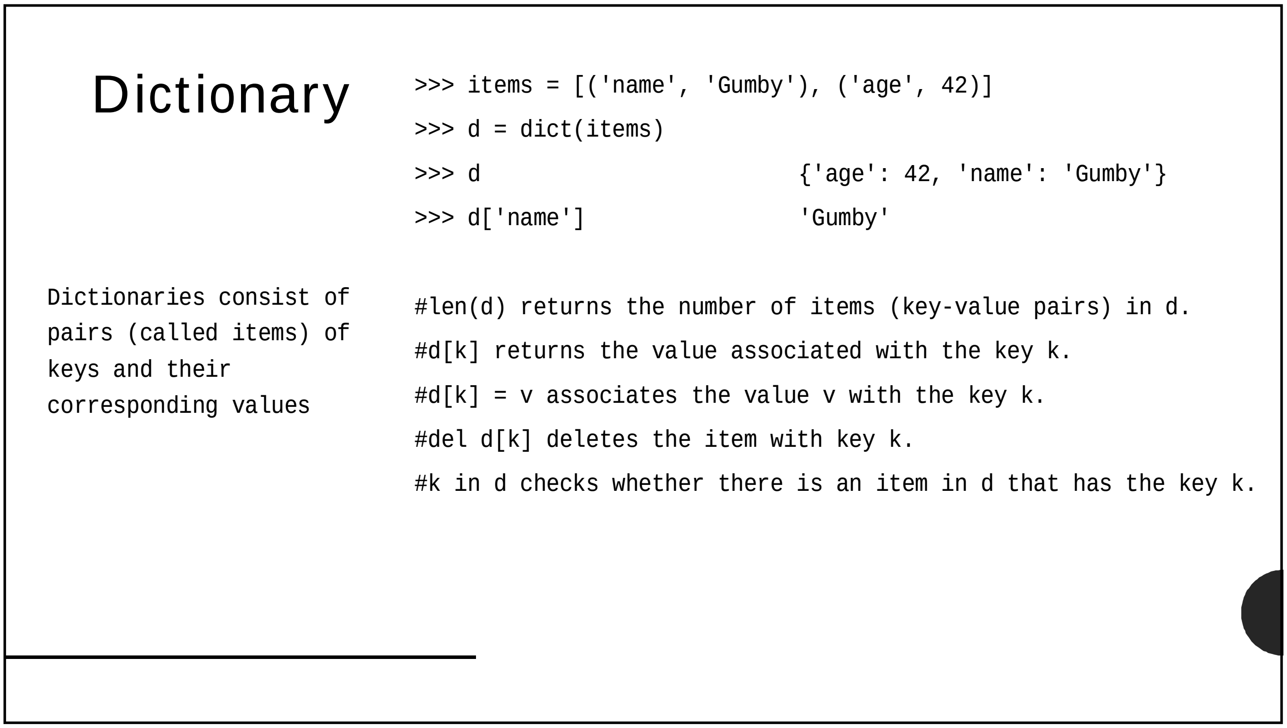
PYTHON

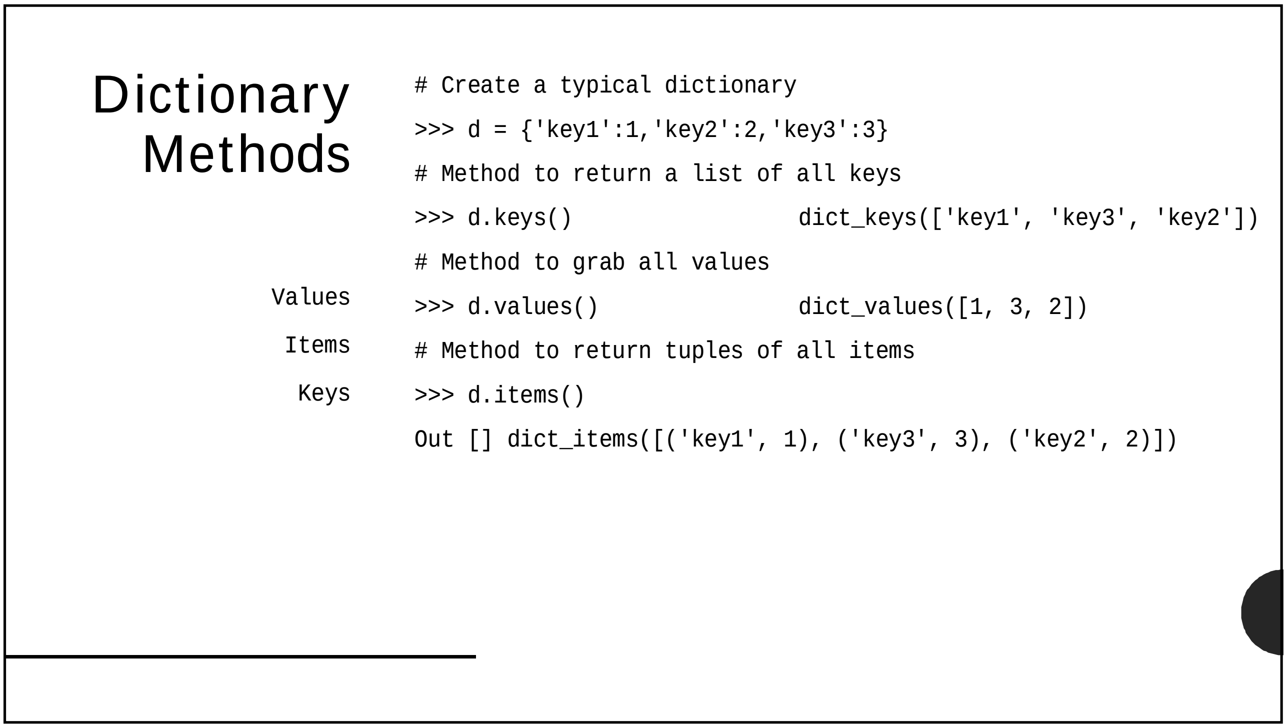
Mohan M J



Dictionary

Dictionaries consist of pairs (called items) of keys and their corresponding values

```
>>> items = [('name', 'Gumby'), ('age', 42)]
>>> d = dict(items)
>>> d                            {'age': 42, 'name': 'Gumby'}
>>> d['name']                    'Gumby'

#len(d) returns the number of items (key-value pairs) in d.
#d[k] returns the value associated with the key k.
#d[k] = v associates the value v with the key k.
#del d[k] deletes the item with key k.
#k in d checks whether there is an item in d that has the key k.
```



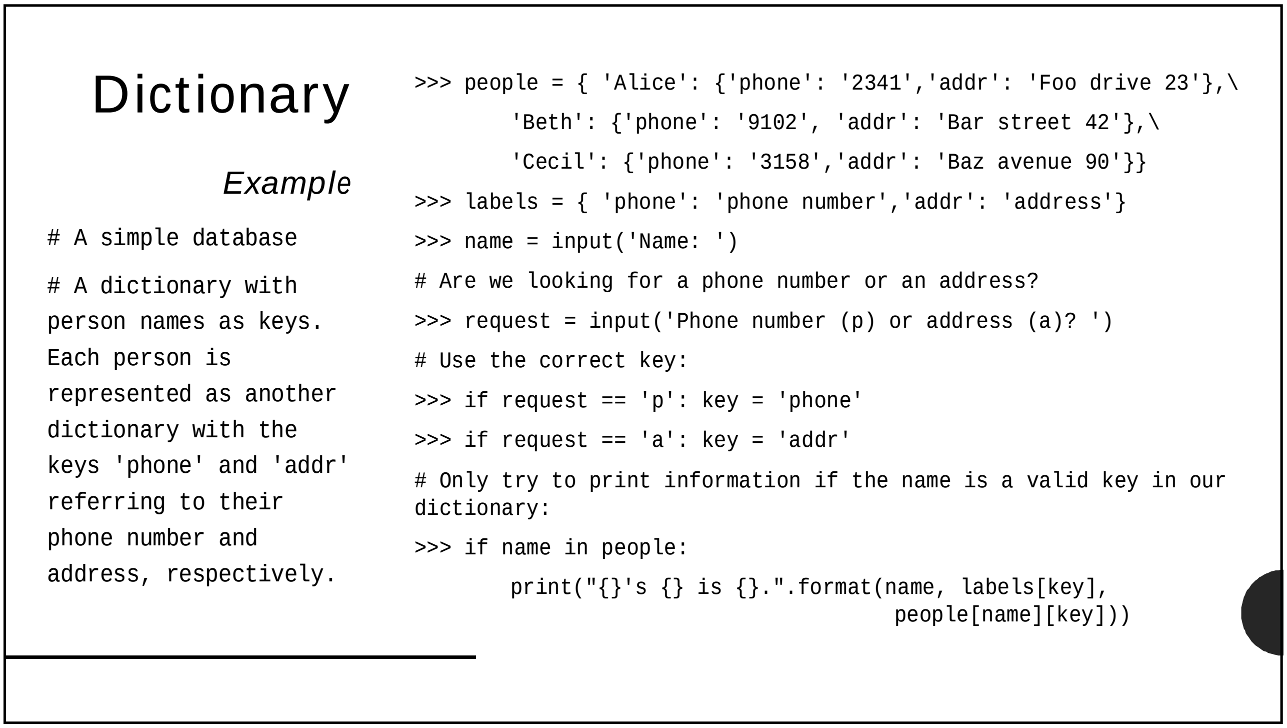
Dictionary Methods

Values

Items

Keys

```
# Create a typical dictionary
>>> d = {'key1':1,'key2':2,'key3':3}
# Method to return a list of all keys
>>> d.keys()                 dict_keys(['key1', 'key3', 'key2'])
# Method to grab all values
>>> d.values()               dict_values([1, 3, 2])
# Method to return tuples of all items
>>> d.items()
Out [] dict_items([('key1', 1), ('key3', 3), ('key2', 2)])
```



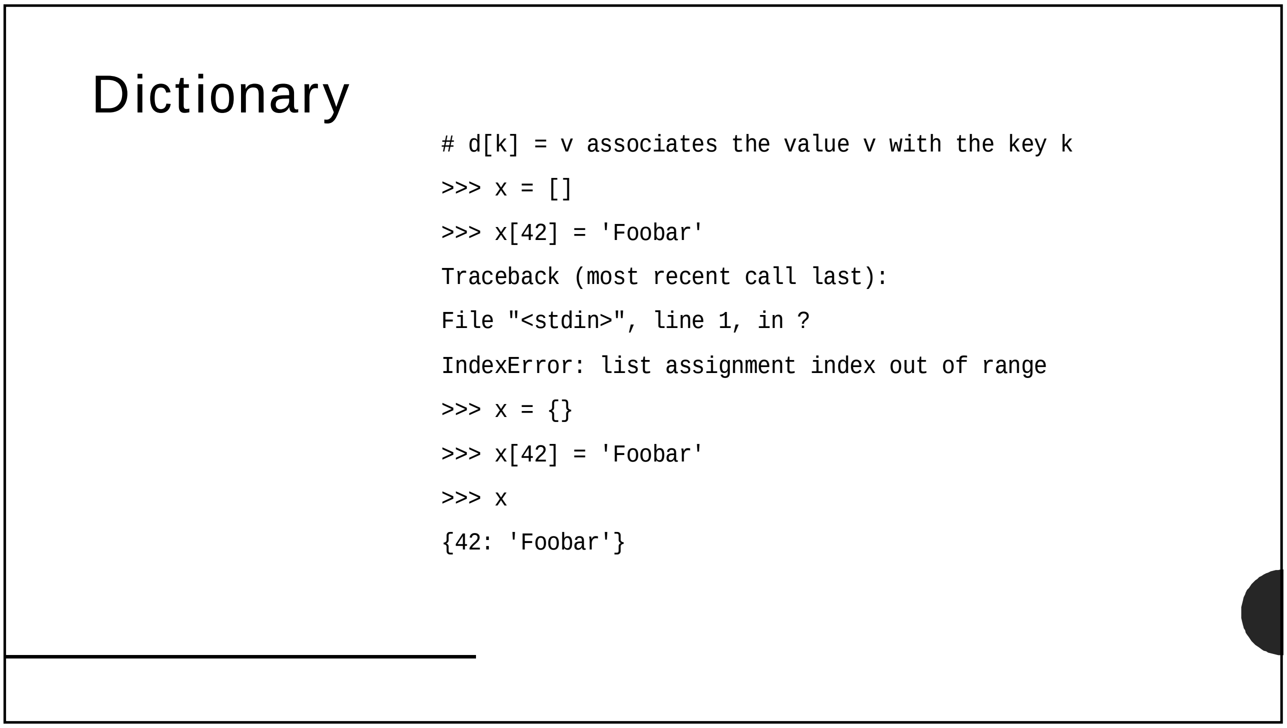
Dictionary

Example

# A simple database

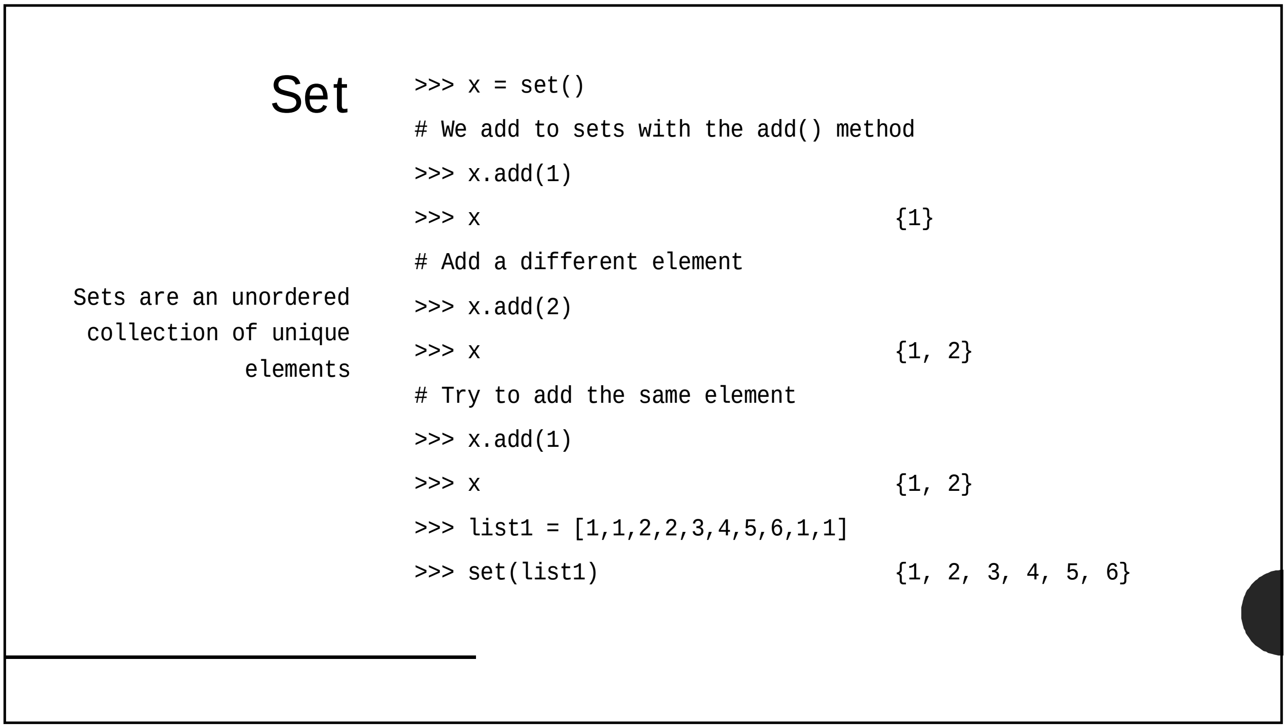
# A dictionary with person names as keys. Each person is represented as another dictionary with the keys 'phone' and 'addr' referring to their phone number and address, respectively.

```
>>> people = { 'Alice': {'phone': '2341','addr': 'Foo drive 23'},\
       'Beth': {'phone': '9102', 'addr': 'Bar street 42'},\
       'Cecil': {'phone': '3158','addr': 'Baz avenue 90'}}
>>> labels = { 'phone': 'phone number','addr': 'address'}
>>> name = input('Name: ')
# Are we looking for a phone number or an address?
>>> request = input('Phone number (p) or address (a)? ')
# Use the correct key:
>>> if request == 'p': key = 'phone'
>>> if request == 'a': key = 'addr'
# Only try to print information if the name is a valid key in our
dictionary:
>>> if name in people:
       print("{}'s {} is {}.".format(name, labels[key],
                                     people[name][key]))
```



Dictionary

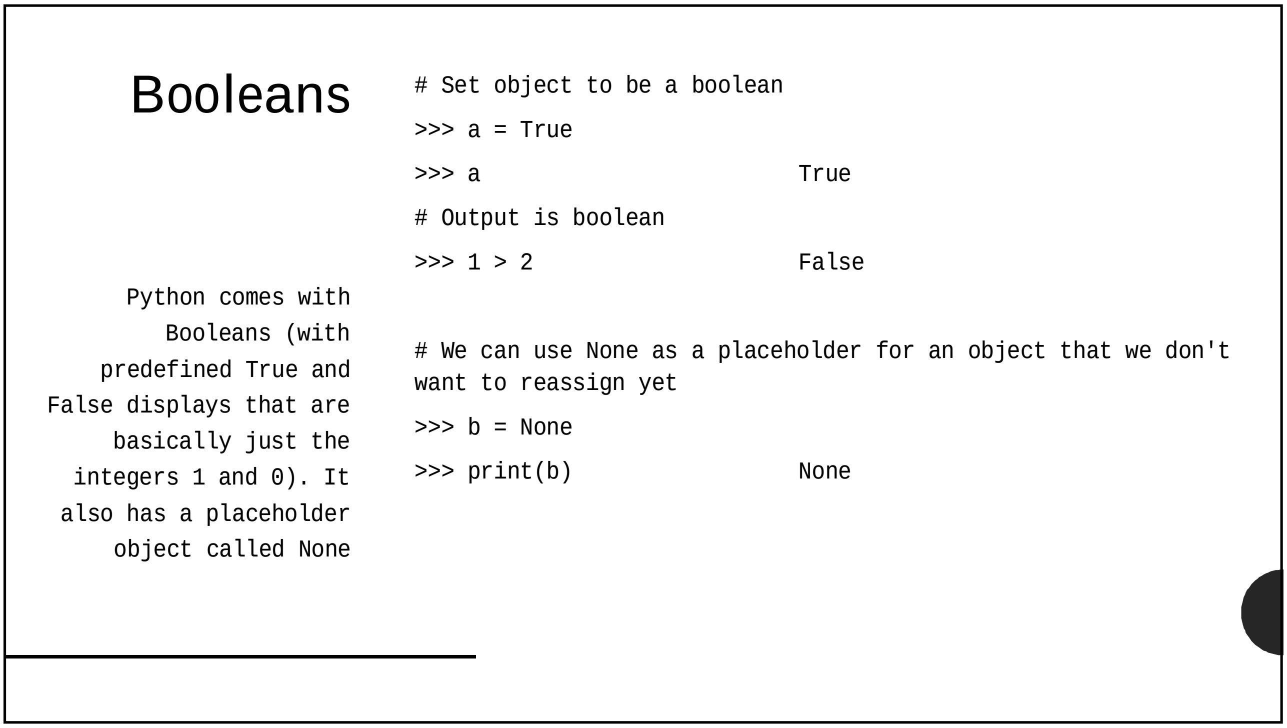
```
# d[k] = v associates the value v with the key k
>>> x = []
>>> x[42] = 'Foobar'
Traceback (most recent call last):
File "<stdin>", line 1, in ?
IndexError: list assignment index out of range
>>> x = {}
>>> x[42] = 'Foobar'
>>> x
{42: 'Foobar'}
```



Set

Sets are an unordered collection of unique elements

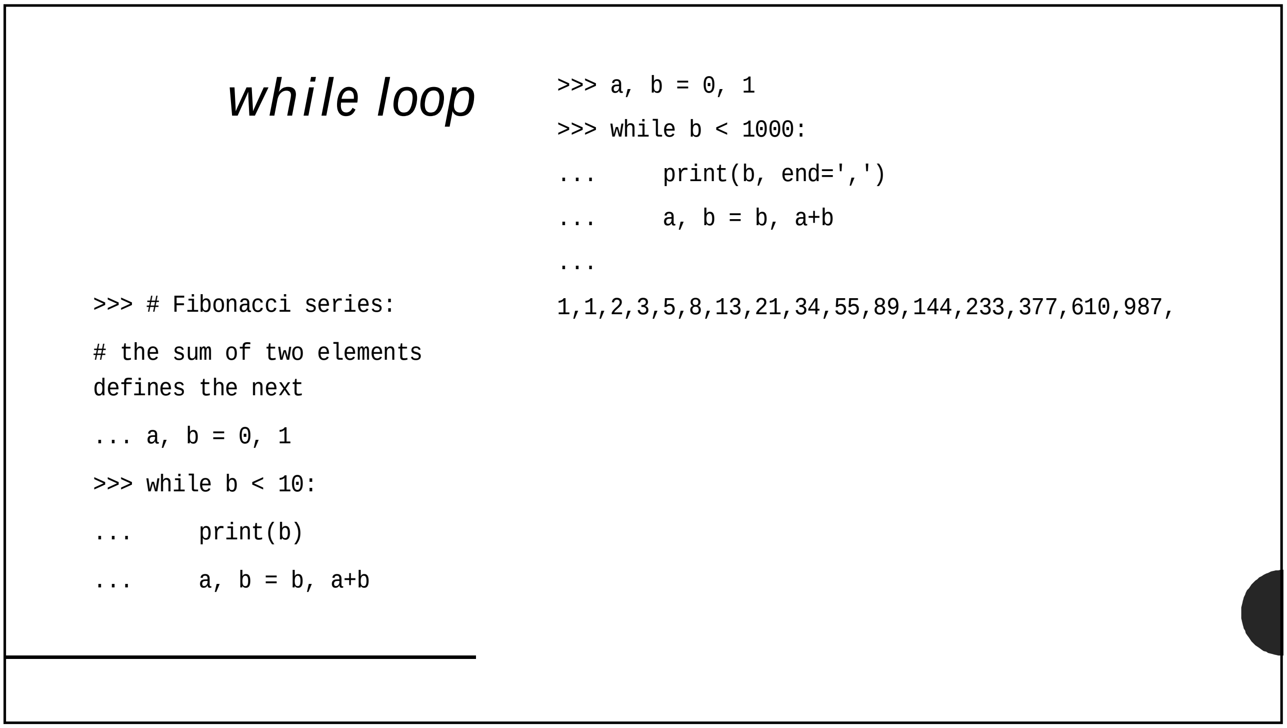
```
>>> x = set()
# We add to sets with the add() method
>>> x.add(1)
>>> x                                {1}
# Add a different element
>>> x.add(2)
>>> x                                {1, 2}
# Try to add the same element
>>> x.add(1)
>>> x                                {1, 2}
>>> list1 = [1,1,2,2,3,4,5,6,1,1]
>>> set(list1)                       {1, 2, 3, 4, 5, 6}
```



Booleans

Python comes with Booleans (with predefined True and False displays that are basically just the integers 1 and 0). It also has a placeholder object called None

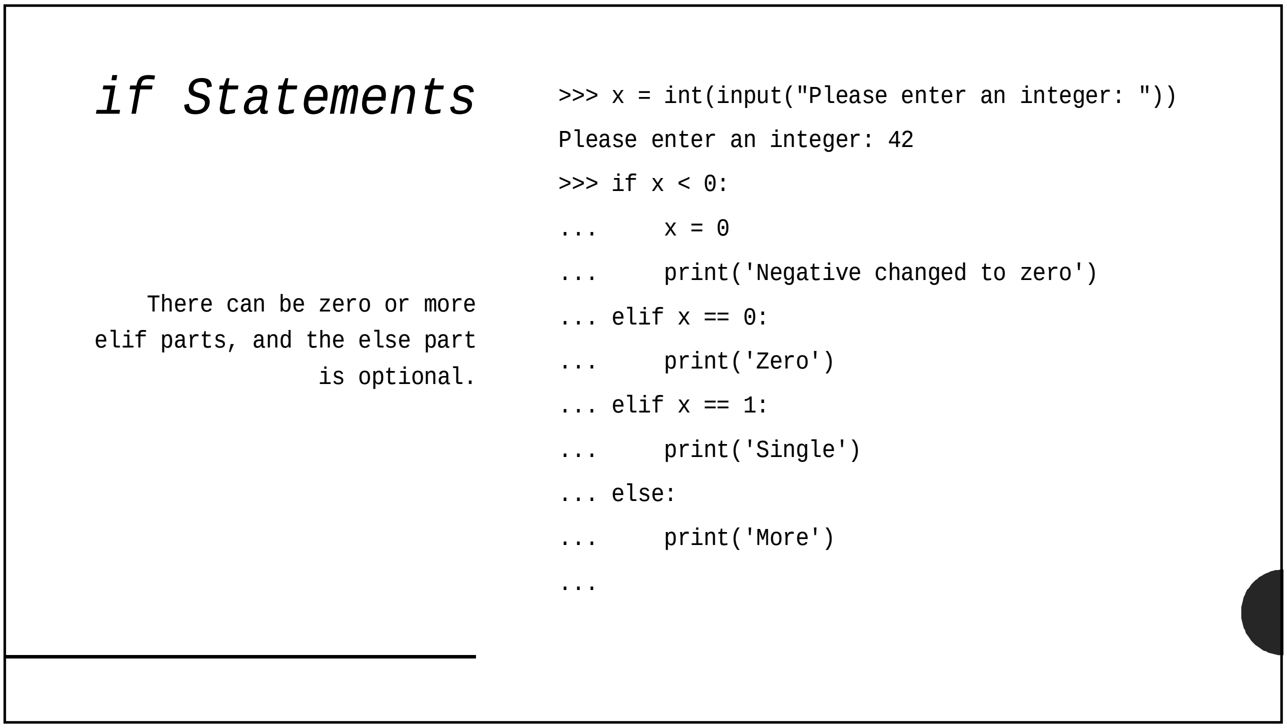
```
# Set object to be a boolean
>>> a = True
>>> a                          True
# Output is boolean
>>> 1 > 2                      False

# We can use None as a placeholder for an object that we don't
want to reassign yet
>>> b = None
>>> print(b)                   None
```



while loop

```
>>> # Fibonacci series:
# the sum of two elements
defines the next
... a, b = 0, 1
>>> while b < 10:
...     print(b)
...     a, b = b, a+b
```

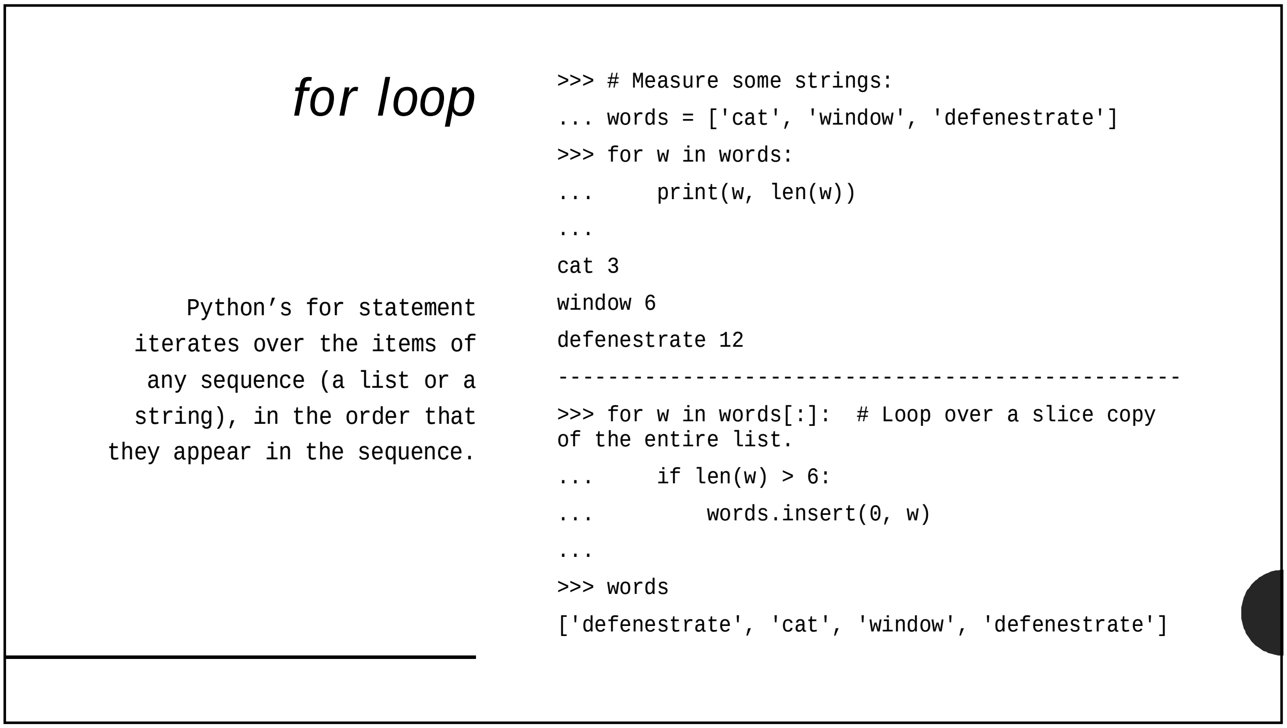
```
>>> a, b = 0, 1
>>> while b < 1000:
...     print(b, end=',')
...     a, b = b, a+b
...
1,1,2,3,5,8,13,21,34,55,89,144,233,377,610,987,
```



if Statements

There can be zero or more elif parts, and the else part is optional.

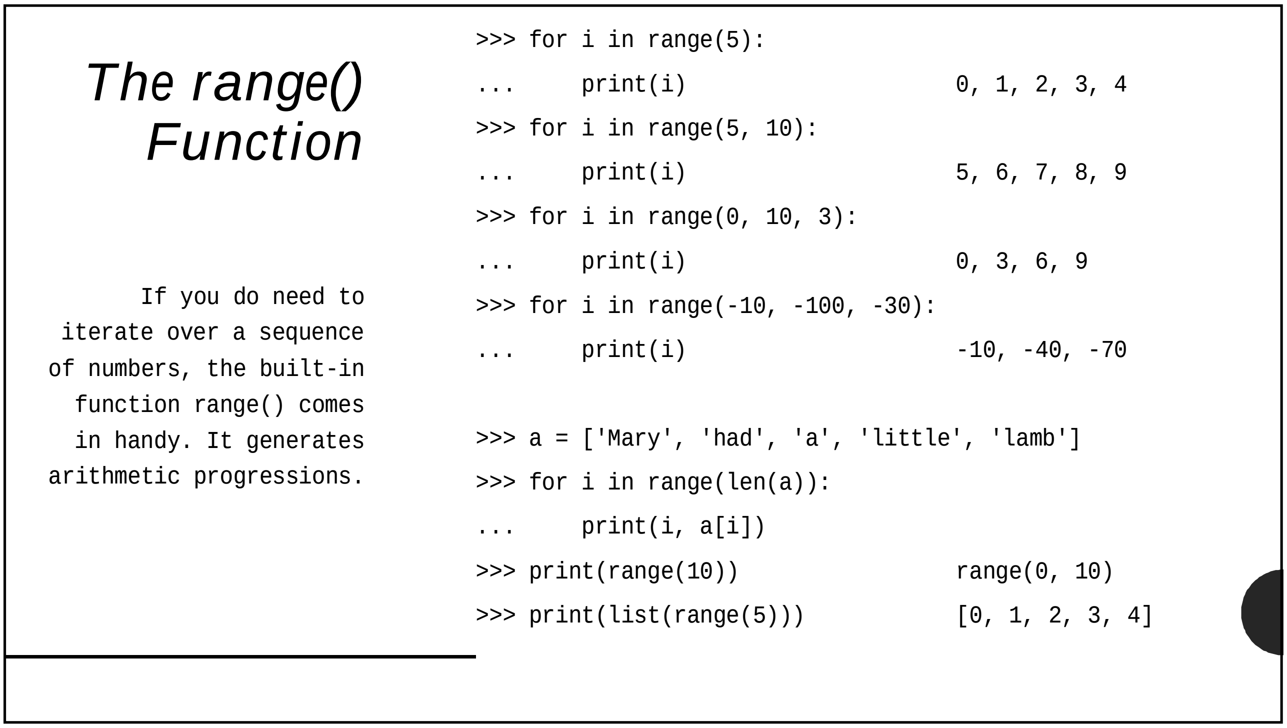
```
>>> x = int(input("Please enter an integer: "))
Please enter an integer: 42
>>> if x < 0:
...     x = 0
...     print('Negative changed to zero')
... elif x == 0:
...     print('Zero')
... elif x == 1:
...     print('Single')
... else:
...     print('More')
...
```



for loop

Python’s for statement iterates over the items of any sequence (a list or a string), in the order that they appear in the sequence.

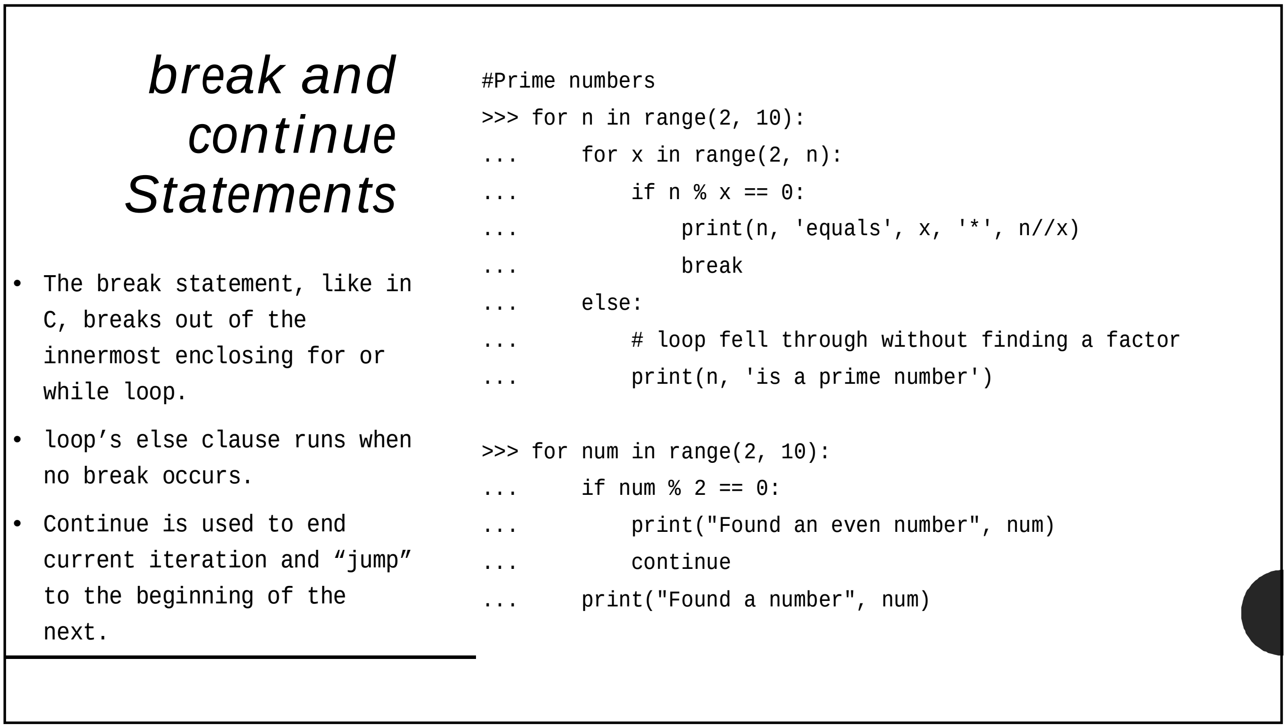
```
>>> # Measure some strings:
... words = ['cat', 'window', 'defenestrate']
>>> for w in words:
...     print(w, len(w))
...
cat 3
window 6
defenestrate 12
--------------------------------------------------
>>> for w in words[:]:  # Loop over a slice copy
of the entire list.
...     if len(w) > 6:
...         words.insert(0, w)
...
>>> words
['defenestrate', 'cat', 'window', 'defenestrate']
```



The range() Function

If you do need to iterate over a sequence of numbers, the built-in function range() comes in handy. It generates arithmetic progressions.

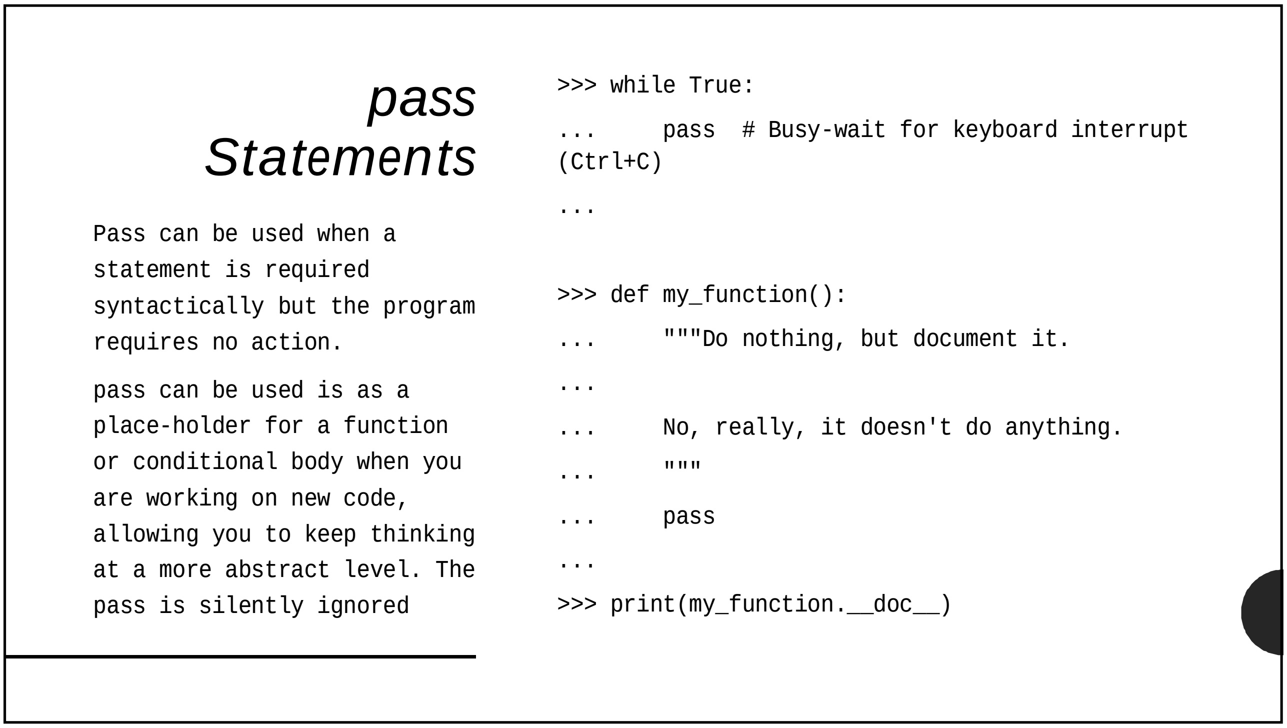
```
>>> for i in range(5):
...     print(i)                     0, 1, 2, 3, 4
>>> for i in range(5, 10):
...     print(i)                     5, 6, 7, 8, 9
>>> for i in range(0, 10, 3):
...     print(i)                     0, 3, 6, 9
>>> for i in range(-10, -100, -30):
...     print(i)                     -10, -40, -70

>>> a = ['Mary', 'had', 'a', 'little', 'lamb']
>>> for i in range(len(a)):
...     print(i, a[i])
>>> print(range(10))                 range(0, 10)
>>> print(list(range(5)))            [0, 1, 2, 3, 4]
```



break and continue Statements

- The break statement, like in C, breaks out of the innermost enclosing for or while loop.
- loop’s else clause runs when no break occurs.
- Continue is used to end current iteration and “jump” to the beginning of the next.

```
#Prime numbers
>>> for n in range(2, 10):
...     for x in range(2, n):
...         if n % x == 0:
...             print(n, 'equals', x, '*', n//x)
...             break
...     else:
...         # loop fell through without finding a factor
...         print(n, 'is a prime number')

>>> for num in range(2, 10):
...     if num % 2 == 0:
...         print("Found an even number", num)
...         continue
...     print("Found a number", num)
```

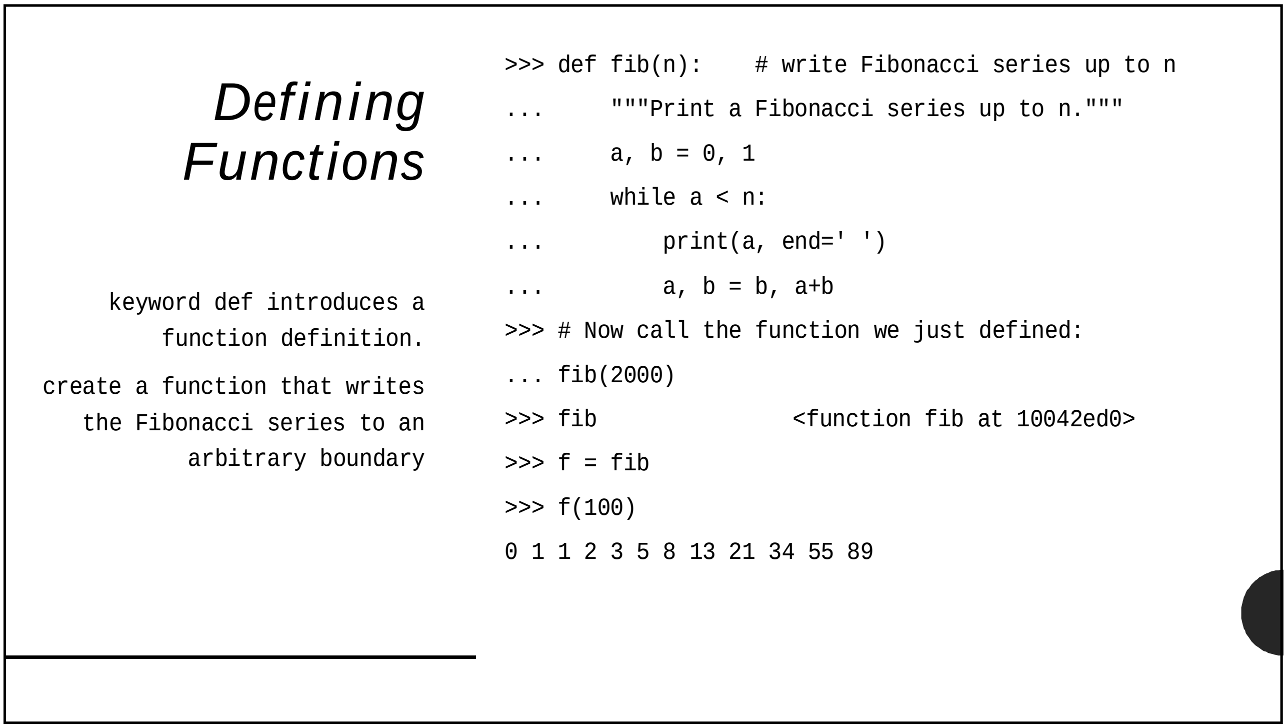


pass Statements

Pass can be used when a statement is required syntactically but the program requires no action.

pass can be used is as a place-holder for a function or conditional body when you are working on new code, allowing you to keep thinking at a more abstract level. The pass is silently ignored

```
>>> while True:
...     pass  # Busy-wait for keyboard interrupt
(Ctrl+C)
...

>>> def my_function():
...     """Do nothing, but document it.
...
...     No, really, it doesn't do anything.
...     """
...     pass
...
>>> print(my_function.__doc__)
```

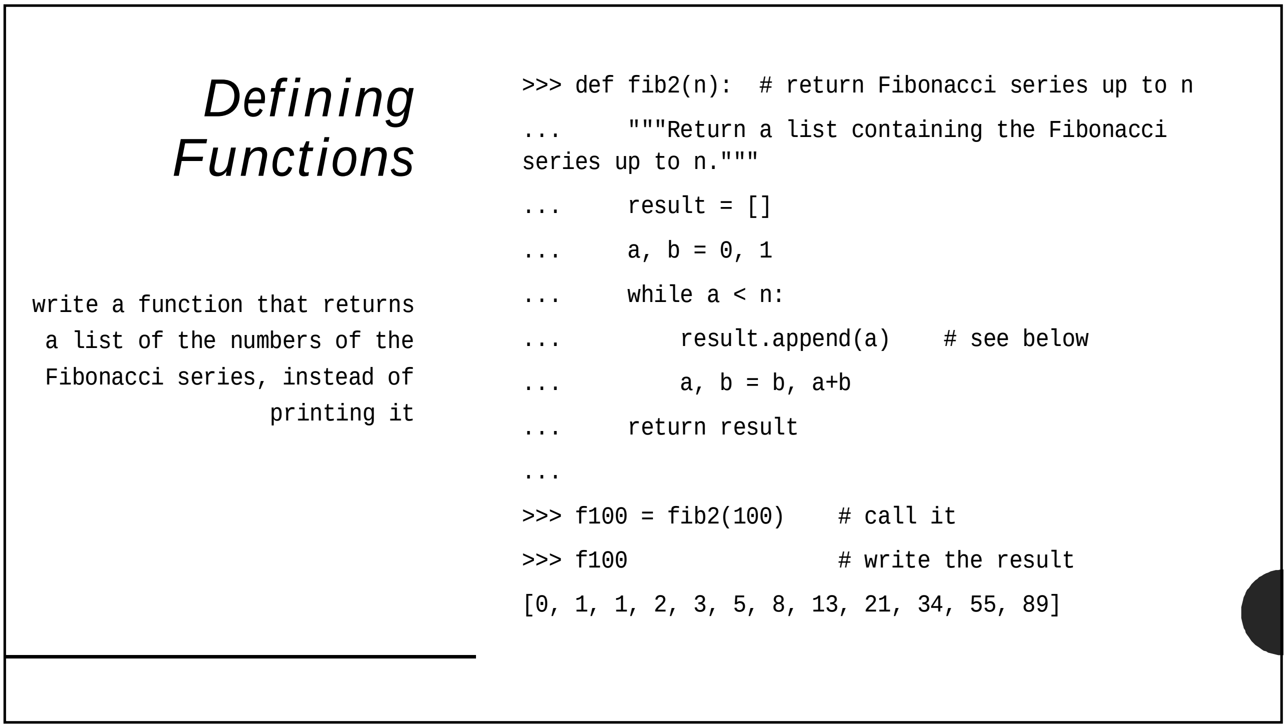


Defining Functions

keyword def introduces a function definition.

create a function that writes the Fibonacci series to an arbitrary boundary

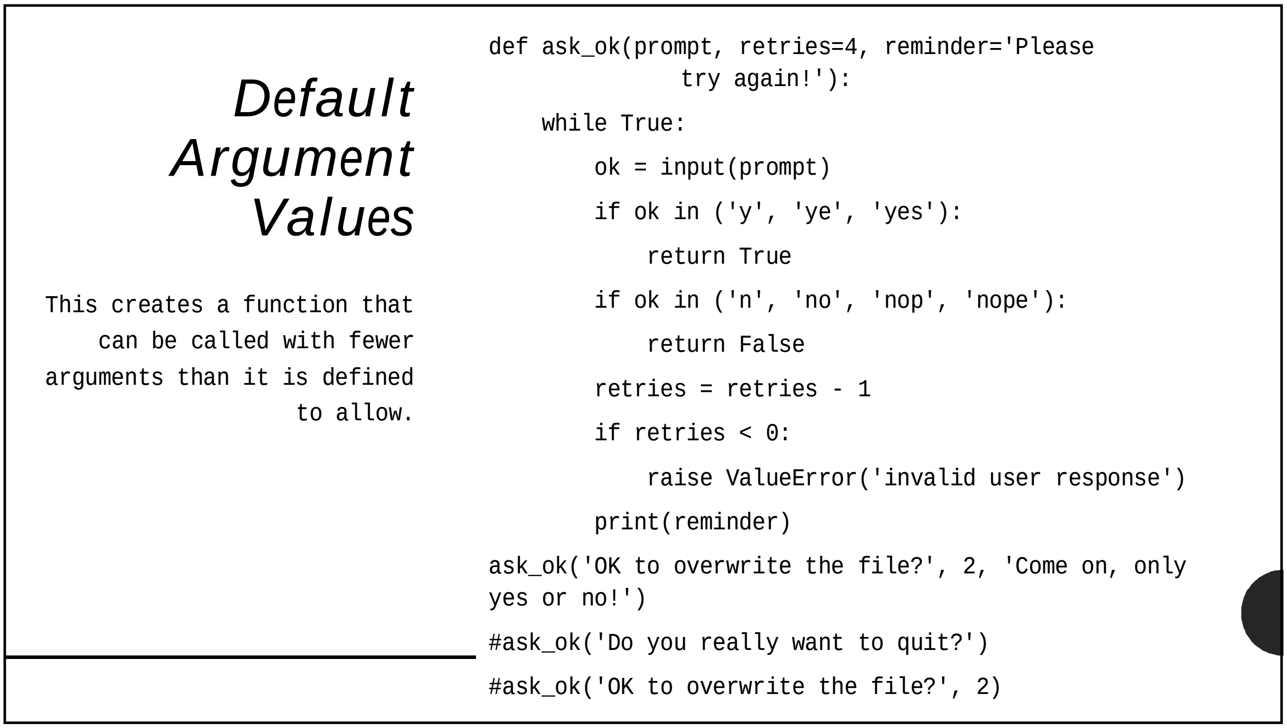
```
>>> def fib(n):    # write Fibonacci series up to n
...     """Print a Fibonacci series up to n."""
...     a, b = 0, 1
...     while a < n:
...         print(a, end=' ')
...         a, b = b, a+b
>>> # Now call the function we just defined:
... fib(2000)
>>> fib              <function fib at 10042ed0>
>>> f = fib
>>> f(100)
0 1 1 2 3 5 8 13 21 34 55 89
```



Defining Functions

write a function that returns a list of the numbers of the Fibonacci series, instead of printing it

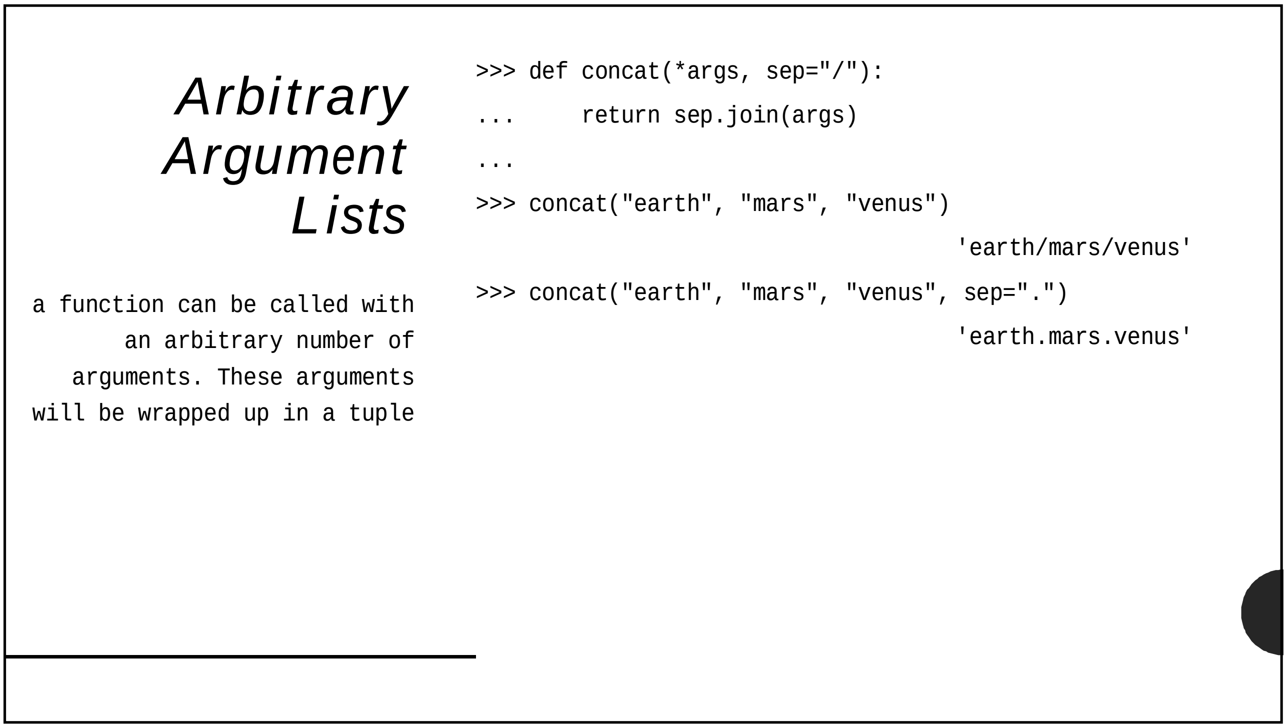
```
>>> def fib2(n):  # return Fibonacci series up to n
...     """Return a list containing the Fibonacci
series up to n."""
...     result = []
...     a, b = 0, 1
...     while a < n:
...         result.append(a)    # see below
...         a, b = b, a+b
...     return result
...
>>> f100 = fib2(100)    # call it
>>> f100                # write the result
[0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89]
```



Default Argument Values

This creates a function that can be called with fewer arguments than it is defined to allow.

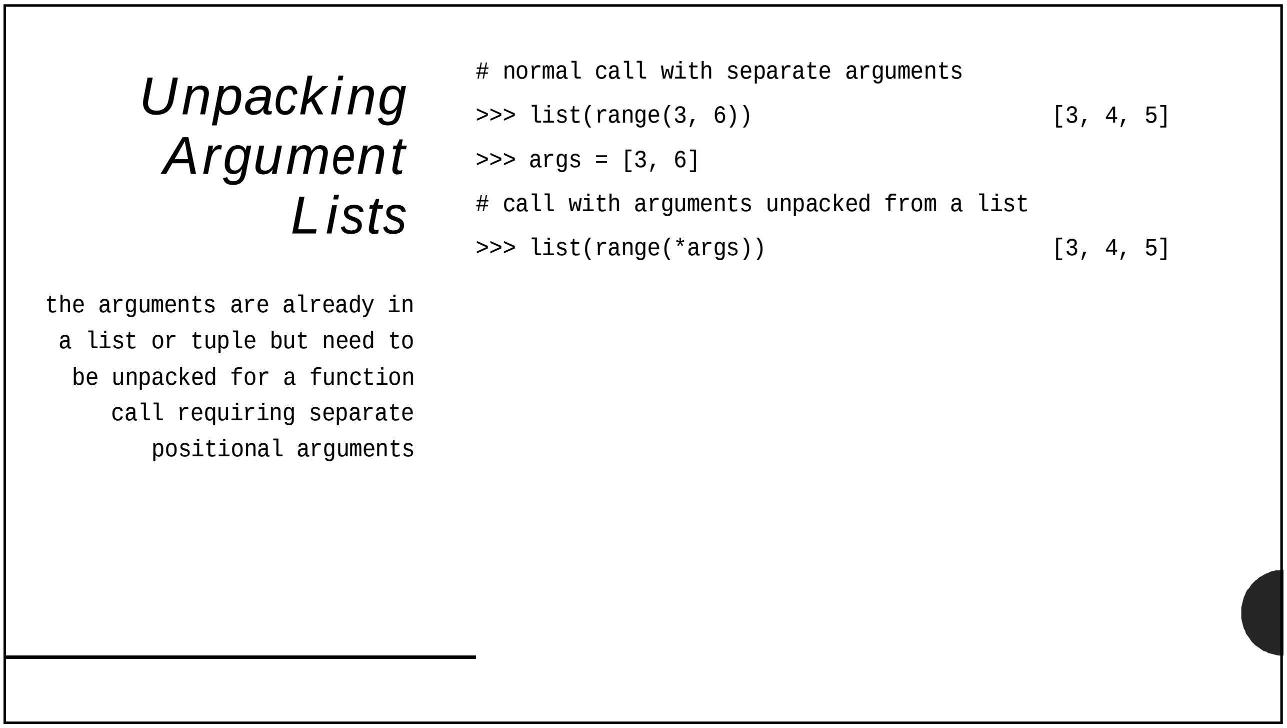
```
def ask_ok(prompt, retries=4, reminder='Please
              try again!'):
    while True:
        ok = input(prompt)
        if ok in ('y', 'ye', 'yes'):
            return True
        if ok in ('n', 'no', 'nop', 'nope'):
            return False
        retries = retries - 1
        if retries < 0:
            raise ValueError('invalid user response')
        print(reminder)
ask_ok('OK to overwrite the file?', 2, 'Come on, only
yes or no!')
#ask_ok('Do you really want to quit?')
#ask_ok('OK to overwrite the file?', 2)
```



Arbitrary Argument Lists

a function can be called with an arbitrary number of arguments. These arguments will be wrapped up in a tuple

```
>>> def concat(*args, sep="/"):
...     return sep.join(args)
...
>>> concat("earth", "mars", "venus")
                                    'earth/mars/venus'
>>> concat("earth", "mars", "venus", sep=".")
                                    'earth.mars.venus'
```



Unpacking Argument Lists

the arguments are already in a list or tuple but need to be unpacked for a function call requiring separate positional arguments

```
# normal call with separate arguments
>>> list(range(3, 6))                      [3, 4, 5]
>>> args = [3, 6]
# call with arguments unpacked from a list
>>> list(range(*args))                     [3, 4, 5]
```

Lambda Expressions, map and filter

Small anonymous functions can be created with the lambda keyword. Lambda functions can be used wherever function objects are required. They are syntactically restricted to a single expression.

```
>>> def make_incrementor(n):
...     return lambda x: x + n
>>> f = make_incrementor(42)
>>> f(0)                           42
>>> f(1)                           43
>>> def times2(var):
       return var*2
>>> times2(2)                      4
>>> seq = [1,2,3,4,5]
>>> map(times2,seq)
>>> list(map(times2,seq))
>>> list(map(lambda var: var*2,seq))
>>> filter(lambda item: item%2 == 0,seq)
>>> list(filter(lambda item: item%2 == 0,seq))
```

THANK YOU