[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 references
- ...

Function References (Part 1)

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:

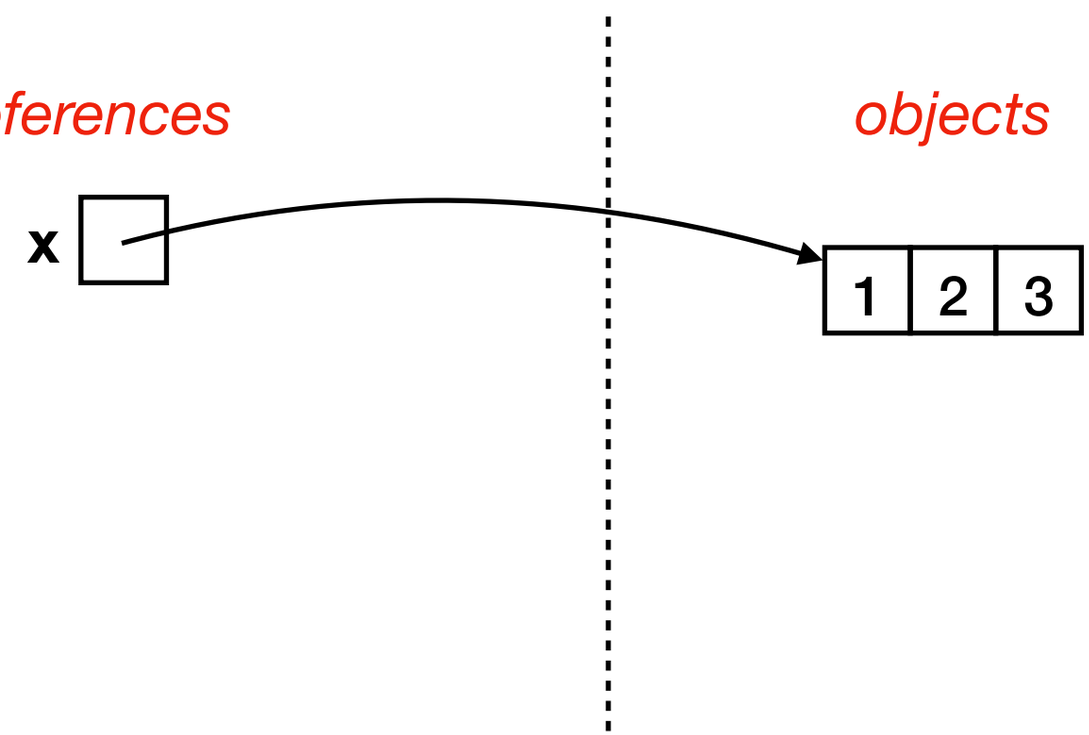
references

objects

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

def f():
    return "hi"

g = f

z = f()
```

Explanation: x should reference a new list object

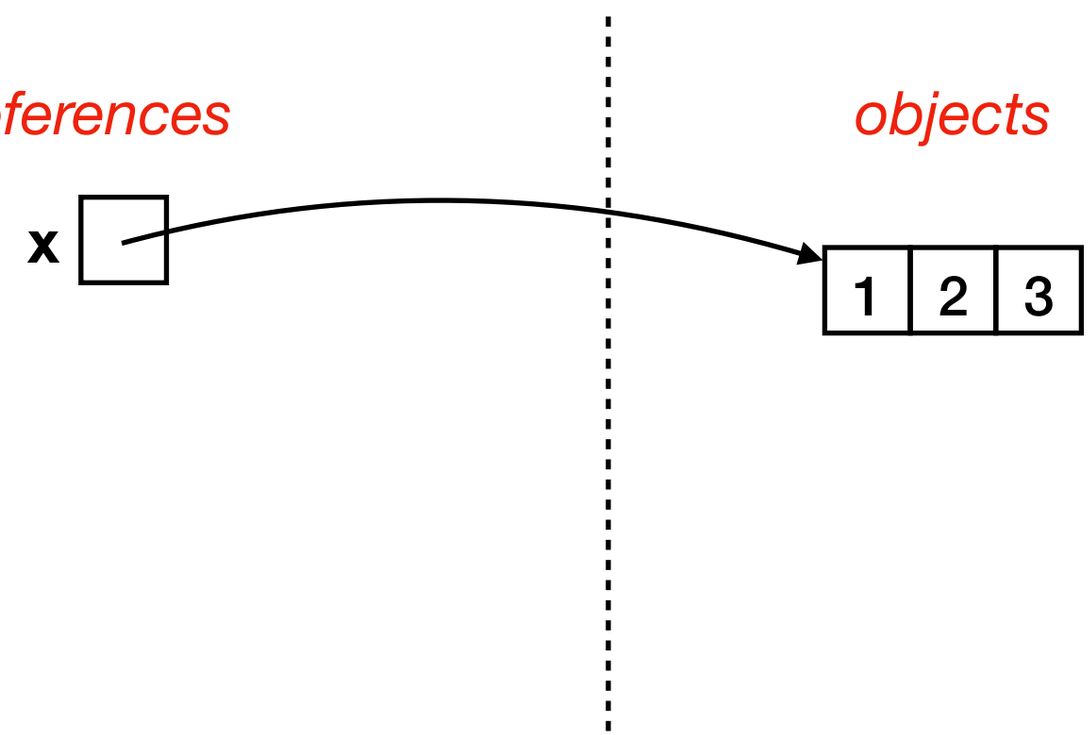
State:



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

def f():
    return "hi"

g = f

z = f()
```

Explanation: x should reference a new list object

State:

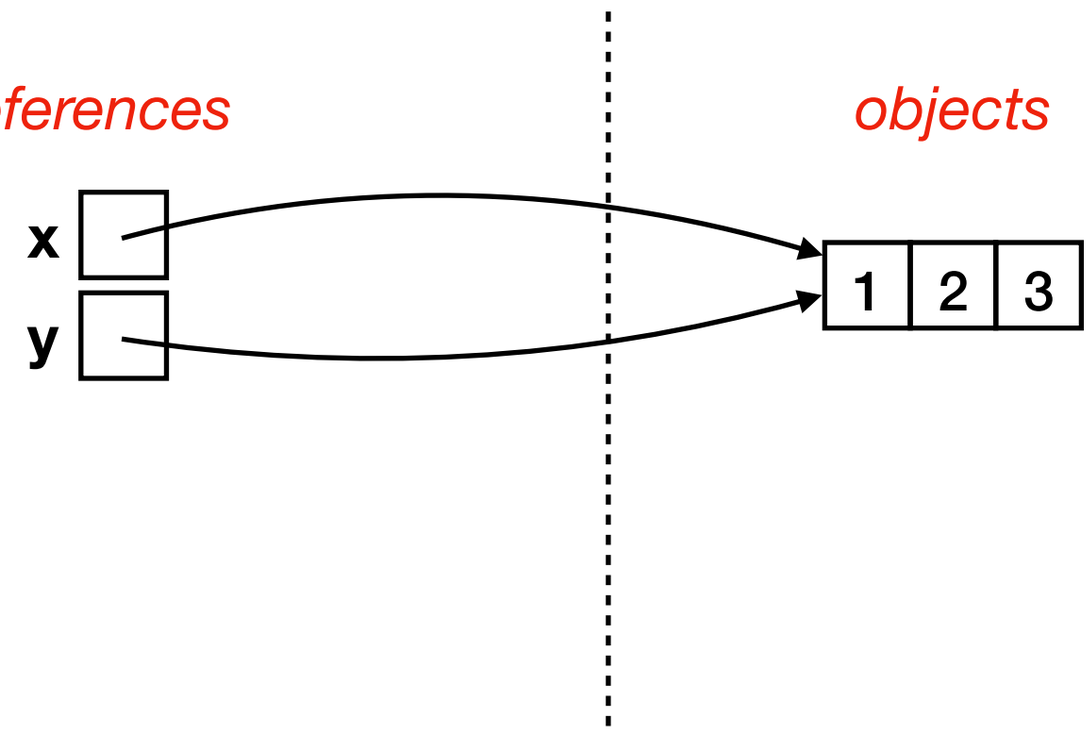


```
x = [1,2,3]
y = x

def f():
    return "hi"

g = f

z = f()
```

Explanation: x should reference a new list object

Explanation: y should reference whatever x references

State:

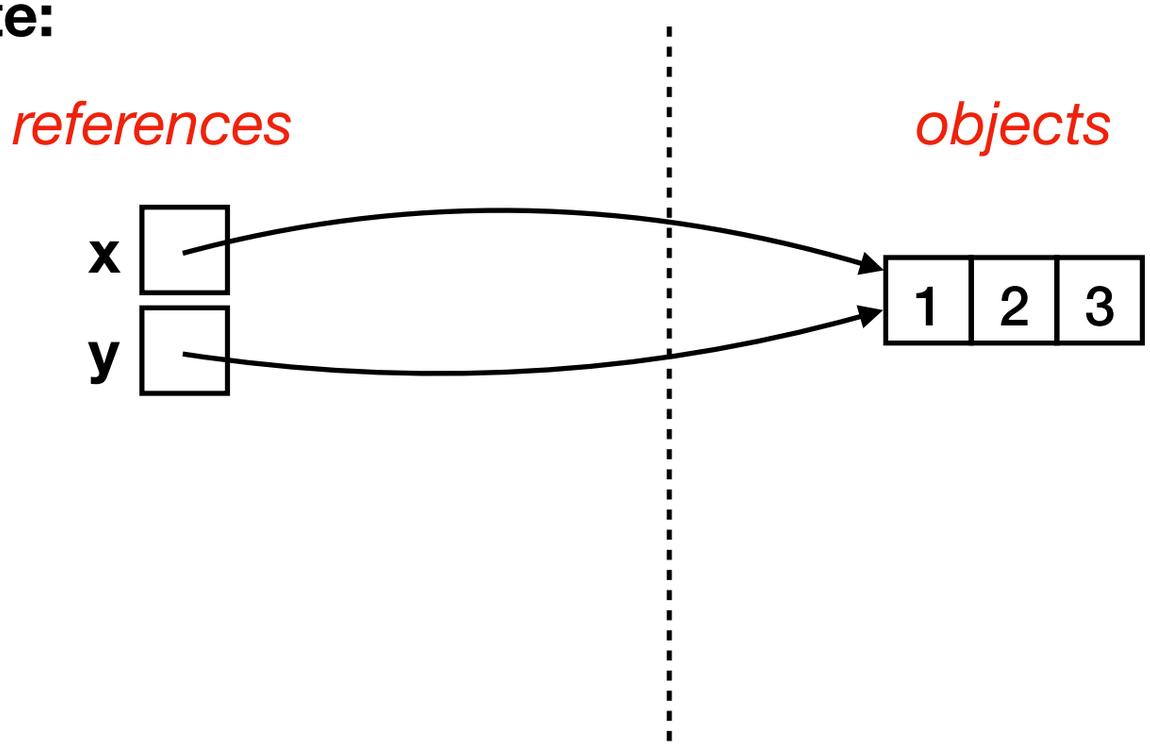


```
x = [1,2,3]
y = x

def f():
    return "hi"

g = f

z = f()
```

Explanation: x should reference a new list object

Explanation: y should reference whatever x references

State:



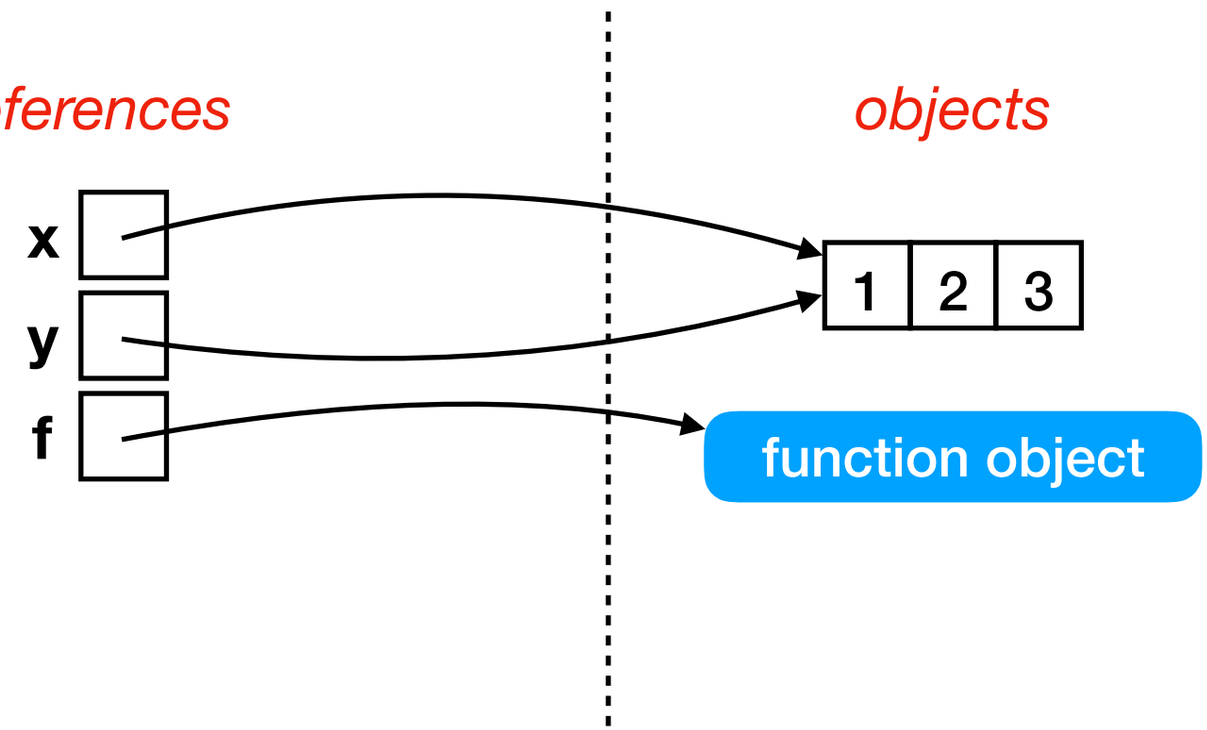
```
x = [1,2,3]
y = x

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:



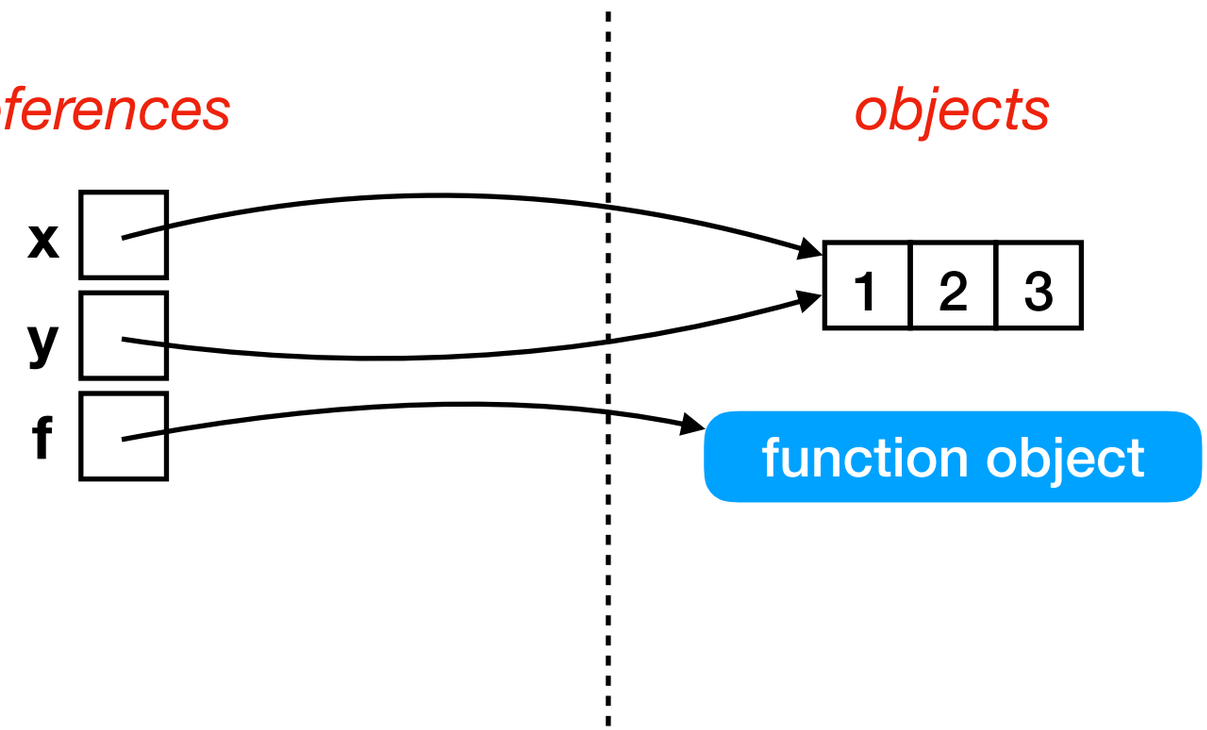
```
x = [1,2,3]
y = x

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()
```

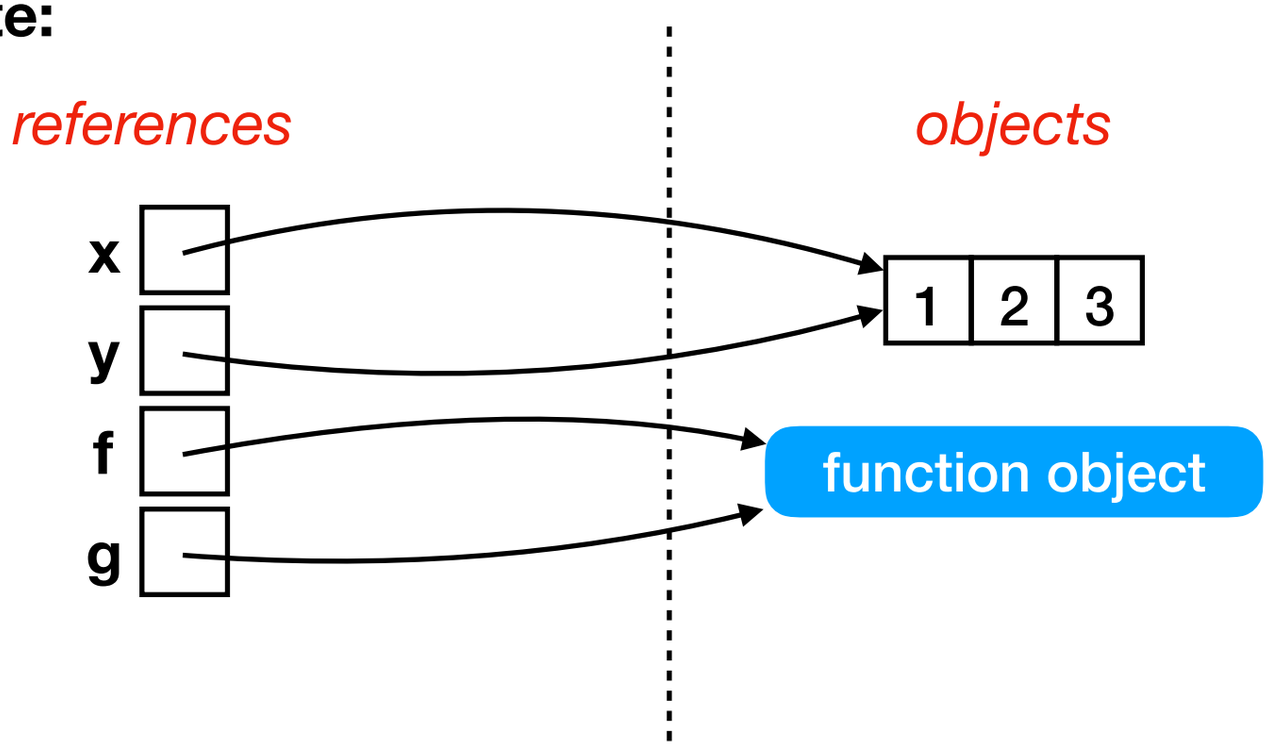
Explanation: x should reference a new list object

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()
```

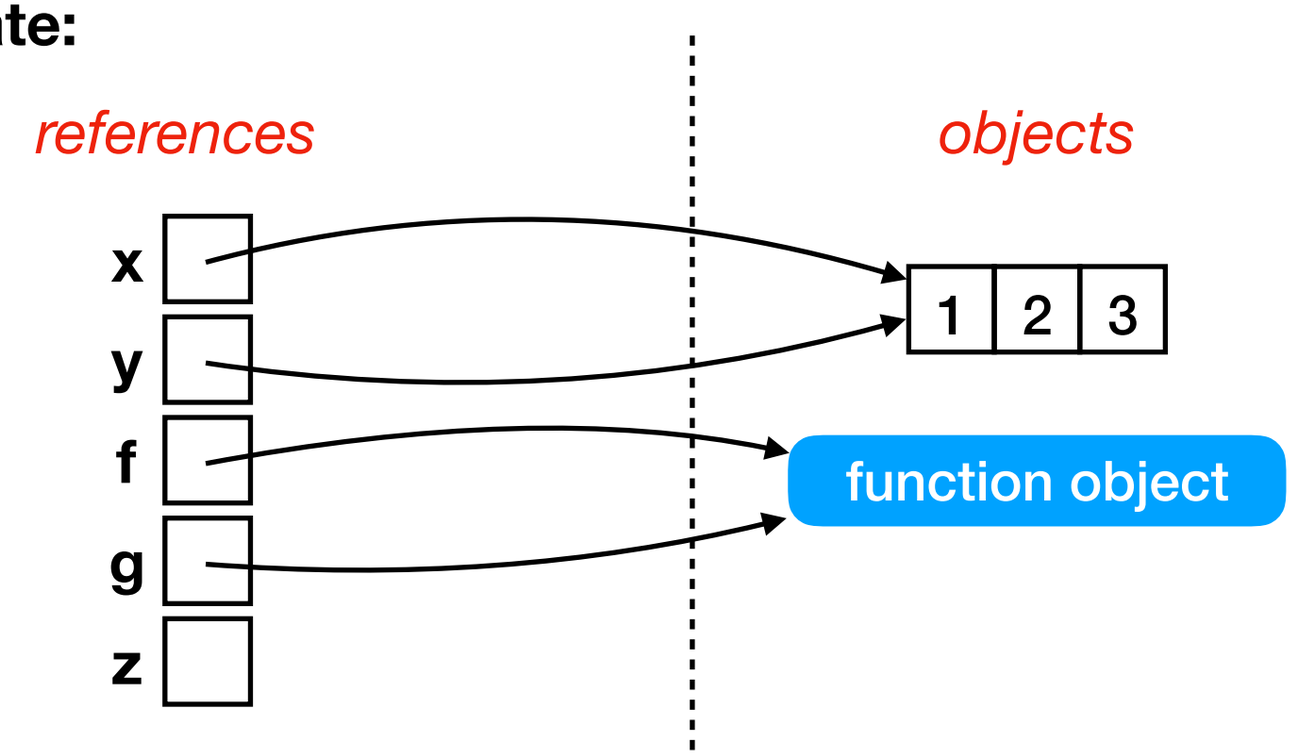
Explanation: x should reference a new list object

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

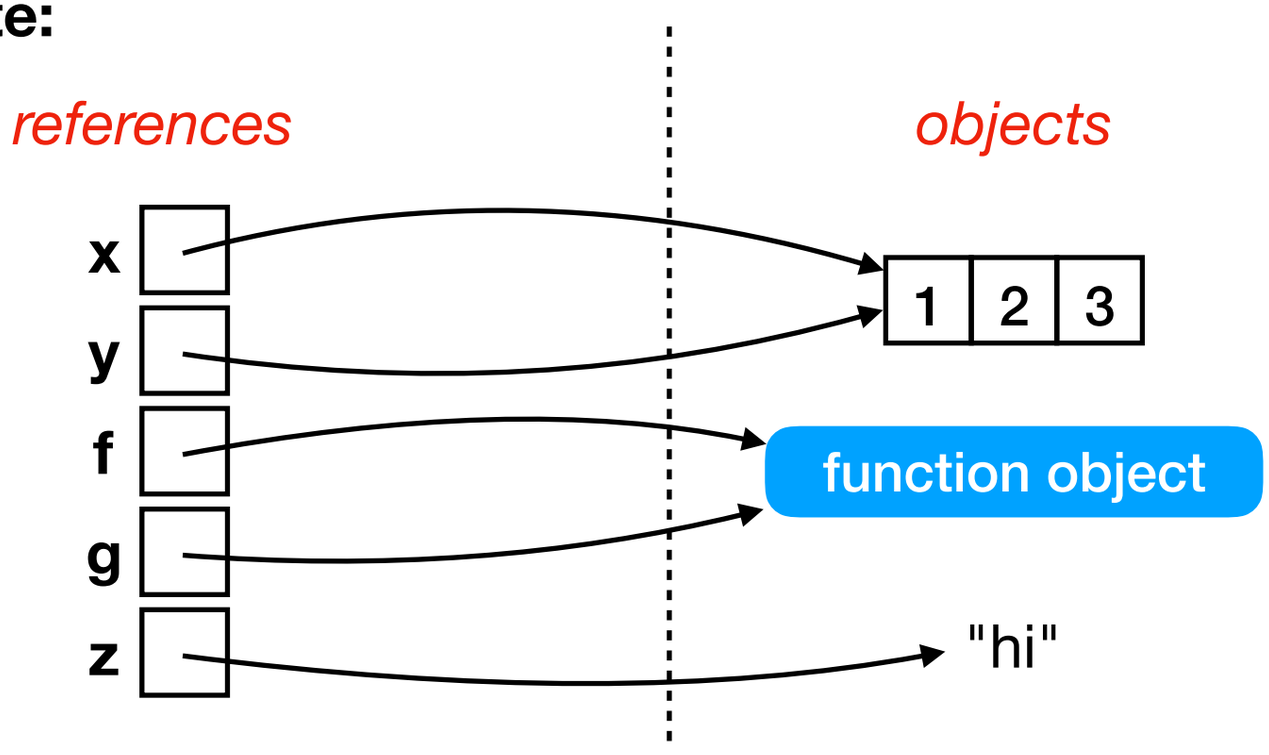
Explanation: y should reference whatever x references

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()
```

Explanation: x should reference a new list object

Explanation: y should reference whatever x references

Explanation: f should reference a new function object

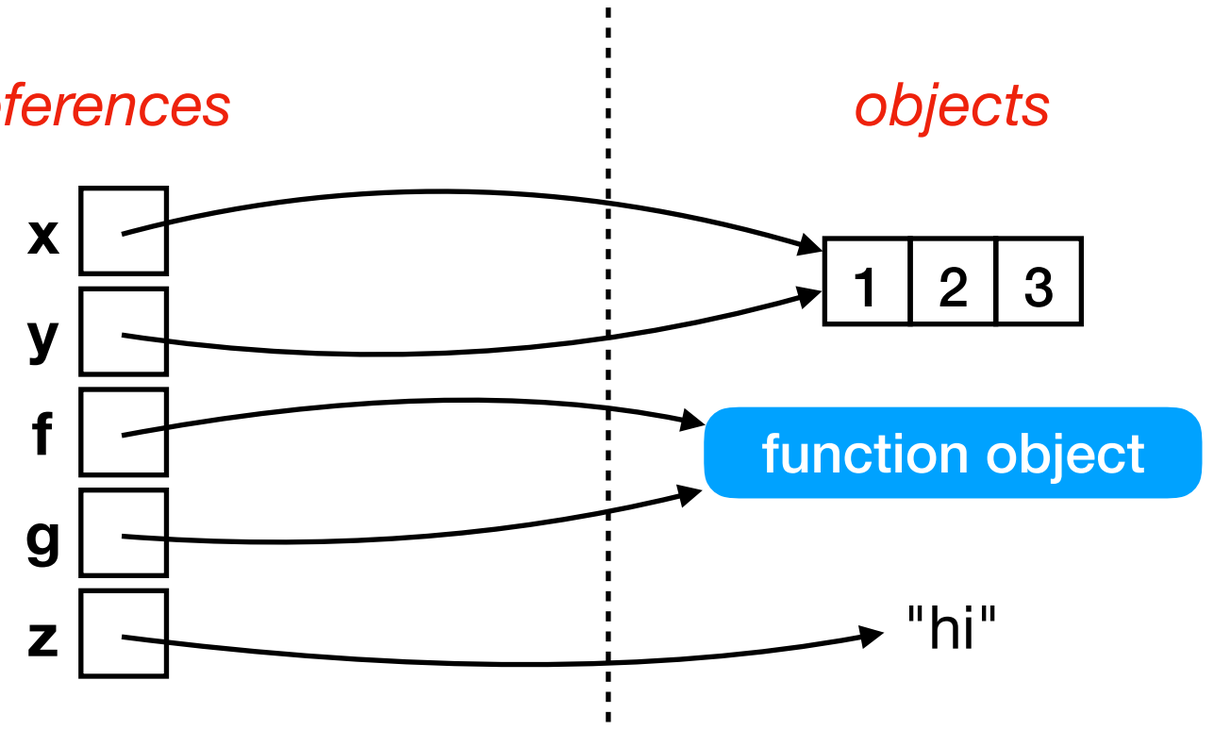
Explanation: g should reference whatever f references

Explanation: z should reference whatever f returns

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

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

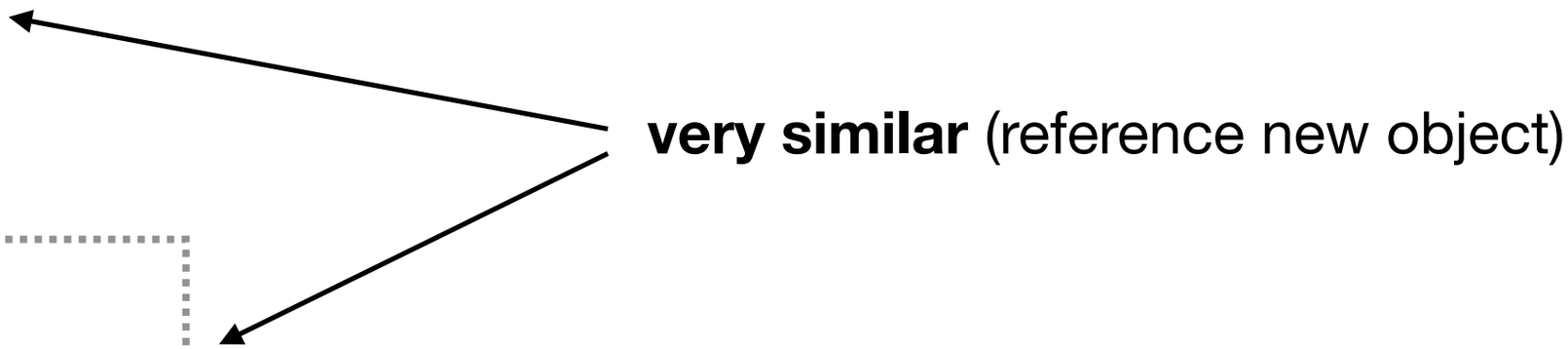
State:



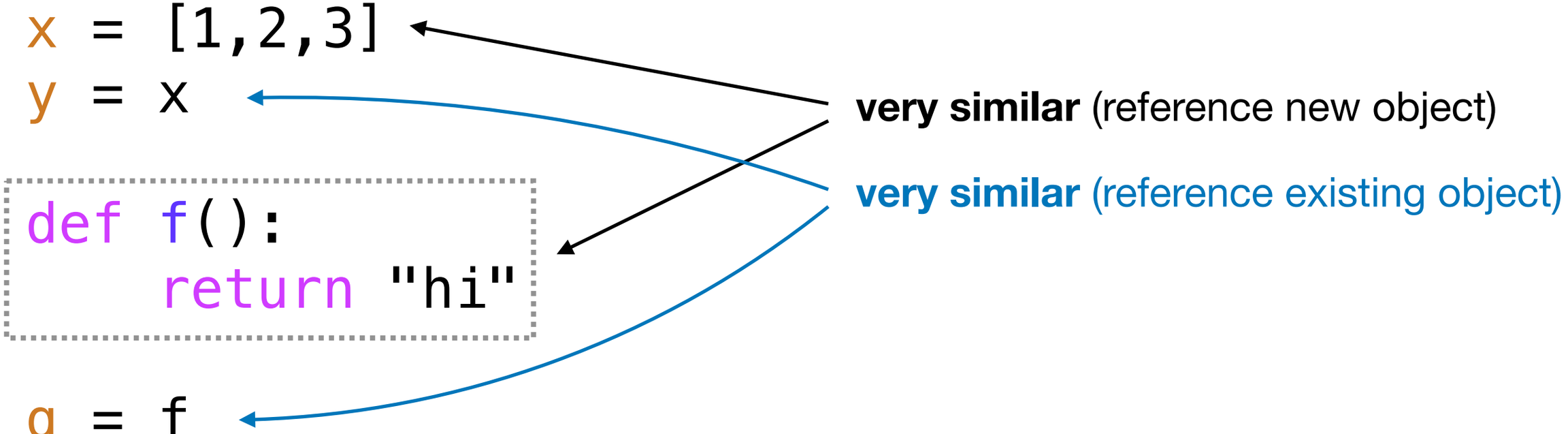
```
x = [1,2,3]
y = x
```

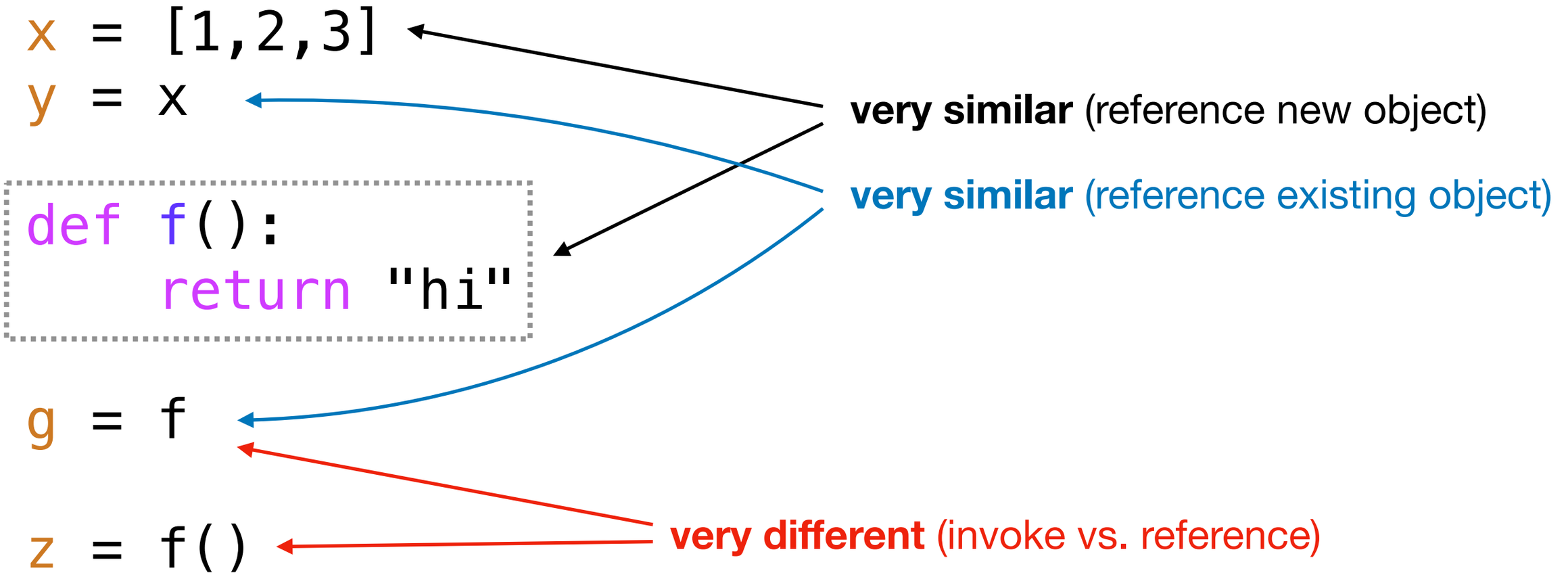
```
def f():
    return "hi"
```

```
g = f

z = f()
```



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

def f():
    return "hi"

g = f

z = f()
```




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)

CODING DEMOS

(Python Tutor)

Function References (Part 1)

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”),
]
```

Catherine	Baker
Bob	Adams
Alice	Clark

```
names.sort()
```

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

Alice	Clark
Bob	Adams
Catherine	Baker

Example: Sorting Names

List of tuples:

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

Catherine	Baker
Bob	Adams
Alice	Clark

```
names.sort()
```

what if we want to sort by the last name?

Alice	Clark
Bob	Adams
Catherine	Baker

Example: Sorting Names

List of tuples:

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

Catherine	Baker
Bob	Adams
Alice	Clark

```
names.sort()
```

what if we want to sort by the last name?

or by the length of the name?

Alice	Clark
Bob	Adams
Catherine	Baker

Example: Sorting Names

List of tuples:

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

Catherine	Baker
Bob	Adams
Alice	Clark

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

names.sort(key=extract)
```

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”),
]
```

Catherine	Baker
Bob	Adams
Alice	Clark

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

names.sort(key=extract)
```

Example: Sorting Names

List of tuples:

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

Catherine	Baker
Bob	Adams
Alice	Clark

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

names.sort(key=extract)
```

Bob	Adams
Alice	Clark
Catherine	Baker

[301] Advanced Functions

Tyler Caraza-Harter

1. Functions as Objects
2. **Iterators/Generators**

Iterators/Generators (Part 2)

Outline

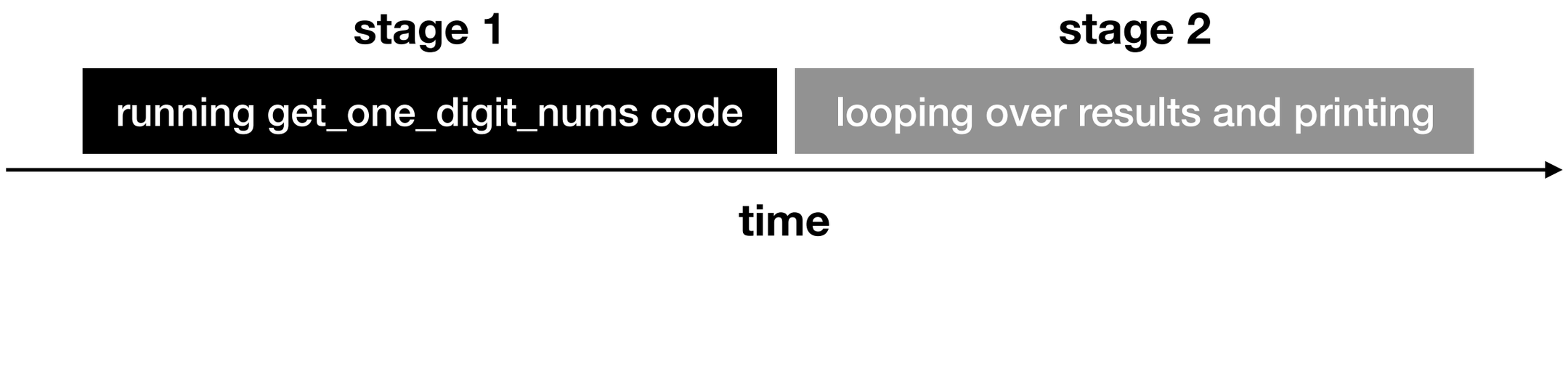
- when normal functions aren't good enough
- yield keyword by example
- the scary vocabulary of iteration
- the open function
- demos

```
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)
```



```
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

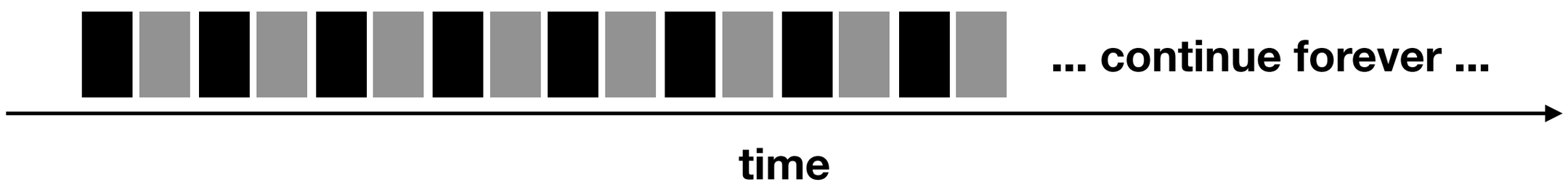
```
def get_primes():
    print("START")
    nums = []
    i = 0
    while True:
        if is_prime(i):
            nums.append(i)
        i += 1
    print("END")
    return nums
```

```
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) 
```

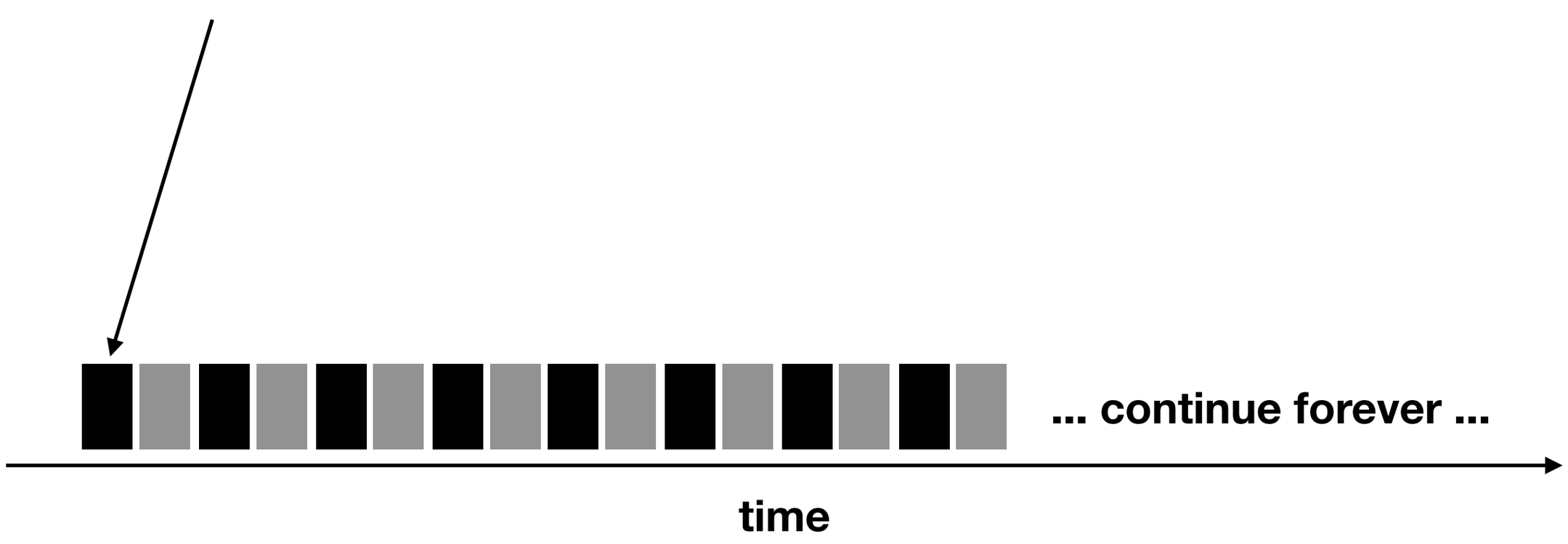
what we want:



```
def get_primes():
    ... ■

for x in get_primes():
    print(x) ■
```

run `get_primes` just long enough to get one prime

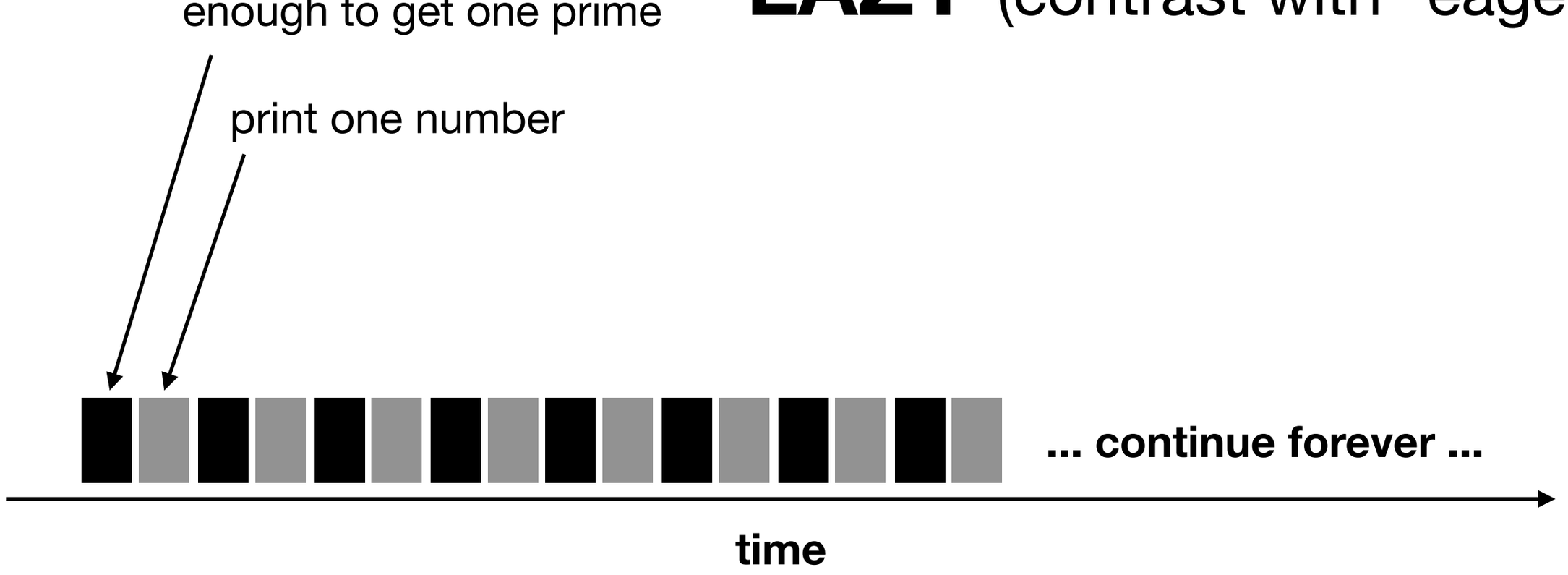
LAZY (contrast with "eager")



```
def get_primes():
    ... ■

for x in get_primes():
    print(x) ■
```

run `get_primes` just long enough to get one prime

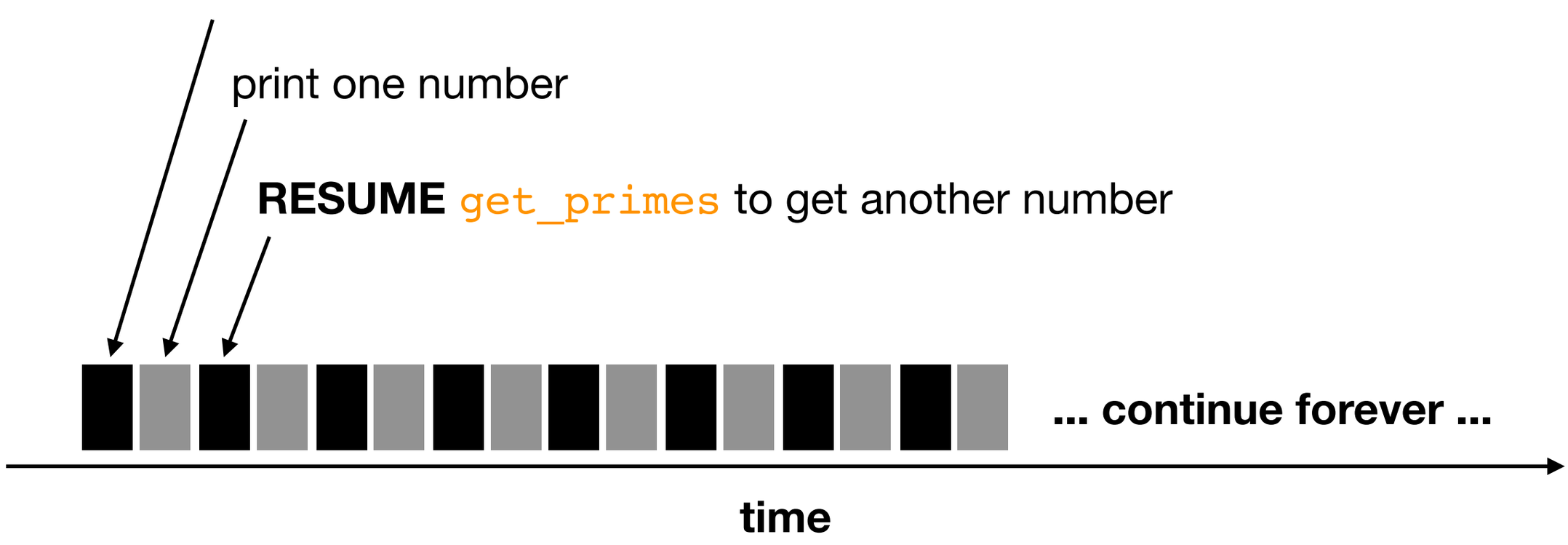
LAZY (contrast with "eager")



```
def get_primes():
    ... ■

for x in get_primes():
    print(x) ■
```

run `get_primes` just long enough to get one prime

LAZY (contrast with "eager")

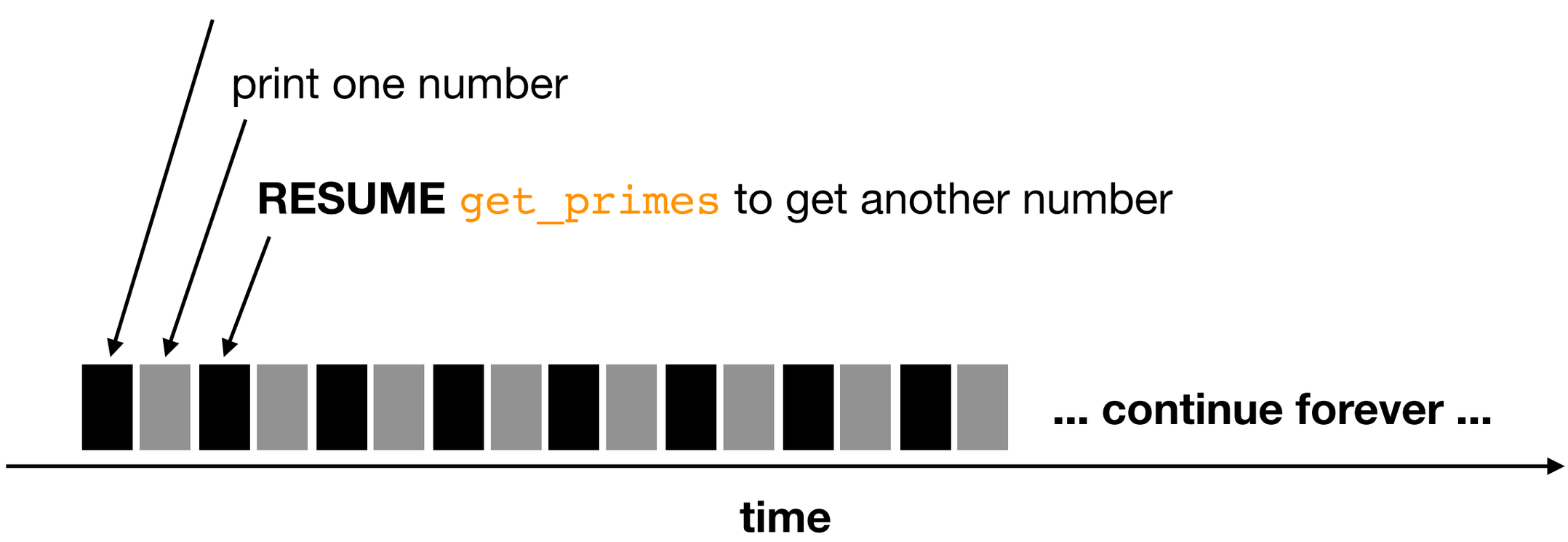


```
def get_primes():
    ...

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

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")



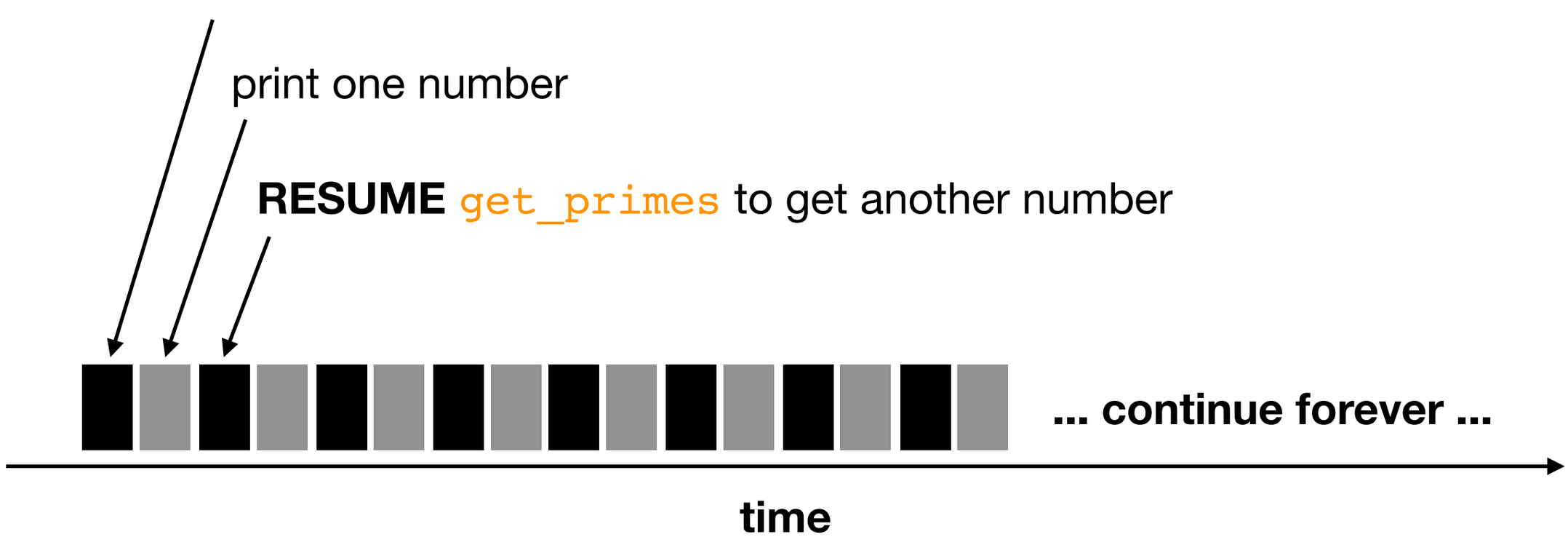
```
def get_primes():
    ... ■

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

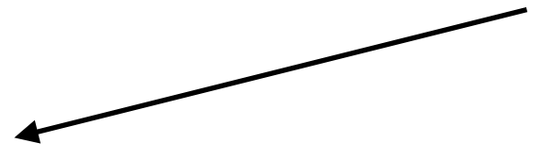
LAZY (contrast with "eager")



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

    yield VALUE

    ... more code ...
```

any function containing the yield keyword anywhere is a generator

```
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?

Guido van Rossum
Python's Benevolent Dictator for Live
(until recently)

https://www.python.org/dev/peps/pep-0255/#bdfl-pronouncements

?

```
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?

Guido van Rossum
Python's Benevolent Dictator for Live
(until recently)

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

https://www.python.org/dev/peps/pep-0255/#bdfl-pronouncements

?

```
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?

Guido van Rossum
Python's Benevolent Dictator for Live
(until recently)

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*"

https://www.python.org/dev/peps/pep-0255/#bdfl-pronouncements

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

    yield VALUE

    ... more code ...
```

always scan a function for yields when trying to understand it

Guido van Rossum
Python's Benevolent Dictator for Live
(until recently)

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*"

https://www.python.org/dev/peps/pep-0255/#bdfl-pronouncements

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

```
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)
```

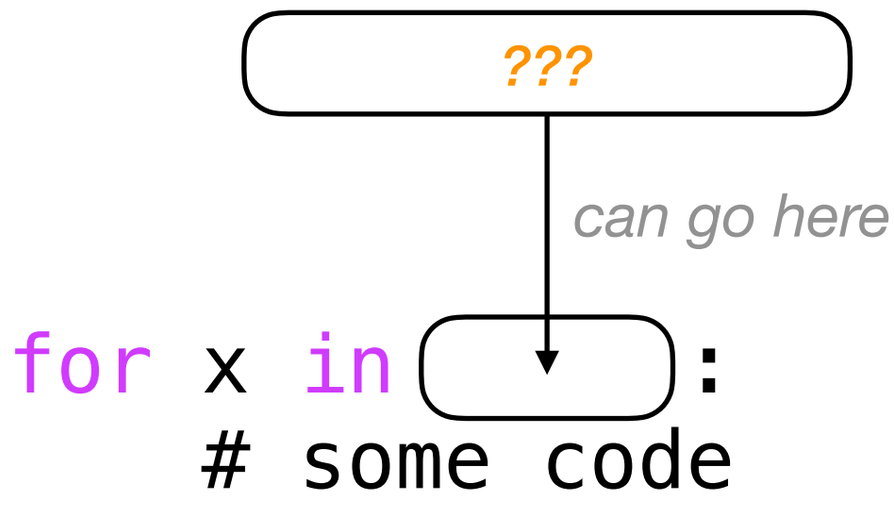
```
def f():
    yield 1
    yield 2
    yield 3

gen = f()
for x in g:
    print(x, y)
```

Iterators/Generators (Part 2)

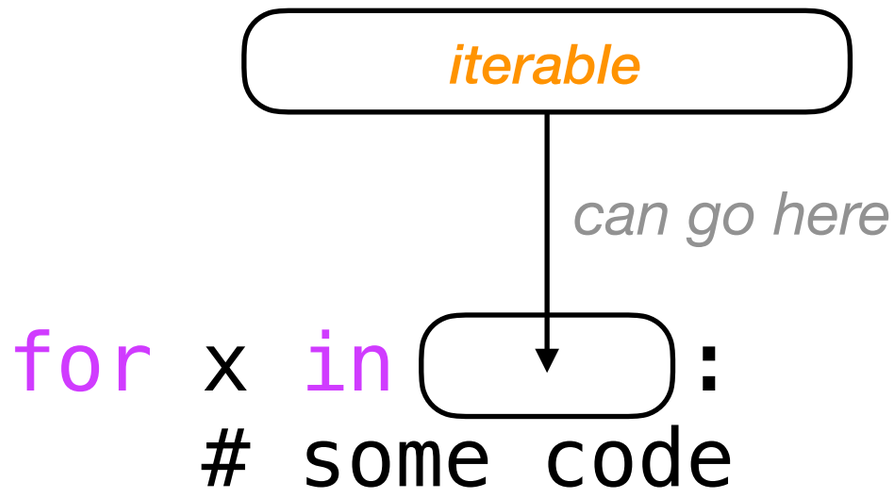
Outline

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

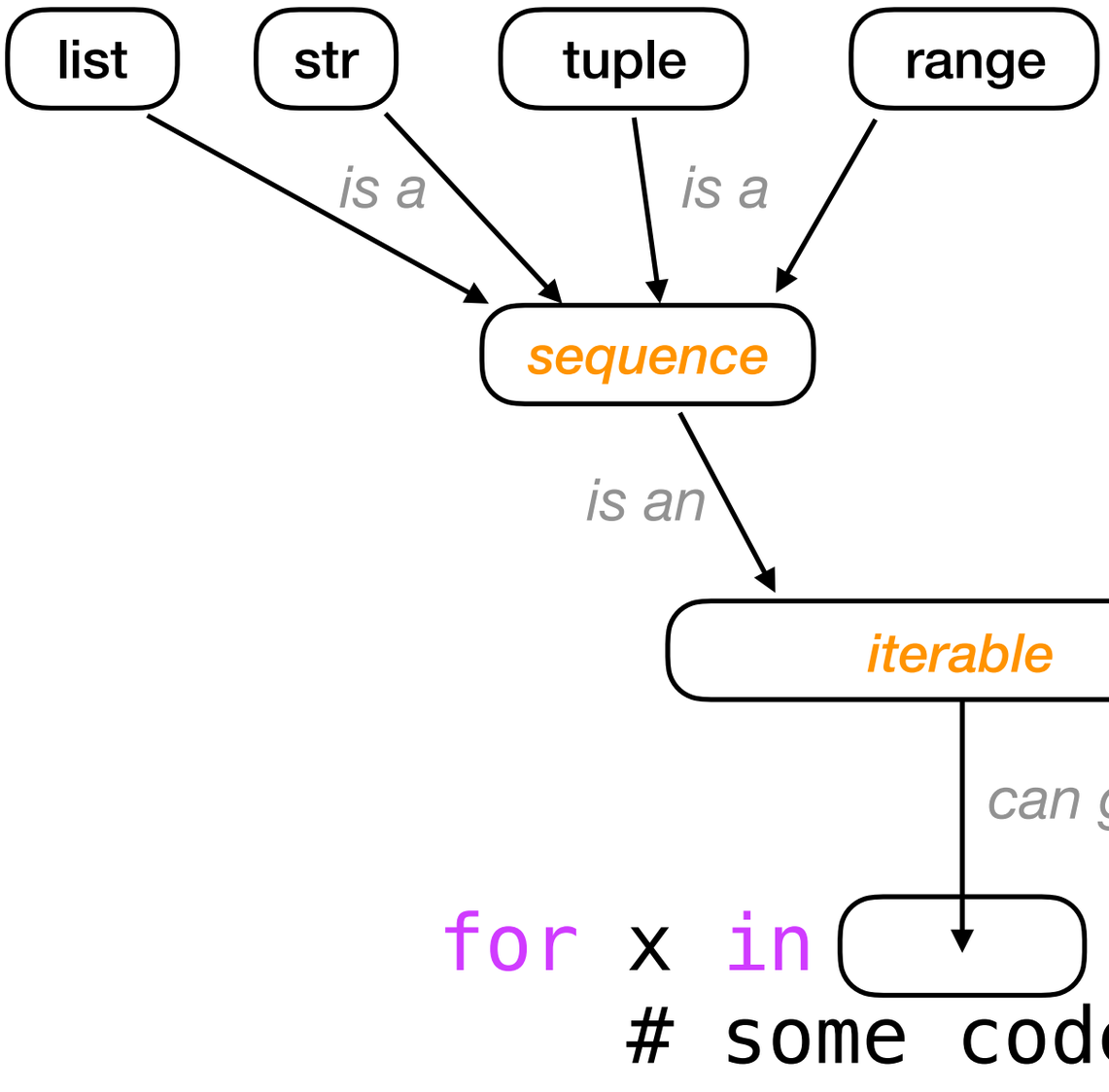
The Vocabulary of Iteration



The Vocabulary of Iteration

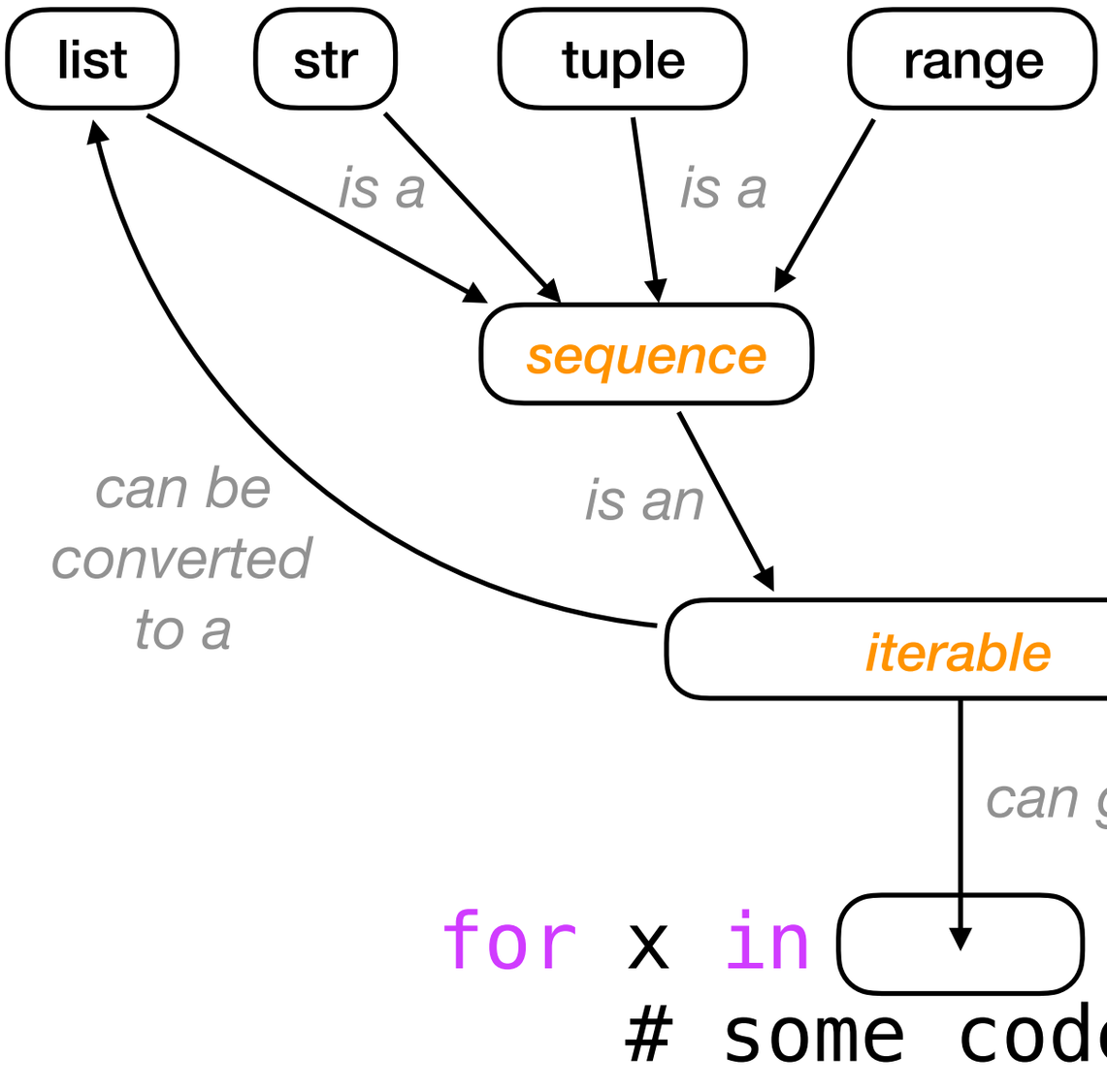


The Vocabulary of Iteration



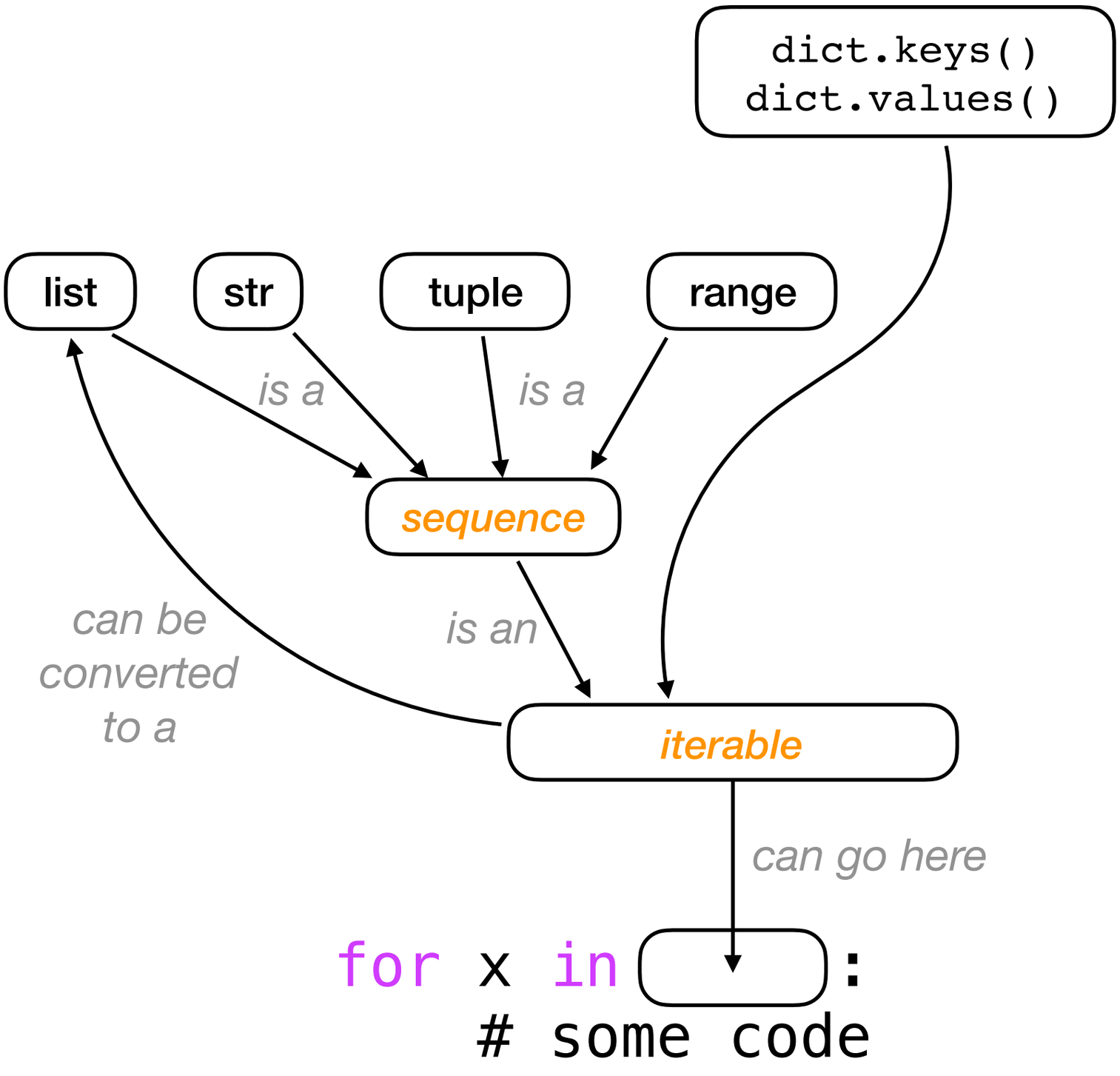

```
for x in      :
    # some code
```

The Vocabulary of Iteration

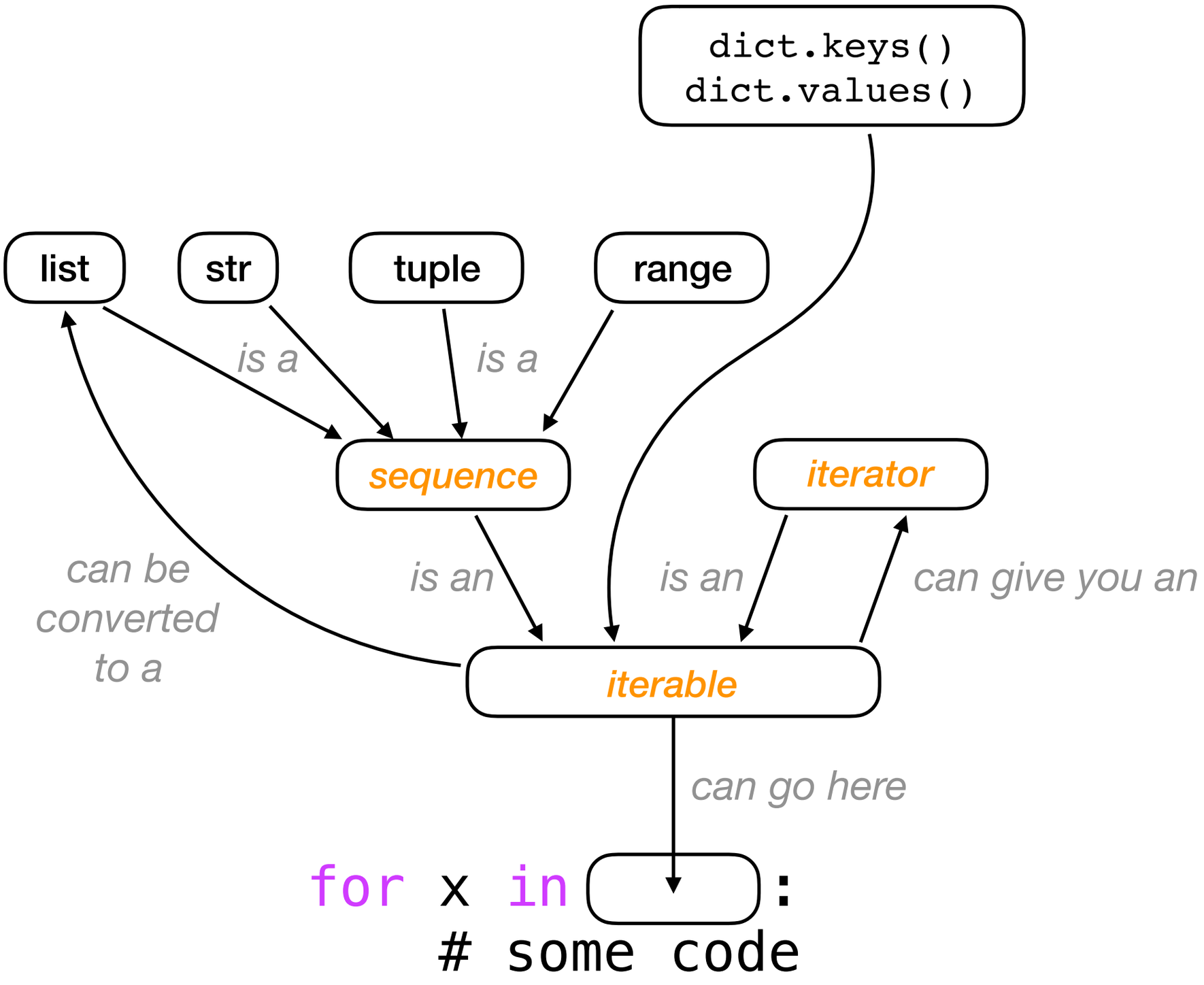



```
for x in      :
    # some code
```

The Vocabulary of Iteration

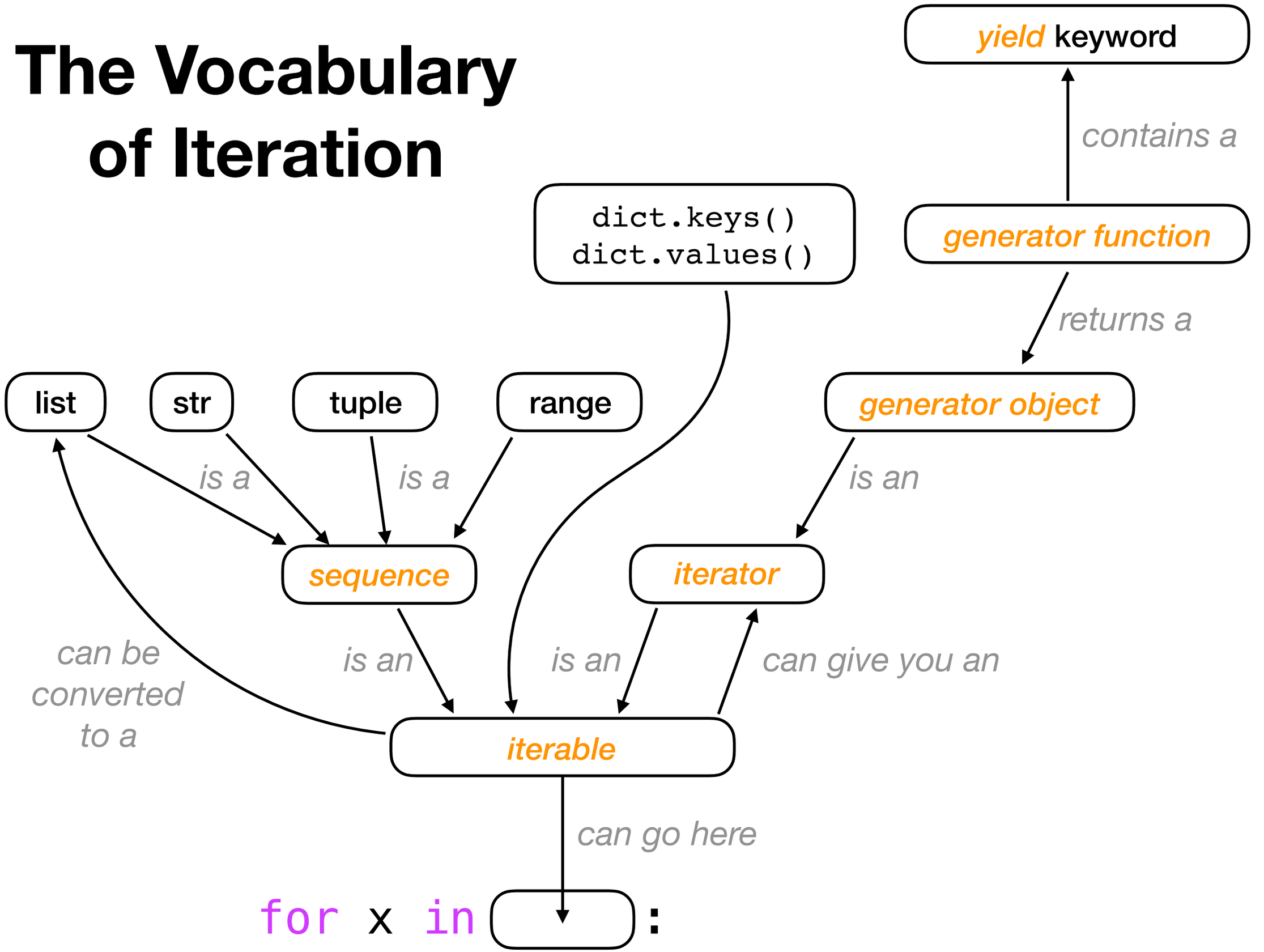


The Vocabulary of Iteration

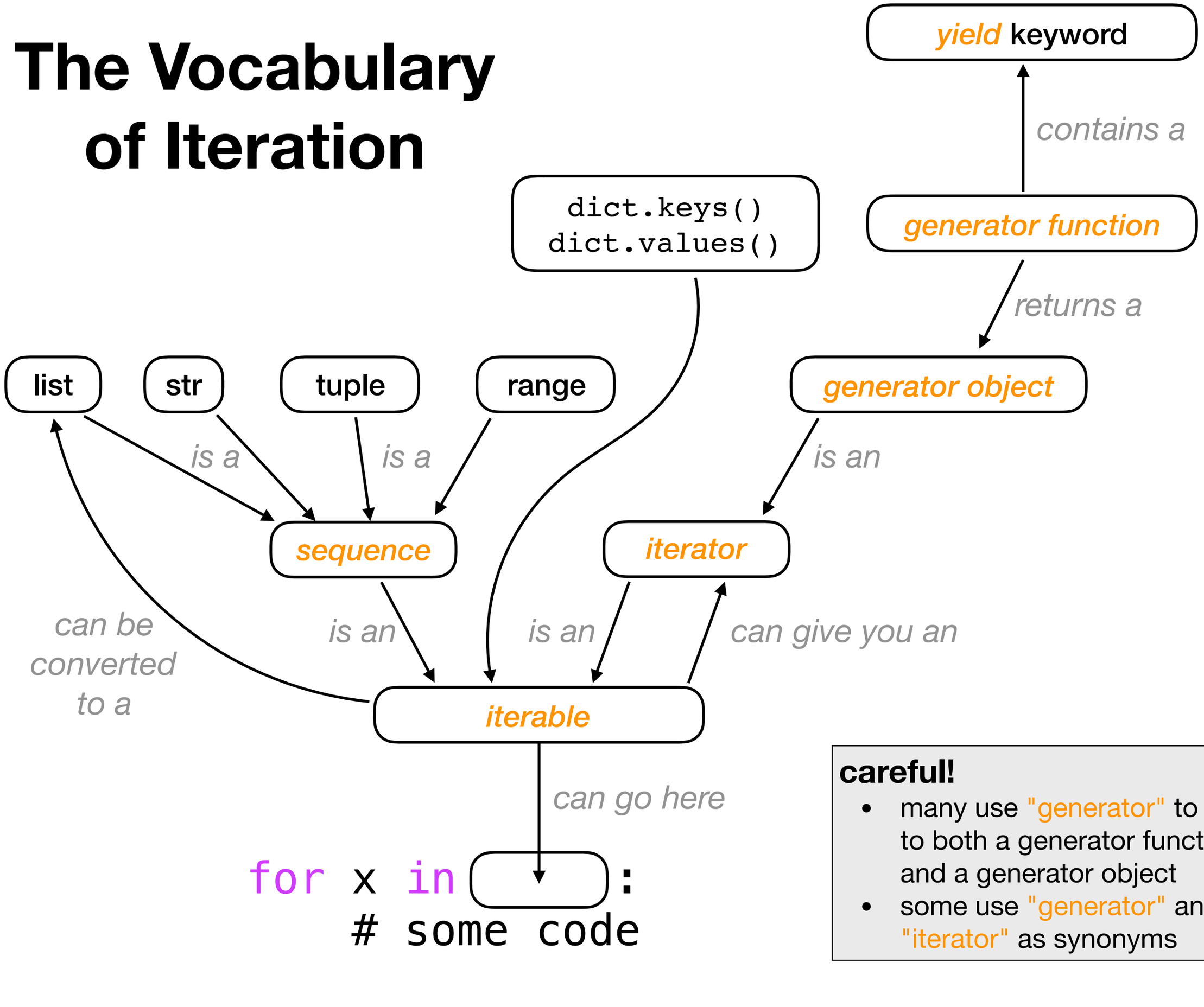



```
for x in      :
    # some code
```

The Vocabulary of Iteration



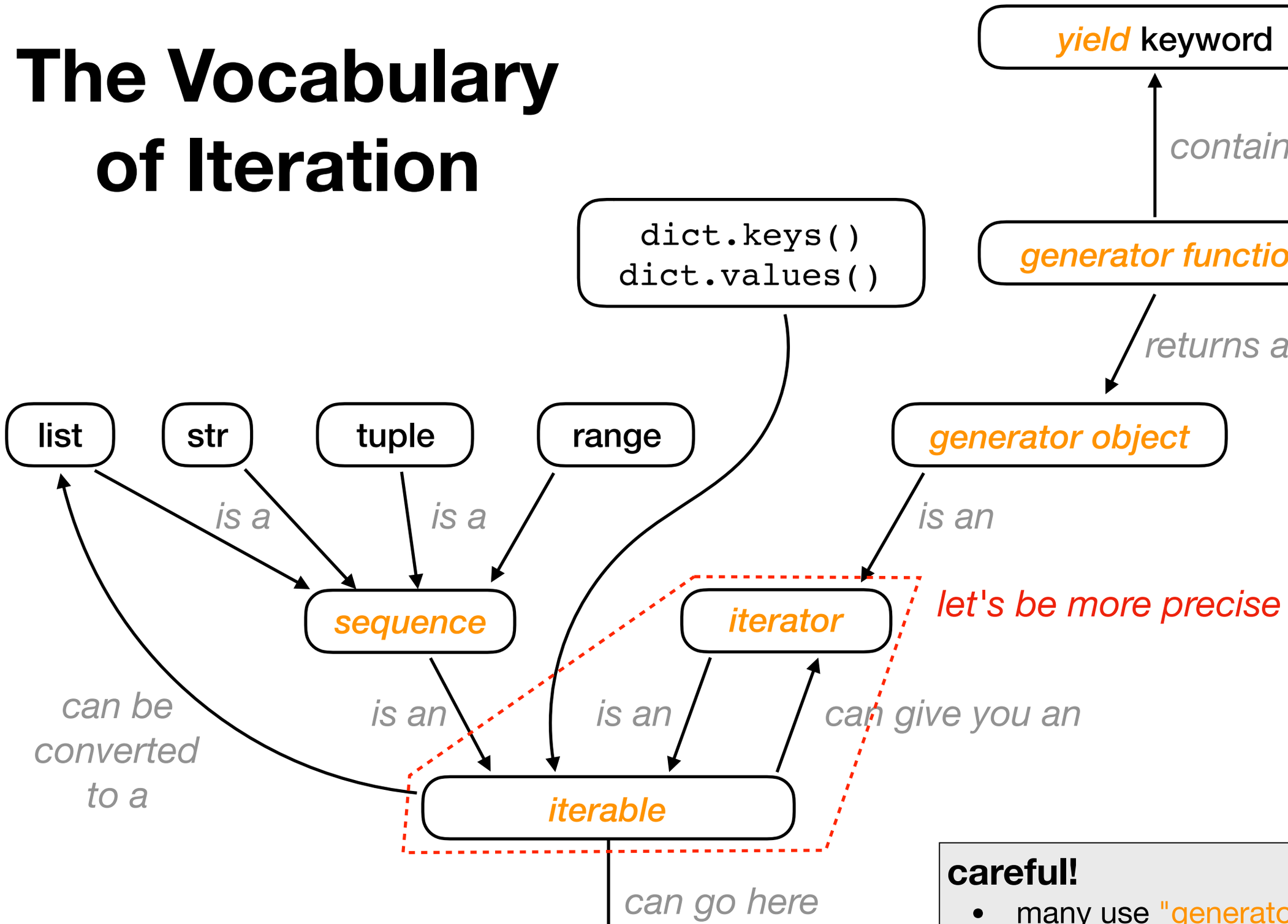
The Vocabulary of Iteration



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



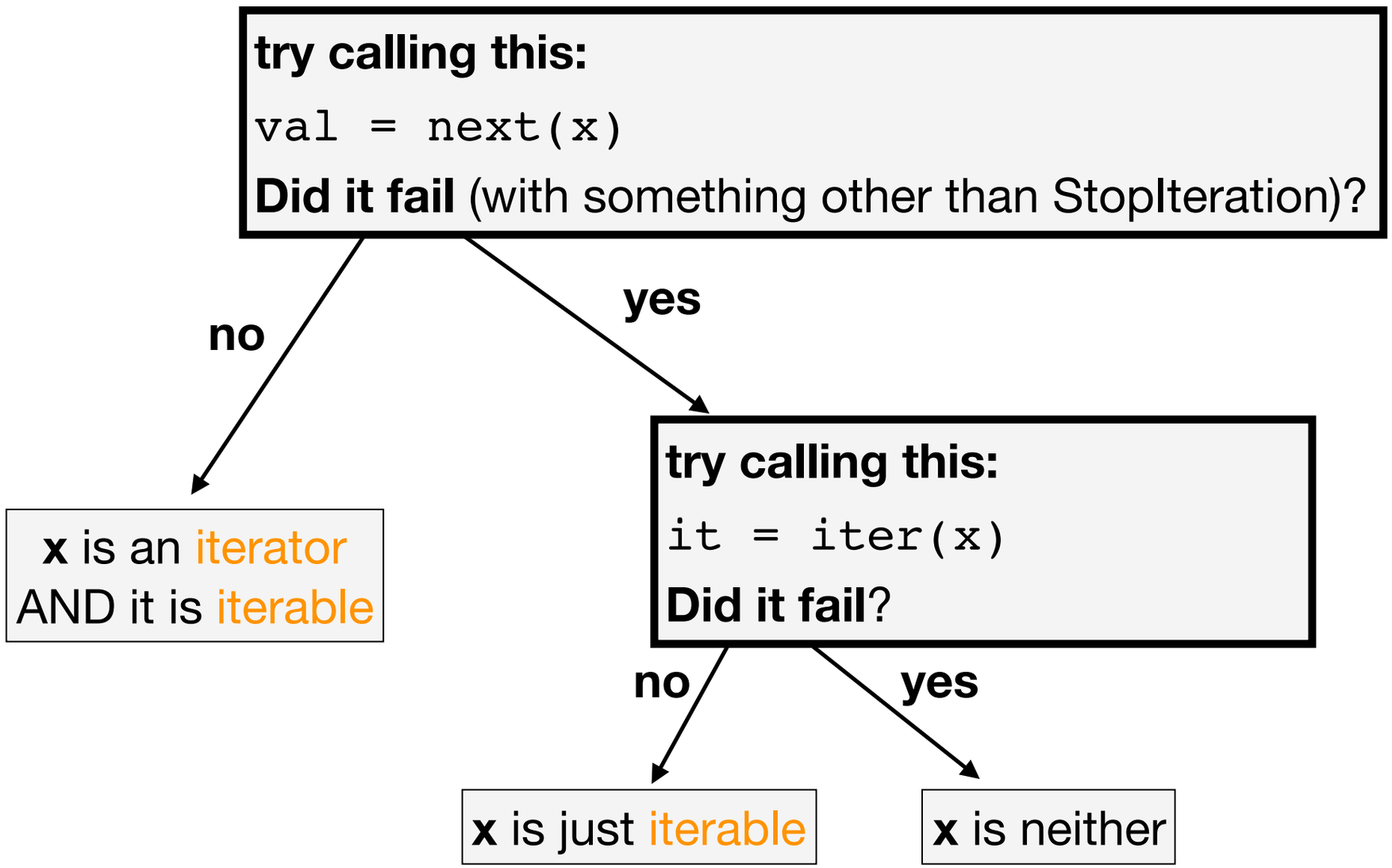
careful!

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

Say **x** references an object

- Is the object an iterator?
- Is the object iterable?

We can use the `iter()` and `next()` function to find out.



Can you classify x, y, and z?

1

```
>>> 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)
```

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 iterators!

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 £ 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

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

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:

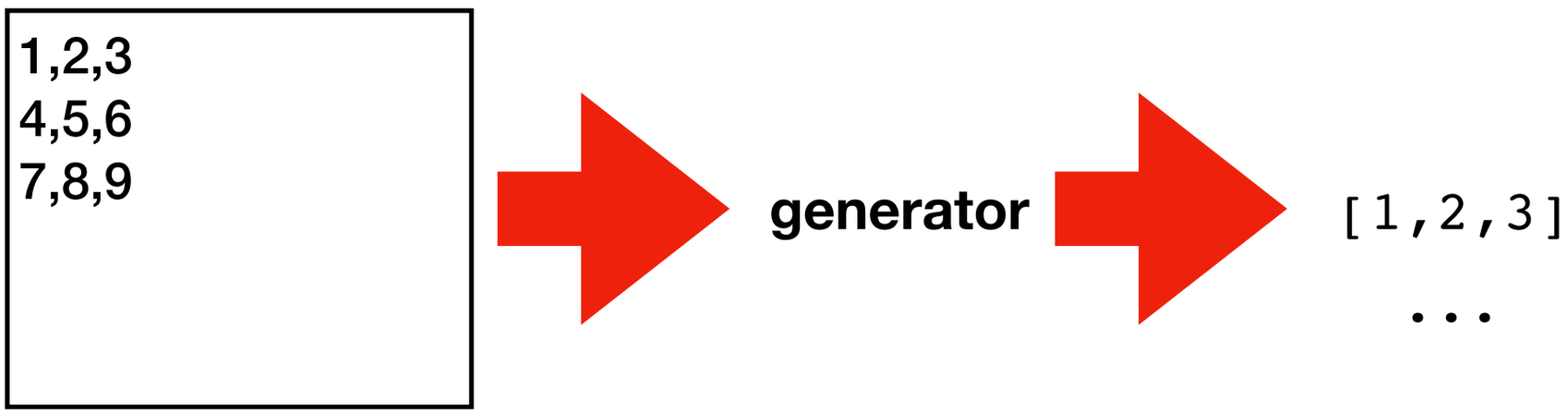
- print lines sorted

Demo 4: matrix load

Goal: load a matrix of integers from a file

Input:

- file name

Output:

- generator that yields lists of ints