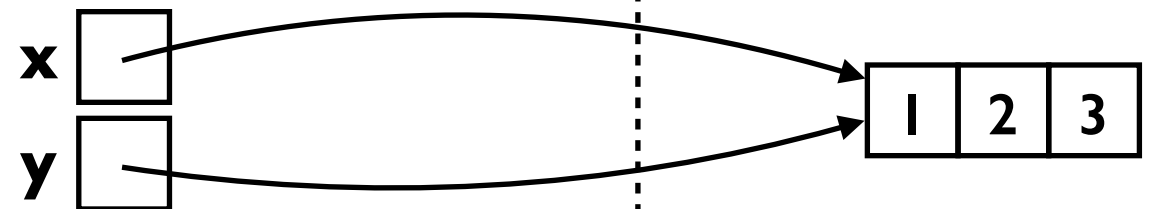
Explanation: x should reference a new list object

Explanation: y should reference whatever x references

State:

references

objects

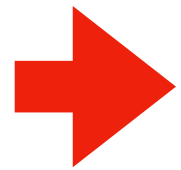


x = [1, 2, 3]

y = x

Explanation: x should reference a new list object

Explanation: y should reference whatever x references



```
def f():  
    return "hi"
```

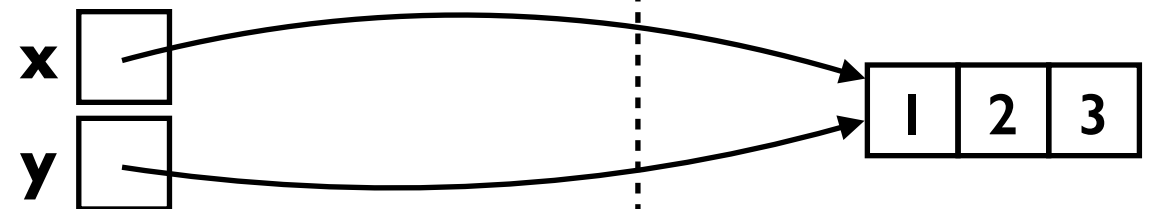
g = f

z = f()

State:

references

objects



x = [1, 2, 3]

y = x

def f():
 return "hi"

g = f

z = f()

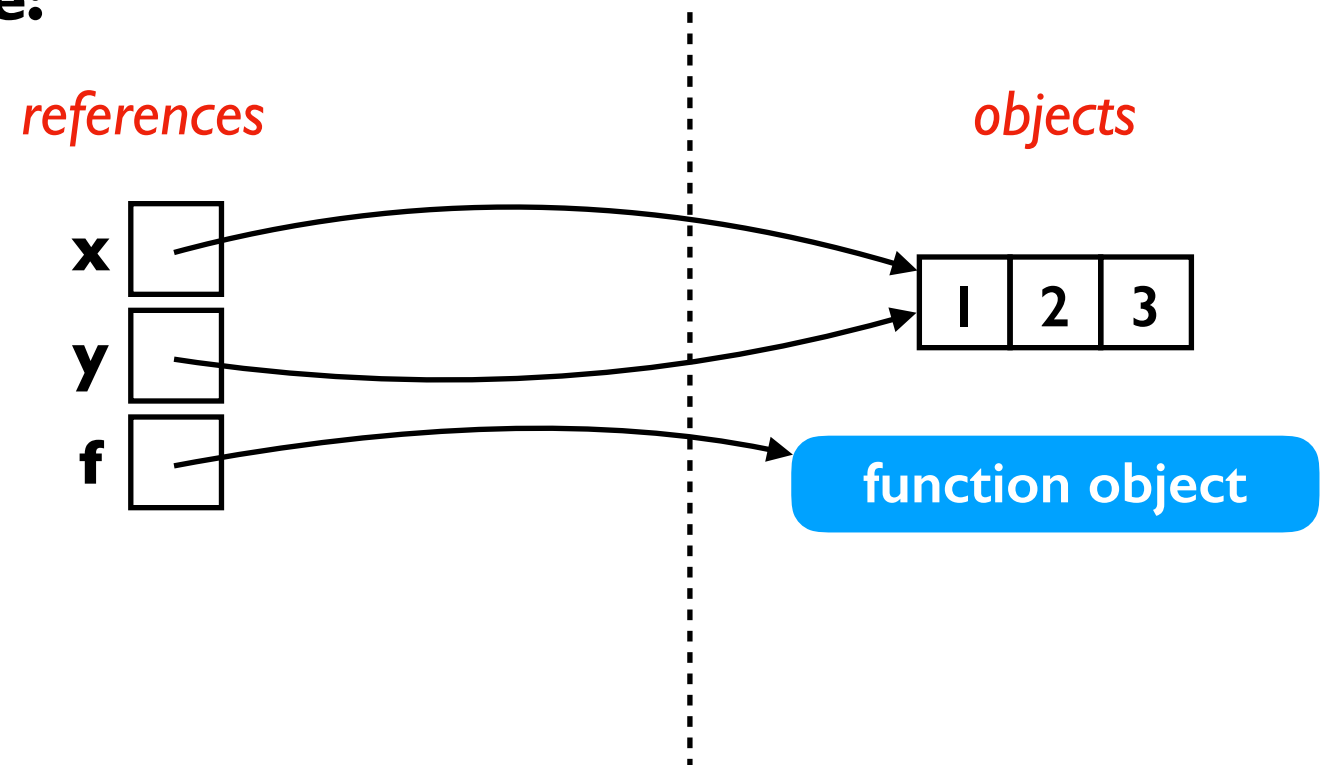
Explanation: x should reference a new list object

Explanation: y should reference whatever x references

Explanation: f should reference a new function object



State:



`x = [1, 2, 3]`

`y = x`

`def f():`
 `return "hi"`

➔ `g = f`

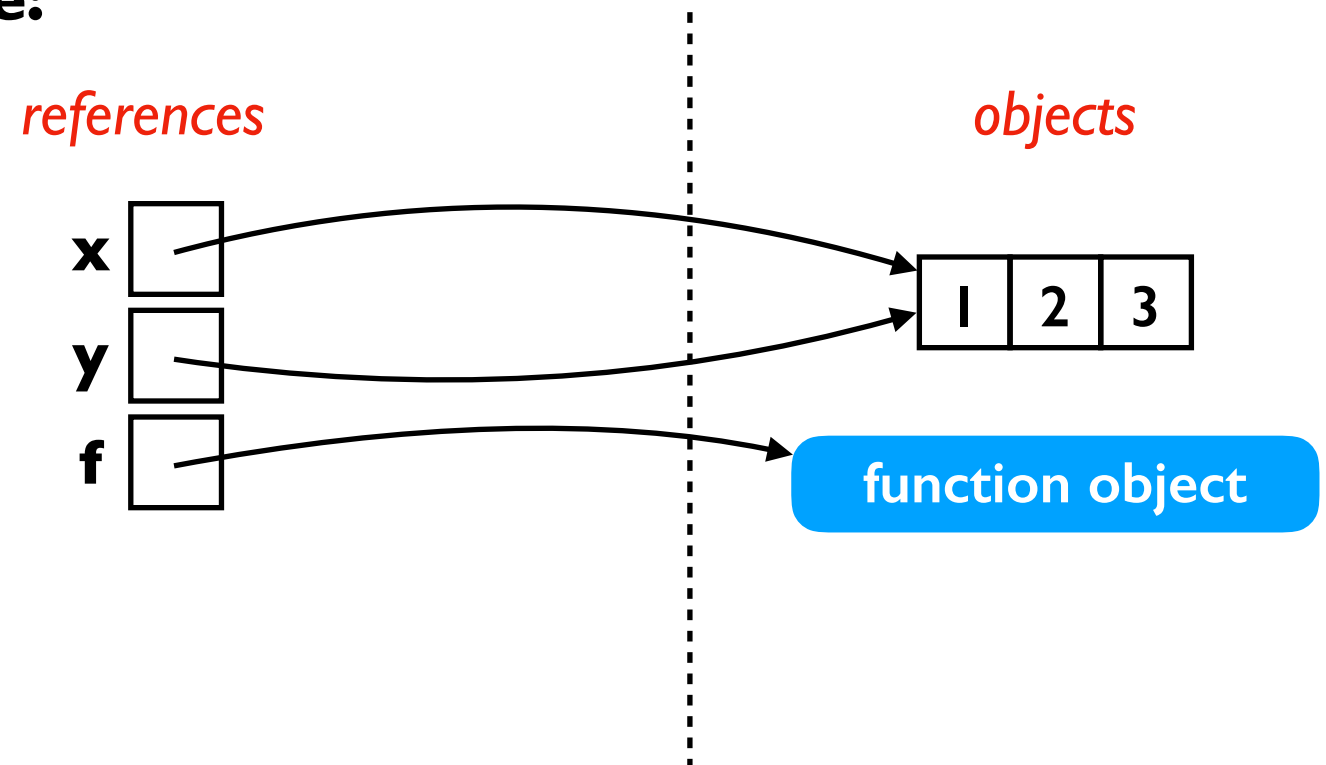
`z = f()`

Explanation: x should reference a new list object

Explanation: y should reference whatever x references

Explanation: f should reference a new function object

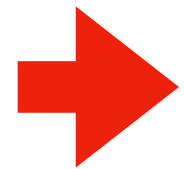
State:



x = [1, 2, 3]

y = x

def f():
 return "hi"



g = f

z = f()

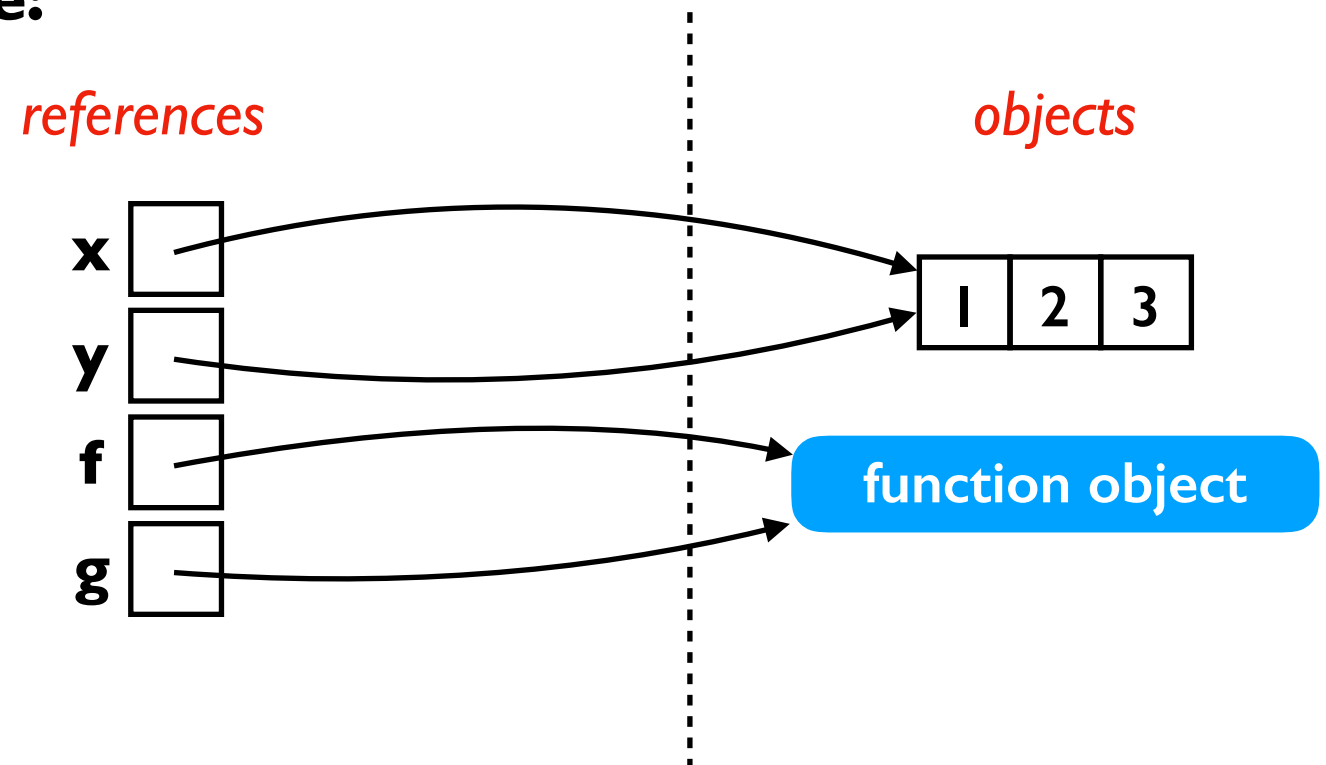
Explanation: x should reference a new list object

Explanation: y should reference whatever x references

Explanation: f should reference a new function object

Explanation: g should reference whatever f references

State:



x = [1, 2, 3]

y = x

def f():
 return "hi"

g = f

➔ z = f()

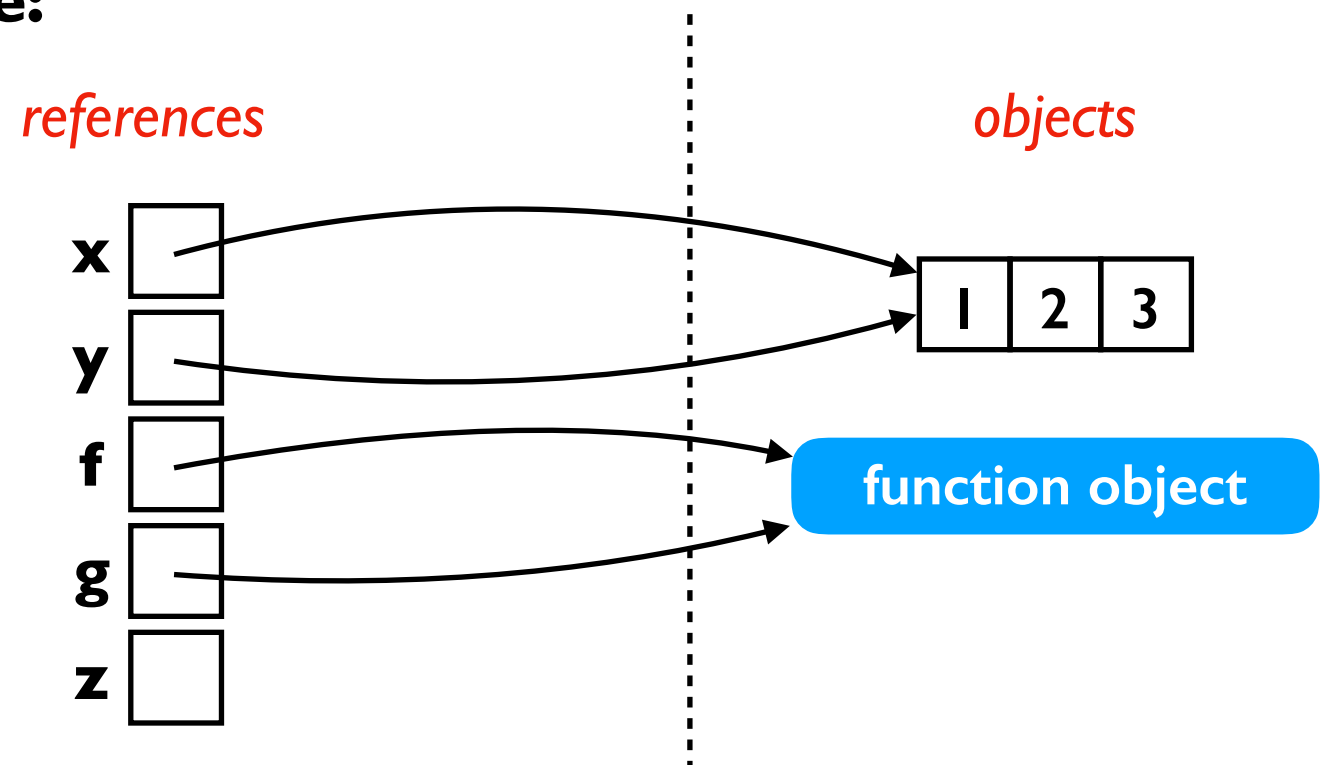
Explanation: x should reference a new list object

Explanation: y should reference whatever x references

Explanation: f should reference a new function object

Explanation: g should reference whatever f references

State:



x = [1, 2, 3]

y = x

def f():
 return "hi"

g = f

➔ z = f()

Explanation: x should reference a new list object

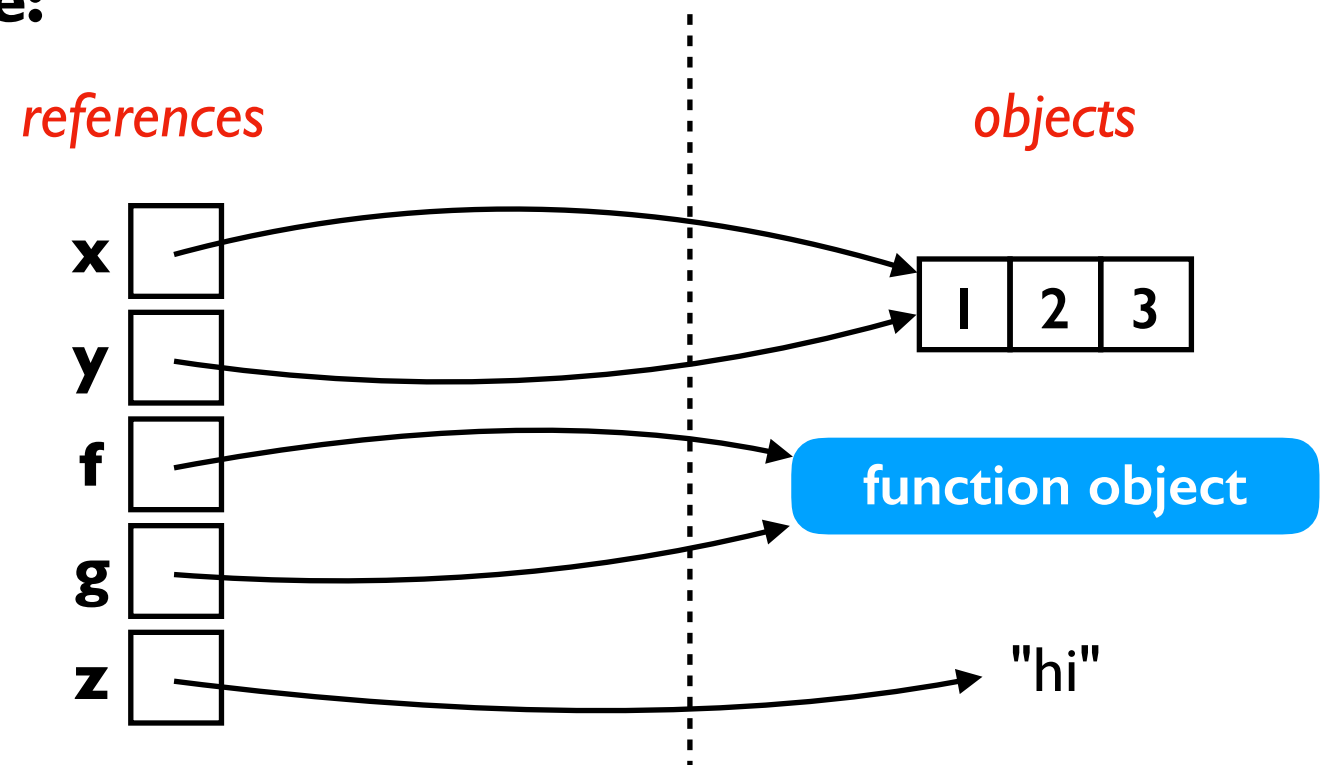
Explanation: y should reference whatever x references

Explanation: f should reference a new function object

Explanation: g should reference whatever f references

Explanation: z should reference whatever f returns

State:



`x = [1, 2, 3]`

`y = x`

`def f():`
 `return "hi"`

`g = f`

➔ `z = f()`

both of these calls would
have run the same code,
returning the same result:

- `z = f()`
- `z = g()`

Explanation: x should reference a new list object

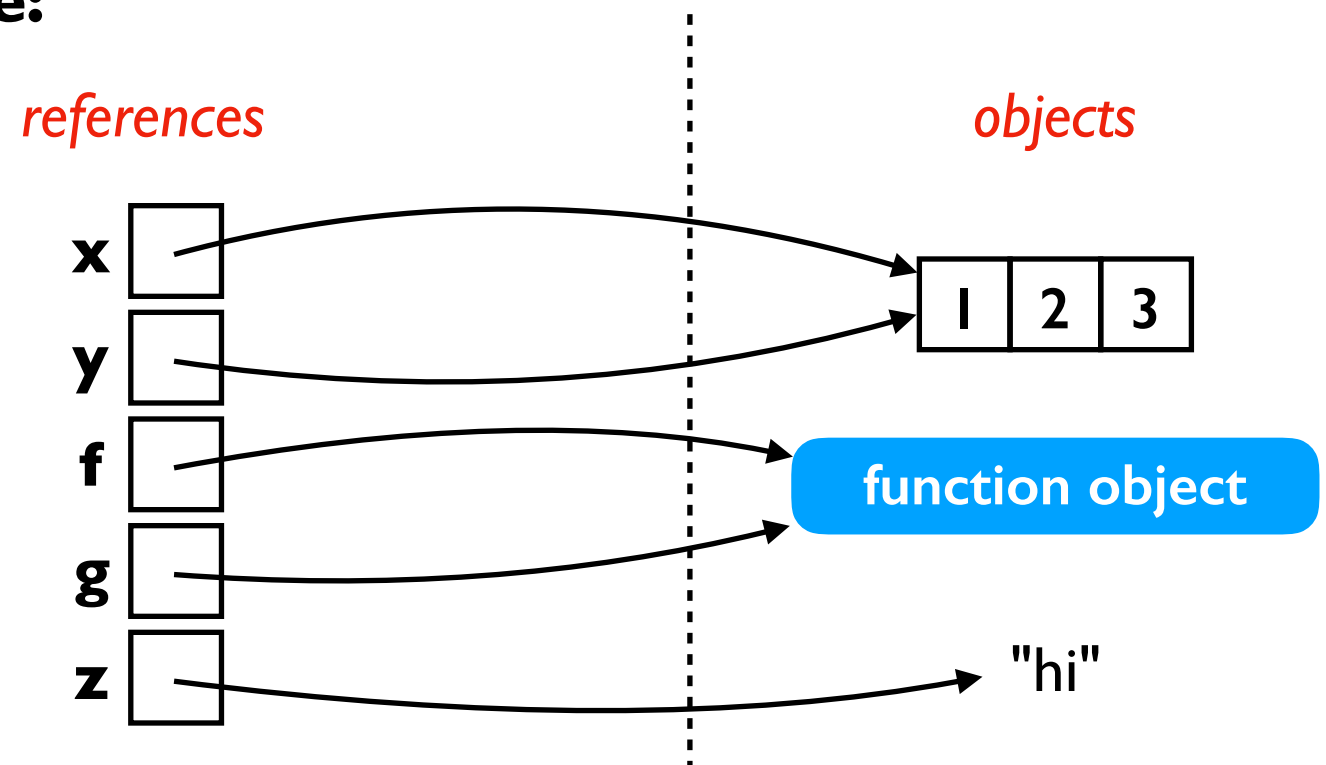
Explanation: y should reference whatever x references

Explanation: f should reference a new function object

Explanation: g should reference whatever f references

Explanation: z should reference whatever f returns

State:



x = [1, 2, 3]

y = x

```
def f():  
    return "hi"
```

g = f

z = f()

very similar (reference new object)



x = [1,2,3]

y = x

```
def f():  
    return "hi"
```

g = f

z = f()

very similar (reference new object)

very similar (reference existing object)

x = [1,2,3]

y = x

```
def f():  
    return "hi"
```

g = f

z = f()

very similar (reference new object)

very similar (reference existing object)

very different (invoke vs. reference)

```
graph LR; x["x = [1,2,3]"] -- black --> A["very similar (reference new object)"]; y["y = x"] -- blue --> A; y -- blue --> B["very similar (reference existing object)"]; f["def f():  
    return 'hi'"] -- black --> B; f -- blue --> C["very different (invoke vs. reference)"]; g["g = f"] -- blue --> C; z["z = f()"] -- red --> C;
```

CODING DEMOS

[Python Tutor]

Function References (Part I)

Outline

- functions as objects
- `sort`

Example: Sorting Names

List of tuples:

```
names = [  
    ("Catherine", "Baker"),  
    ("Alice", "Clark"),  
    ("Bob", "Adams"),  
]
```

Catherine	Baker
Bob	Adams
Alice	Clark

Example: Sorting Names

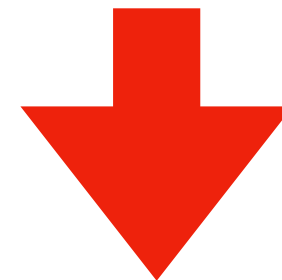
List of tuples:

```
names = [  
    ("Catherine", "Baker"),  
    ("Alice", "Clark"),  
    ("Bob", "Adams"),  
]
```

```
names.sort()
```

**sorting tuples is done
on first element**
(ties go to 2nd element)

Catherine	Baker
Bob	Adams
Alice	Clark



Alice	Clark
Bob	Adams
Catherine	Baker

Example: Sorting Names

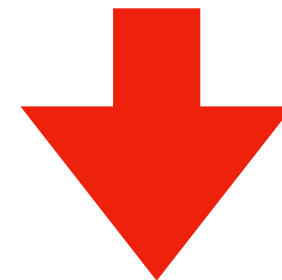
List of tuples:

```
names = [  
    ("Catherine", "Baker"),  
    ("Alice", "Clark"),  
    ("Bob", "Adams"),  
]
```

```
names.sort()
```

**what if we want to
sort by the last name?**

Catherine	Baker
Bob	Adams
Alice	Clark



Alice	Clark
Bob	Adams
Catherine	Baker

Example: Sorting Names

List of tuples:

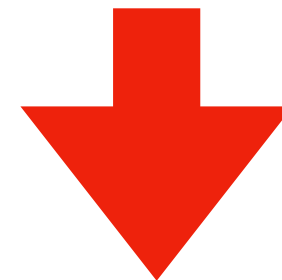
```
names = [  
    ("Catherine", "Baker"),  
    ("Alice", "Clark"),  
    ("Bob", "Adams"),  
]
```

```
names.sort()
```

**what if we want to
sort by the last name?**

or by the length of the name?

Catherine	Baker
Bob	Adams
Alice	Clark



Alice	Clark
Bob	Adams
Catherine	Baker

Example: Sorting Names

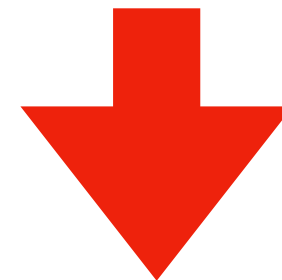
List of tuples:

```
names = [  
    ("Catherine", "Baker"),  
    ("Alice", "Clark"),  
    ("Bob", "Adams"),  
]
```

```
def extract(name_tuple):  
    return name_tuple[1]
```

```
names.sort(key=extract)
```

Catherine	Baker
Bob	Adams
Alice	Clark



Example: Sorting Names

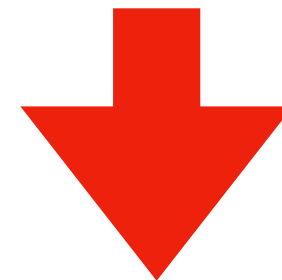
List of tuples:

```
names = [  
    ("Catherine", "Baker"),  
    ("Alice", "Clark"),  
    ("Bob", "Adams"),  
]
```

```
def extract(name_tuple):  
    return name_tuple[1]
```

```
names.sort(key=extract)
```

Catherine	Baker
Bob	Adams
Alice	Clark



Bob	Adams
Catherine	Baker
Alice	Clark

Example: Sorting Names

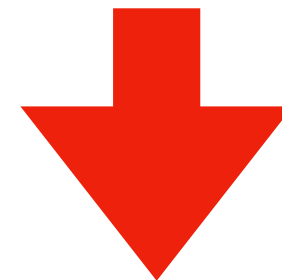
List of tuples:

```
names = [  
    ("Catherine", "Baker"),  
    ("Alice", "Clark"),  
    ("Bob", "Adams"),  
]
```

```
def extract(name_tuple):  
    return len(name_tuple[0])
```

```
names.sort(key=extract)
```

Catherine	Baker
Bob	Adams
Alice	Clark



Example: Sorting Names

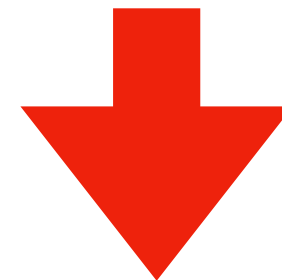
List of tuples:

```
names = [  
    ("Catherine", "Baker"),  
    ("Alice", "Clark"),  
    ("Bob", "Adams"),  
]
```

```
def extract(name_tuple):  
    return len(name_tuple[0])
```

```
names.sort(key=extract)
```

Catherine	Baker
Bob	Adams
Alice	Clark



Bob	Adams
Alice	Clark
Catherine	Baker

[301] Advanced Functions

Tyler Caraza-Harter



Functions as Objects



Iterators/Generators

Iterators/Generators (Part 2)

Outline

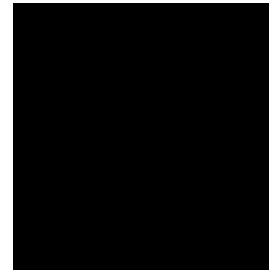
- when normal functions aren't good enough
- yield keyword by example
- the scary vocabulary of iteration
- the open function
- demos

```
def get_one_digit_nums():  
    print("START")  
    nums = []  
    i = 0  
    while i < 10:  
        nums.append(i)  
        i += 1  
    print("END")  
    return nums  
  
for x in get_one_digit_nums():  
    print(x)
```

how many times is the word "START" printed?


```
def get_one_digit_nums():  
    print("START")  
    nums = []  
    i = 0  
    while i < 10:  
        nums.append(i)  
        i += 1  
    print("END")  
    return nums
```

```
for x in get_one_digit_nums():  
    print(x)
```



stage 1

stage 2

running get_one_digit_nums code

looping over results and printing

time

```
def get_primes():  
    print("START")  
    nums = []  
    i = 0  
    while True:  
        if is_prime(i):  
            nums.append(i)  
            i += 1  
    print("END")  
    return nums  
  
for x in get_primes():  
    print(x)
```

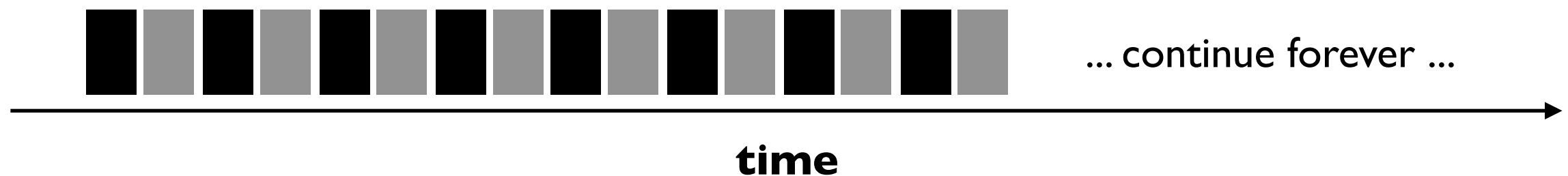
what does this code do?
assume there is an earlier
`is_prime` function

```
def get_primes():  
    print("START")  
    nums = []  
    i = 0  
    while True:  
        if is_prime(i):  
            nums.append(i)  
        i += 1  
    print("END")  
    return nums  
  
for x in get_primes():  
    print(x)
```

to make this work, we'll need to learn a completely new kind of function, the **generator**

```
def get_primes():  
    ...  
  
for x in get_primes():  
    print(x)
```

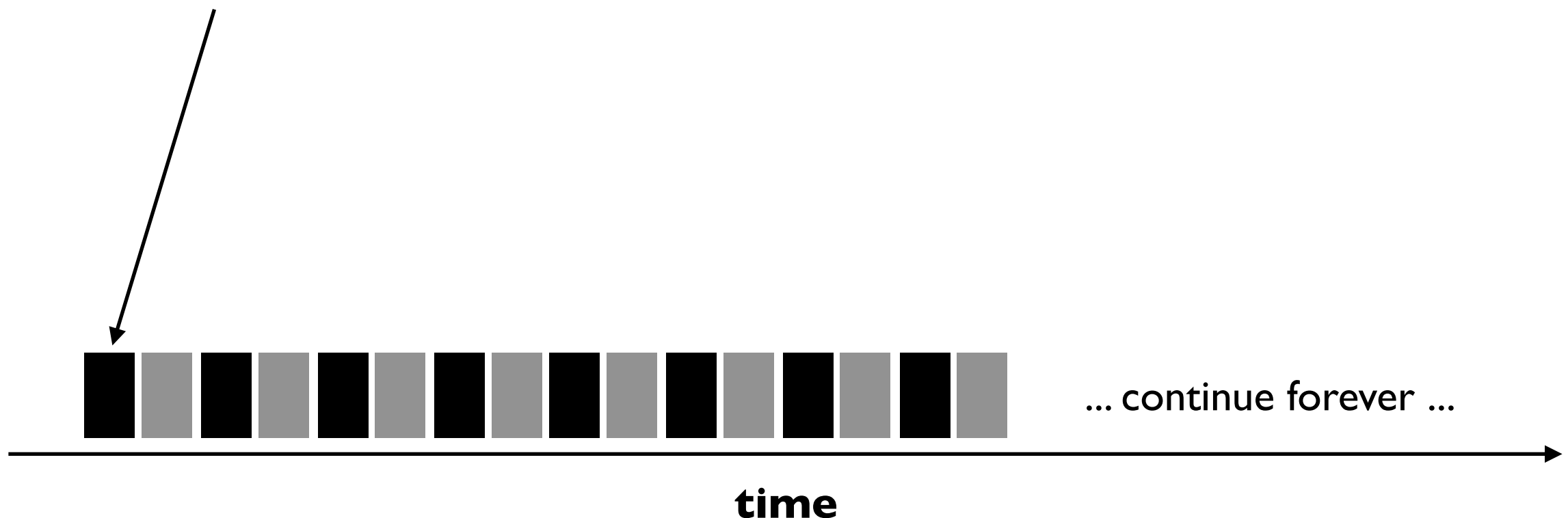
what we want:



```
def get_primes():  
    ...  
  
for x in get_primes():  
    print(x)
```

run `get_primes` just long
enough to get one prime

LAZY (contrast with "eager")



```
def get_primes():  
    ...  
  
for x in get_primes():  
    print(x)
```

run `get_primes` just long
enough to get one prime

print one number

LAZY (contrast with "eager")



... continue forever ...

time

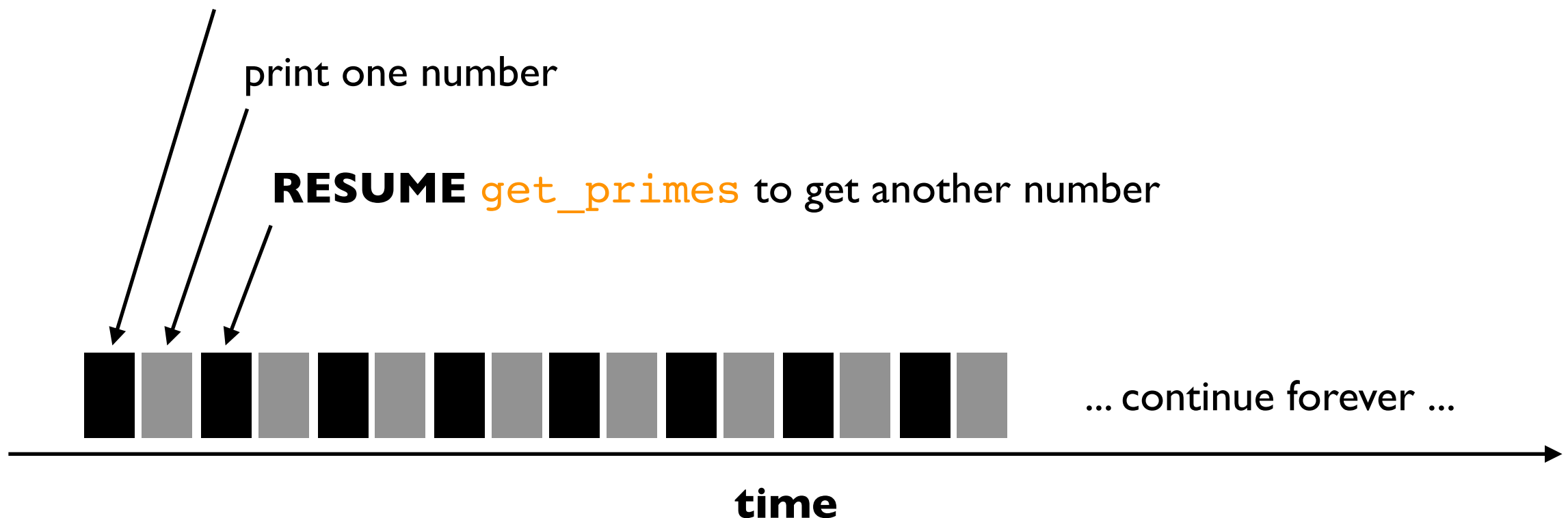
```
def get_primes():  
    ...  
  
for x in get_primes():  
    print(x)
```

run `get_primes` just long
enough to get one prime

LAZY (contrast with "eager")

print one number

RESUME `get_primes` to get another number



```
def get_primes():  
    ...  
  
for x in get_primes():  
    print(x)
```

we will stop and resume running
`get_primes` many times, even
though we only "call" it once

run `get_primes` just long
enough to get one prime

LAZY (contrast with "eager")

print one number

RESUME `get_primes` to get another number



... continue forever ...

time


```
def get_primes():  
    ...  
  
for x in get_primes():  
    print(x)
```

we will stop and resume running `get_primes` many times, even though we only "call" it once

functions with this stop/resume behavior are called generators

run `get_primes` just long enough to get one prime

LAZY (contrast with "eager")

print one number

RESUME `get_primes` to get another number



... continue forever ...

time

```
def get_primes():  
    ... some code ...
```

```
yield VALUE
```

```
    ... more code ...
```

any function containing the **yield** keyword anywhere is a generator

if you see this, all bets are off
regarding how you currently
understand functions to behave

?

```
gen def get_primes():  
    ... some code ...  
  
    yield VALUE  
  
    ... more code ...
```

any function containing the **yield** keyword anywhere is a generator

if you see this, all bets are off
regarding how you currently
understand functions to behave

should we even consider it a function?

?

```
gen def get_primes():  
    ... some code ...  
  
    yield VALUE  
  
    ... more code ...
```

any function containing the **yield** keyword anywhere is a generator

if you see this, all bets are off regarding how you currently understand functions to behave

should we even consider it a function?



Should we "introduce another new keyword (say, *gen* or *generator*) in place of *def*"?

Guido van Rossum

Python's Benevolent Dictator for Life
(until recently)

?

```
gen def get_primes():  
    ... some code ...  
  
    yield VALUE  
  
    ... more code ...
```

any function containing the **yield** keyword anywhere is a generator

if you see this, all bets are off regarding how you currently understand functions to behave

should we even consider it a function?



Argument for gen: *"a yield statement buried in the body is not enough warning that the semantics are so different"*

Argument for def: *"generators are functions, but with the twist that they're resumable"*

Guido van Rossum

Python's Benevolent Dictator for Life
(until recently)

```
def get_primes():  
    ... some code ...
```

```
yield VALUE
```

```
... more code ...
```

*always scan a function for yields
when trying to understand it*



*Argument for gen: "a yield statement buried
in the body is not enough warning that the semantics
are so different"*



*Argument for def: "generators are functions, but
with the twist that they're resumable"*



Guido van Rossum

Python's Benevolent Dictator for Life
(until recently)

Iterators/Generators (Part 2)

Outline

- when normal functions aren't good enough
- **yield keyword by example**
- the scary vocabulary of iteration
- the open function
- demos

yield by example (note, PyTutor does a bad job showing generators)

```
def f():  
    yield 1  
    yield 2  
    yield 3  
  
for x in f():  
    print(x)
```

```
def f():  
    print("A")  
    yield 1  
    print("B")  
    yield 2  
    print("C")  
    yield 3  
  
for x in f():  
    print(x)
```

```
def f():  
    yield 1  
    yield 2  
    yield 3  
  
for x in f():  
    print(x)  
  
for x in f():  
    print(x)
```

```
def f():  
    yield 1  
    yield 2  
    yield 3  
  
for x in f():  
    for y in f():  
        print(x, y)
```

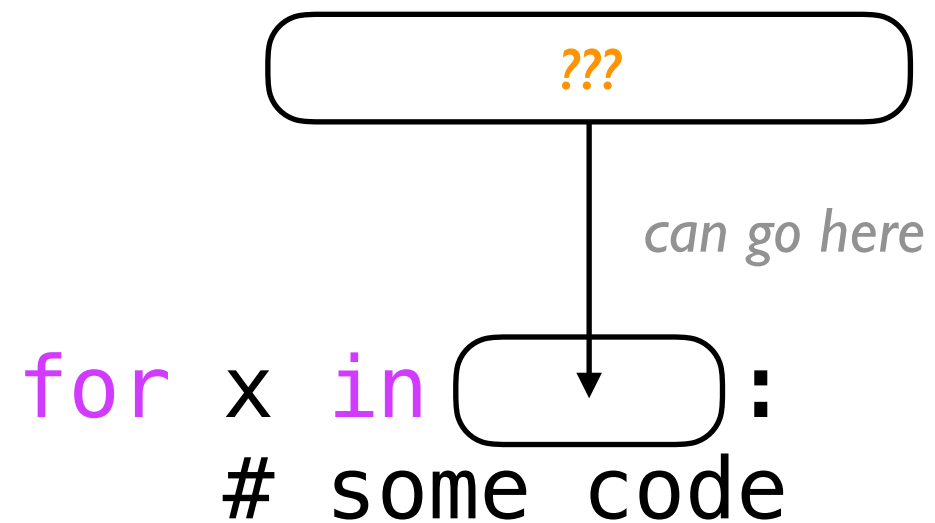
```
def f():  
    yield 1  
    yield 2  
    yield 3  
  
gen = f()  
for x in gen:  
    print(x)
```


Iterators/Generators (Part 2)

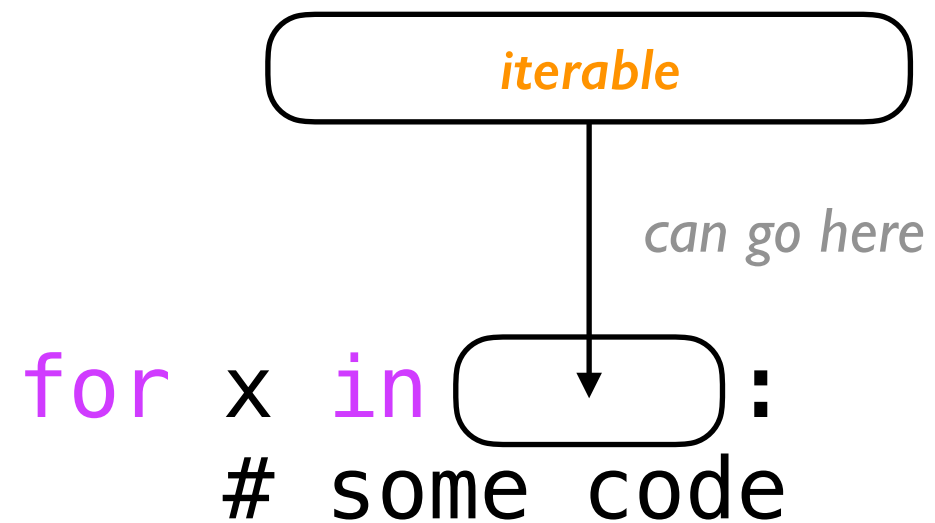
Outline

- when normal functions aren't good enough
- yield keyword by example
- the scary vocabulary of iteration
- the open function
- demos

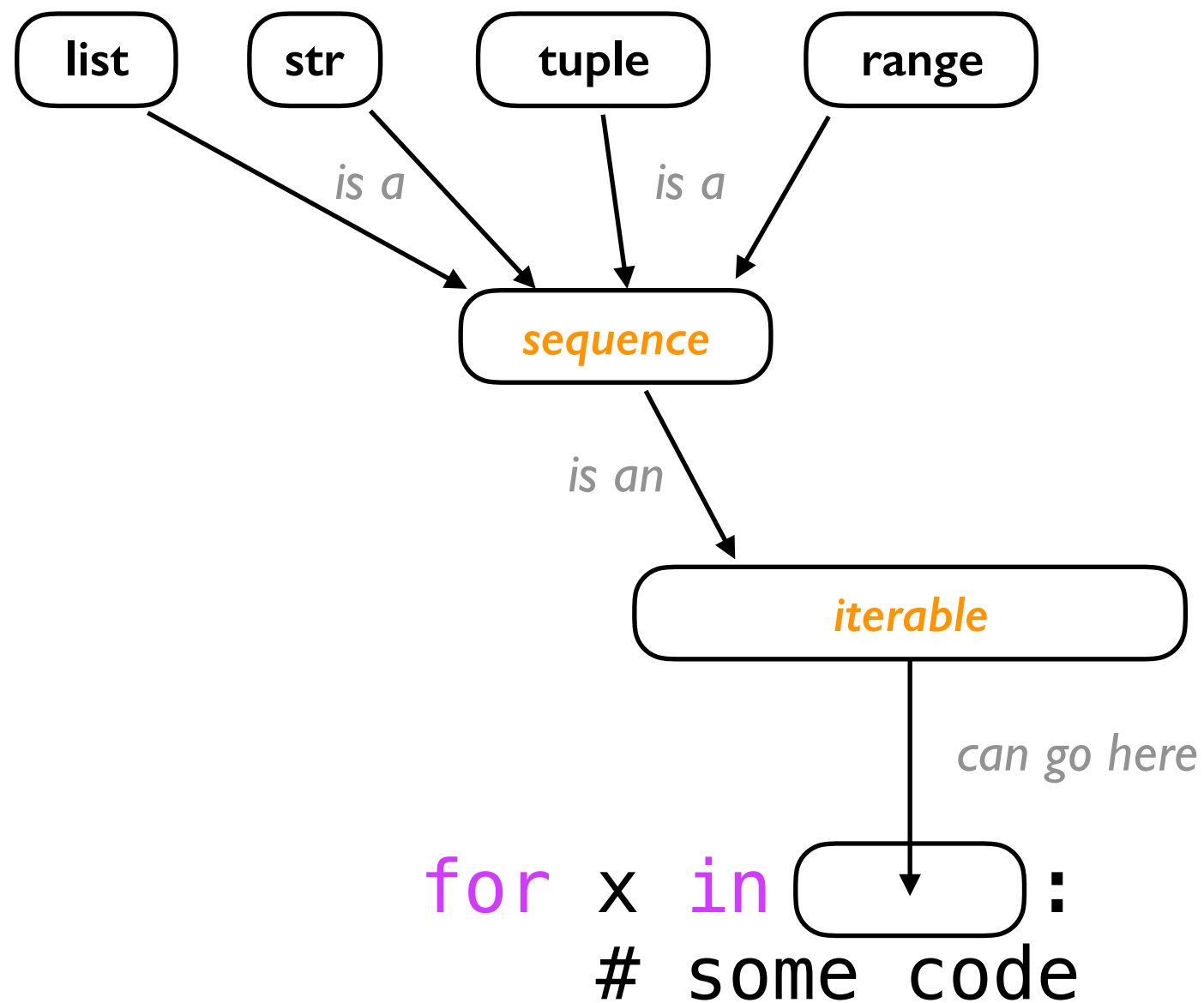
The Vocabulary of Iteration



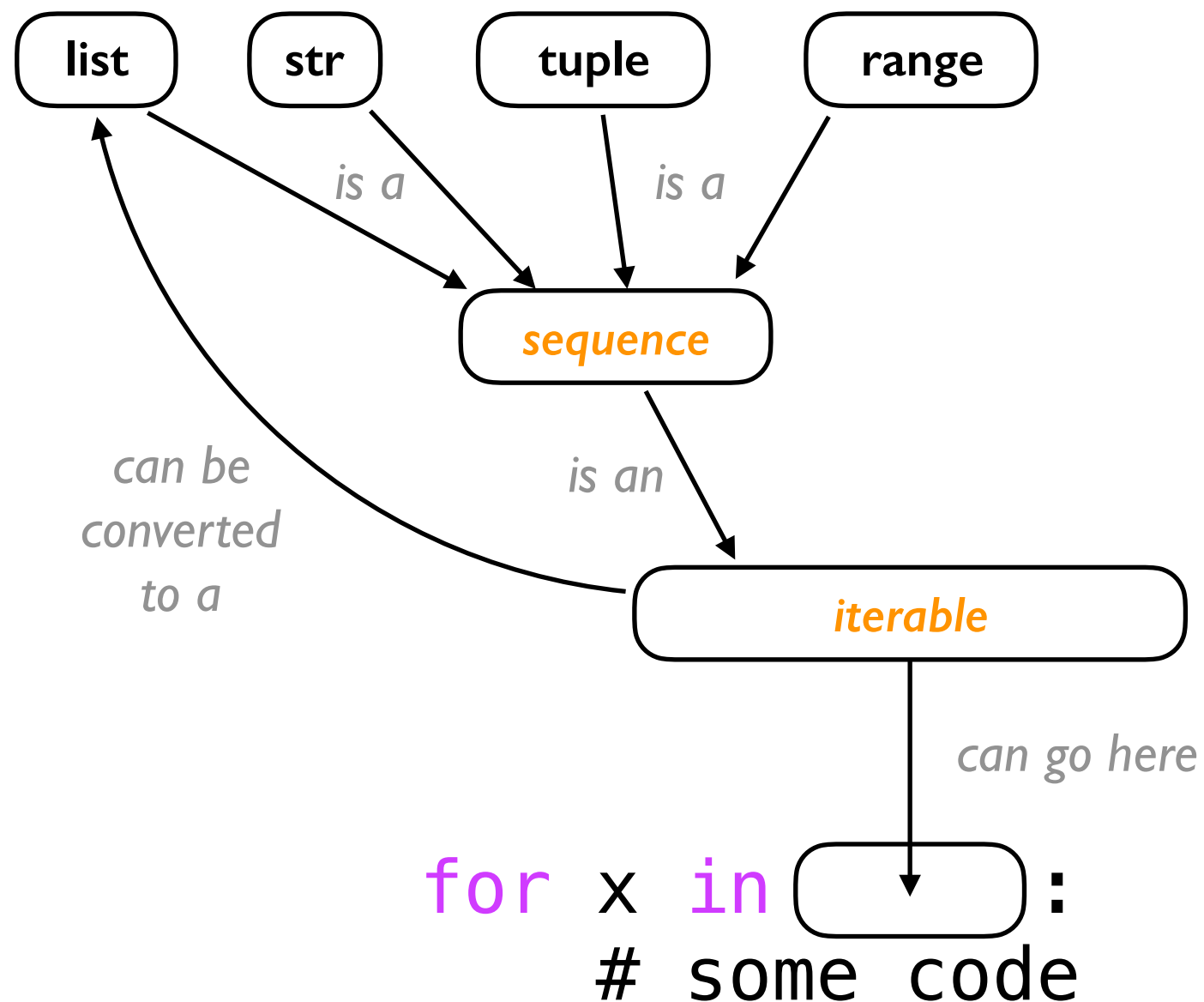
The Vocabulary of Iteration



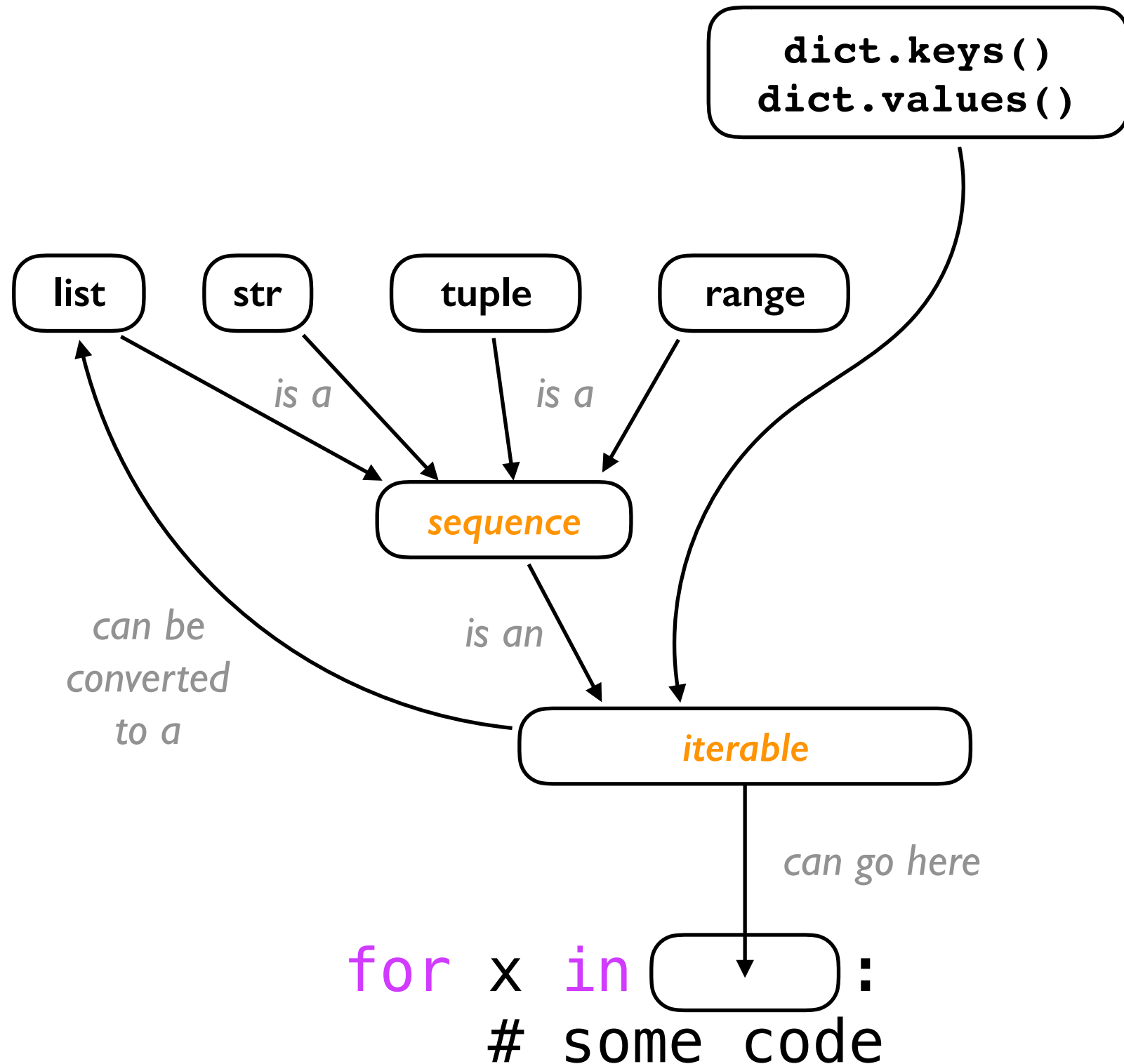
The Vocabulary of Iteration



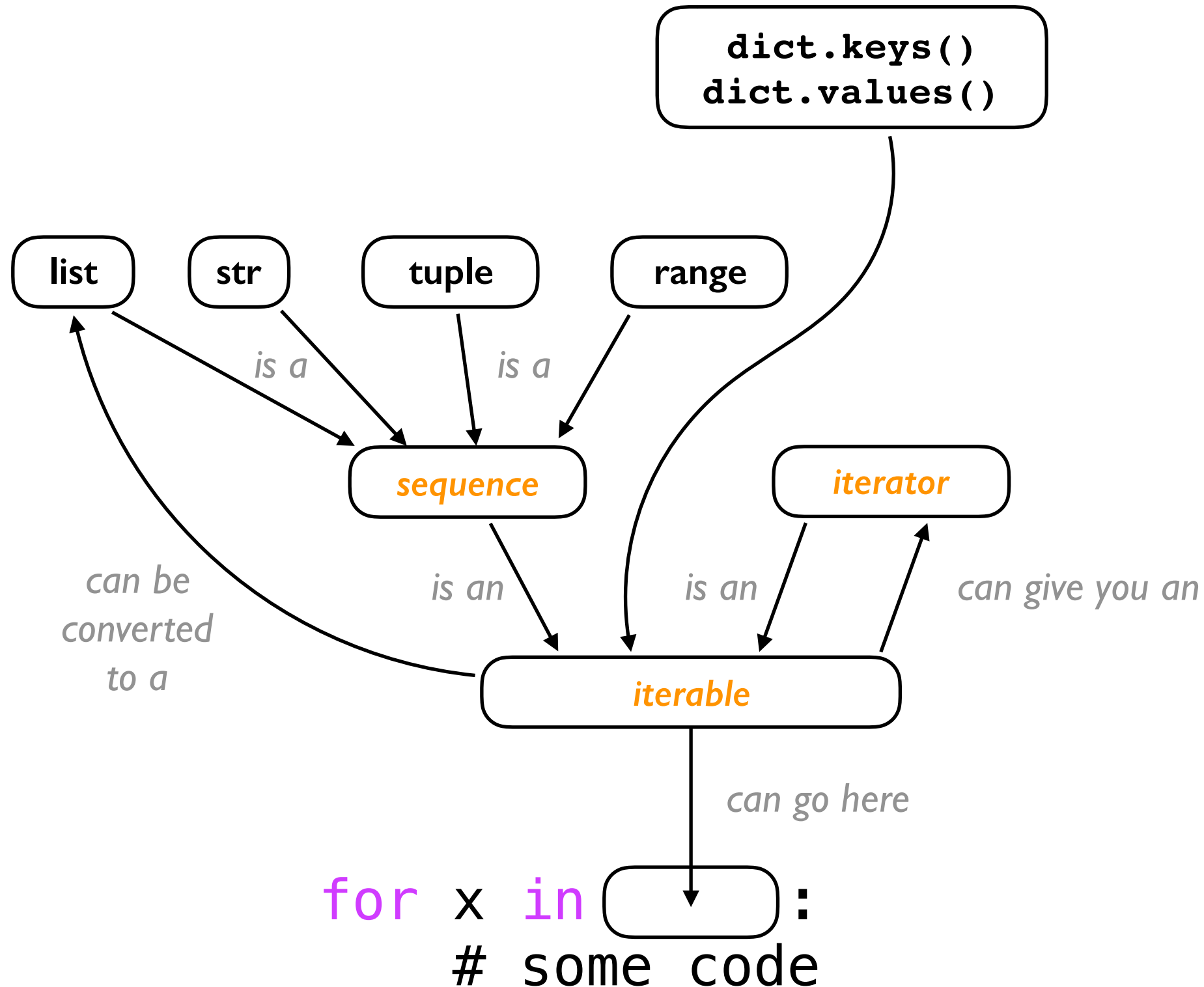
The Vocabulary of Iteration



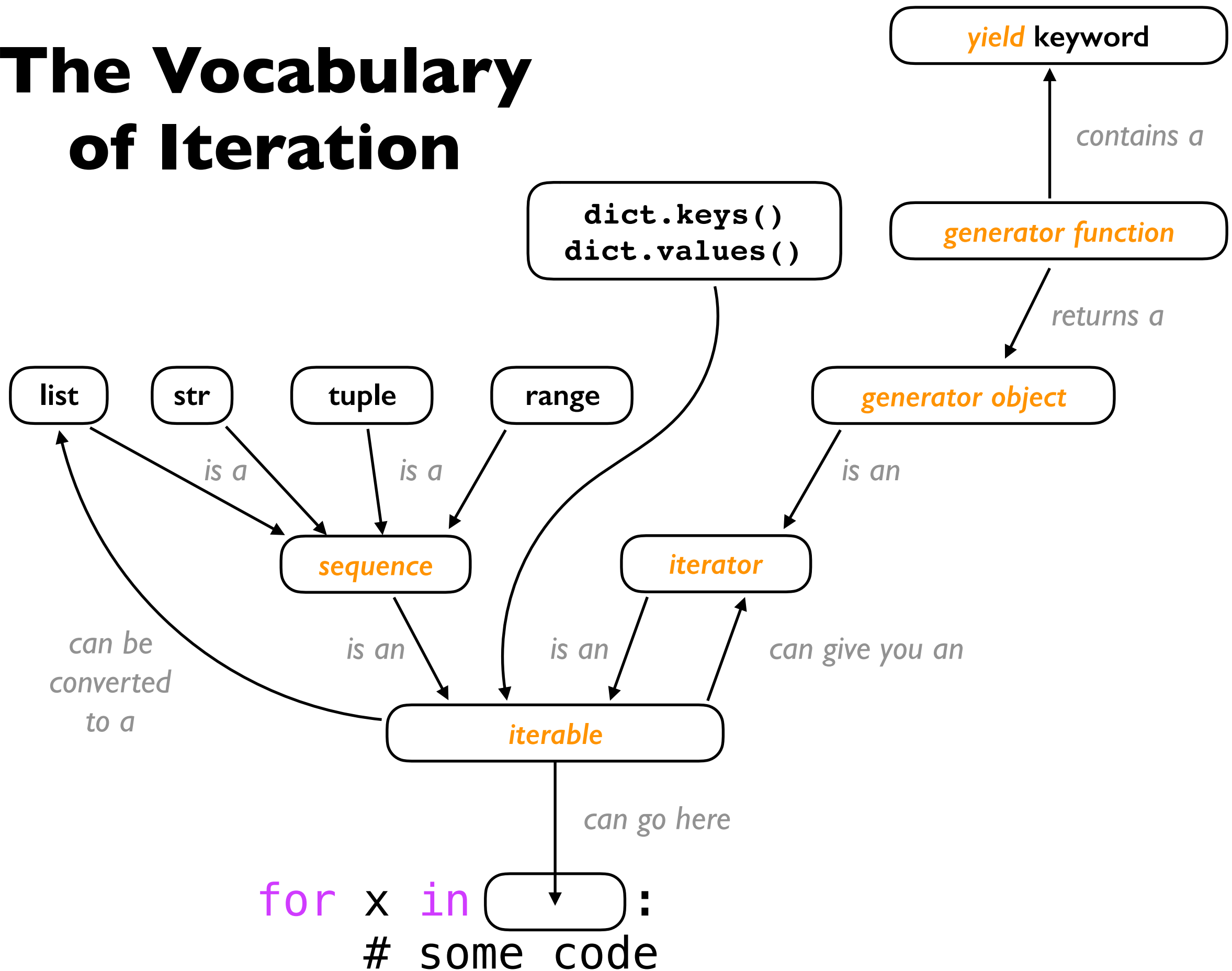
The Vocabulary of Iteration



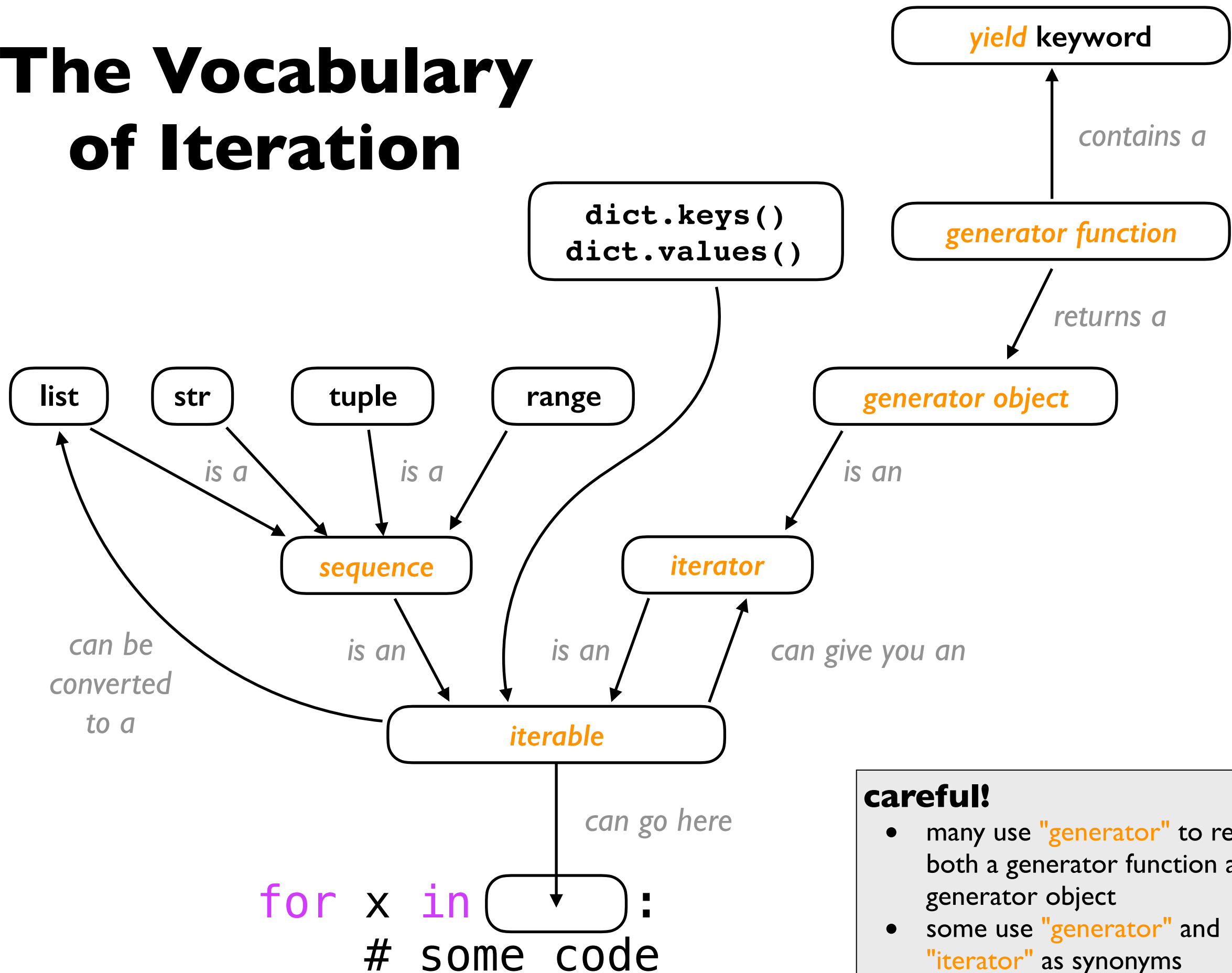
The Vocabulary of Iteration



The Vocabulary of Iteration



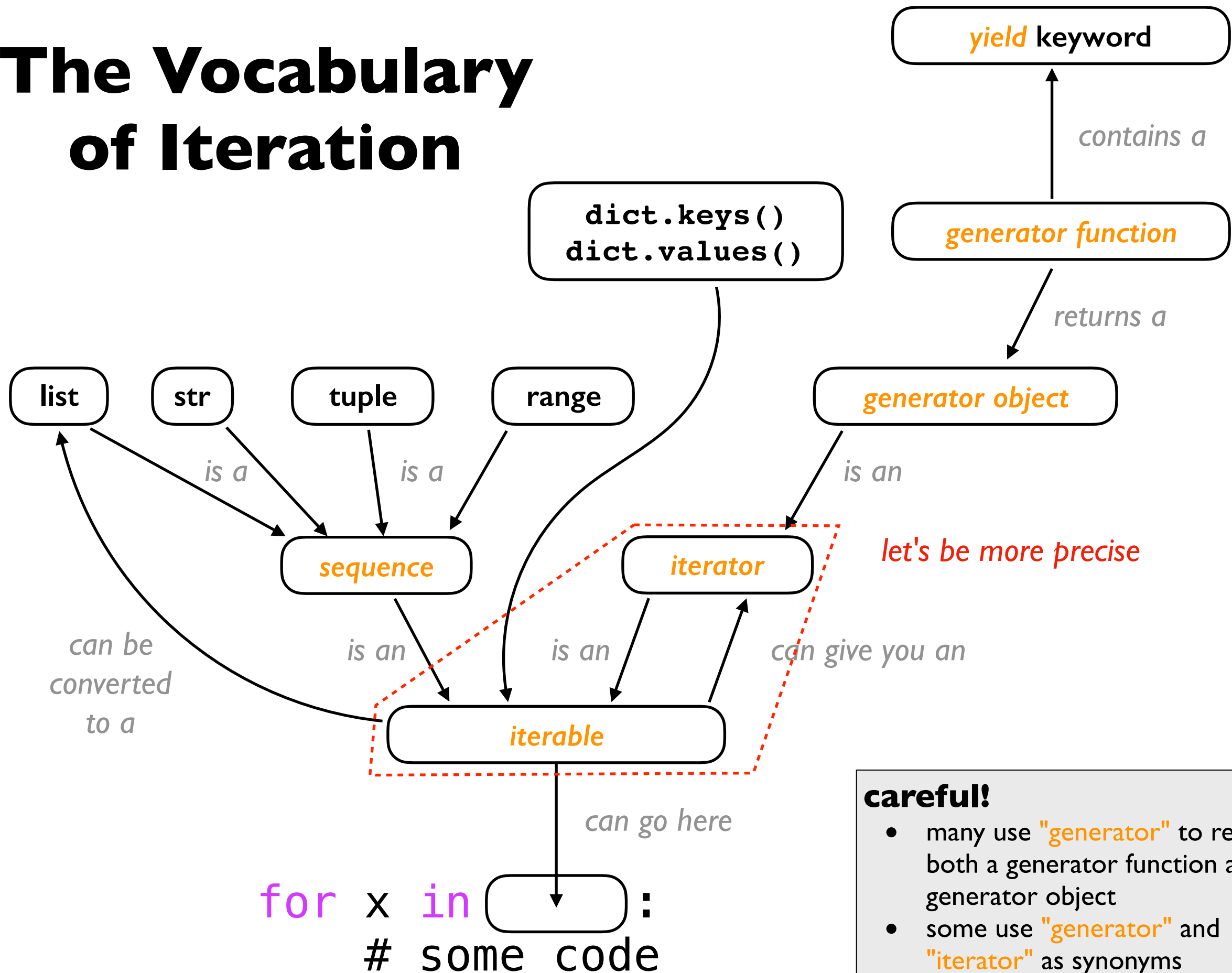
The Vocabulary of Iteration



careful!

- many use "generator" to refer to both a generator function and a generator object
- some use "generator" and "iterator" as synonyms

The Vocabulary of Iteration



careful!

- many use "generator" to refer to both a generator function and a generator object
- some use "generator" and "iterator" as synonyms

is `x` iterable?

if this works, then yes:

```
iter(x)
```

is `x` iterable?

if this works, then yes:

`iter(x)` **returns an iterator over `x`**

is `x` **iterable**?

if this works, then yes:

`iter(x)` returns an iterator over `x`

is `y` an **iterator**?

if this works, then yes:

`next(y)`

is `x` **iterable**?

if this works, then yes:

`iter(x)` **returns an iterator over `x`**

is `y` an **iterator**?

if this works, then yes:

`next(y)` **returns next value from `y`**

is `x` iterable?

if this works, then yes:

`y = iter(x)` returns an iterator over `x`

is `y` an iterator?

if this works, then yes:

`next(y)` returns next value from `y`

Can you classify x, y, and z?

1

```
>>> x = [1,2,3]
>>> next(x)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'list' object is not an iterator
>>> iter(x)
<list_iterator object at 0x1067bfb38>
```

2

```
>>> y = enumerate([1,2,3])
>>> next(y)
(0, 1)
```

3

```
>>> z = 3
>>> next(z)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'int' object is not an iterator
>>> iter(z)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'int' object is not iterable
```

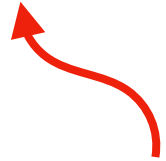

Iterators/Generators (Part 2)

Outline

- when normal functions aren't good enough
- yield keyword by example
- the scary vocabulary of iteration
- the open function
- demos

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 objects are iterators!

file.txt

This is a test!

3

2

1

Go!

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!

Iterators/Generators (Part 2)

Outline

- when normal functions aren't good enough
- yield keyword by example
- the scary vocabulary of iteration
- the open function
- demos

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

Demo 2: handy functions

Learn these:

- enumerate
- zip

Demo 3: sorting files by line length

Goal: output file contents, with shortest line first

Input:

- a text file

Output:

- print lines sorted

Demo 4: matrix load

Goal: load a matrix of integers from a file

Input:

- file name

Output:

- generator that yields lists of ints

