Programming Principles in Python (CSCI 503/490)

Functions & Dictionaries

Dr. David Koop

(some slides adapted from Dr. Reva Freedman)

Sequences and Mutability

- Lists are **mutable**: you can modify them after creating them
  - ```
    my_list = [1,2,3]
    my_list[2] = 200
    ```
- Tuples and strings are immutable
  - ```
    my_tuple = (1,2,3)
    my_tuple[2] = 200 # Error
    ```

List methods

Method	Meaning
<list>.append(d)	Add element d to end of list.
<list>.extend(s)	Add **all** elements in s to end of list.
<list>.insert(i, d)	Insert d into list at index i.
<list>.pop(i)	Deletes ith element of the list and returns its value.
<list>.sort()	Sort the list.
<list>.reverse()	Reverse the list.
<list>.remove(d)	Deletes first occurrence of d in list.
<list>.index(d)	Returns index of first occurrence of d.
<list>.count(d)	Returns the number of occurrences of d in list.

List methods

Mutate

Method	Meaning
<list>.append(d)	Add element d to end of list.
<list>.extend(s)	Add **all** elements in s to end of list.
<list>.insert(i, d)	Insert d into list at index i.
<list>.pop(i)	Deletes ith element of the list and returns its value.
<list>.sort()	Sort the list.
<list>.reverse()	Reverse the list.
<list>.remove(d)	Deletes first occurrence of d in list.
<list>.index(d)	Returns index of first occurrence of d.
<list>.count(d)	Returns the number of occurrences of d in list.

Updating collections

- There are three ways to deal with operations that update collections:
  - Returns an **updated copy** of the collection
  - Updates the collection **in place**
  - Updates the collection in place **and returns it**
- `list.sort` and `list.reverse` work **in place** and **don't return** it
- `sorted` returns an **updated copy,** `reversed` returns an iterator
  - `reversed` actually returns an iterator
  - these also work for immutable sequences like strings and tuples

enumerate

- Often you **do not** need the index when iterating through a sequence
- If you need an index while looping through a sequence, use `enumerate`
- ```
  for i, d in enumerate(my_list):
      print("index:", i, "element:", d)
  ```
- Each time through the loop, it yields **two** items, the **index** `i` & the **element** `d`
- `i, d` is actually a **tuple**
- Automatically **unpacked** above, can manually do this, but don't!
- ```
  for t in enumerate(my_list):
      i = t[0]
      d = t[1]
      print("index:", i, "element:", d)
  ```

enumerate

- Often you **do not** need the index when iterating through a sequence
- If you need an index while looping through a sequence, use `enumerate`
- ```
  for i, d in enumerate(my_list):
      print("index:", i, "element:", d)
  ```
- Each time through the loop, it yields **two** items, the **index** `i` & the **element** `d`
- `i, d` is actually a **tuple**
- Automatically **unpacked** above, can manually do this, but don't!
- ```
  for t in enumerate(my_list):
      i = t[0]
      d = t[1]
      print("index:", i, "element:", d)
  ```

Tuple Packing and Unpacking

- ```
  def f(a, b):
      if a > 3:
          return a, b-a # tuple packing
      return a+b, b # tuple packing
  ```
- `c, d = f(4, 3) # tuple unpacking`

- Make sure to unpack the correct number of variables!
- `c, d = a+b, a-b, 2*a # ValueError: too many values to unpack`
- Sometimes, check return value before unpacking:
  - ```
    retval = f(42)
    if retval is not None:
        c, d = retval
    ```

Tuple Packing and Unpacking

- ```
  def f(a, b):
      if a > 3:
          return a, b-a # tuple packing
      return a+b, b # tuple packing
  ```

```
t = (a, b-a)
return t
```

- `c, d = f(4, 3) # tuple unpacking`

- Make sure to unpack the correct number of variables!
- `c, d = a+b, a-b, 2*a # ValueError: too many values to unpack`
- Sometimes, check return value before unpacking:
  - ```
    retval = f(42)
    if retval is not None:
        c, d = retval
    ```

Tuple Packing and Unpacking

- ```
  def f(a, b):
      if a > 3:
          return a, b-a # tuple packing
      return a+b, b # tuple packing
  ```
- `c, d = f(4, 3) # tuple unpacking`

```
t = (a, b-a)
return t
```

```
t = f(4, 3)
(c, d) = t
```

- Make sure to unpack the correct number of variables!
- `c, d = a+b, a-b, 2*a # ValueError: too many values to unpack`
- Sometimes, check return value before unpacking:
  - ```
    retval = f(42)
    if retval is not None:
        c, d = retval
    ```

Scope

- The **scope** of a variable refers to where in a program it can be referenced
- Python has three scopes:
  - **global**: defined outside a function
  - **local**: in a function, only valid in the function
  - **nonlocal**: can be used with nested functions
- Python allows variables in different scopes to have the **same name**

Scope

- Global Read
  - ```
    def f():
        print(x) # x is global
    x = 1
    f()
    print(x) # x is 1
    ```
- Local Read/Write
  - ```
    def f():
        x = 2
        print(x) # x is local
    x = 1
    f()
    print(x) # x is 1
    ```
- Global Read/Write
  - ```
    def f():
        global x
        x = 2
        print(x) # x is global
    x = 1
    f()
    print(x) # x is 2
    ```
    ```

Assignment 3

- Out Soon

What is the scope of a parameter of a function?

Depends on whether Python is
pass-by-value or pass-by-reference

Pass by value

- Detour to C++ land:
  - ```
    void f(int x) {
        x = 2;
        cout << "Value of x in f: " << x << endl;
    }

    main() {
        int x = 1;
        f(x);
        cout << "Value of x in main: " << x;
    }
    ```
    ```

Pass by value

- Detour to C++ land:
  - ```
    void f(int x) {
        x = 2;
        cout << "Value of x in f: " << x << endl;
    }

    main() {
        int x = 1;
        f(x);
        cout << "Value of x in main: " << x;
    }
    ```

Output:

```
Value of x in f: 2
Value of x in main: 1
```
```

Pass by reference

- Detour to C++ land:
  - ```
    void f(int & x) {
        x = 2;
        cout << "Value of x in f: " << x << endl;
    }

    main() {
        int x = 1;
        f(x);
        cout << "Value of x in main: " << x;
    }
    ```
    ```

Pass by reference

- Detour to C++ land:
  - ```
    void f(int & x) {
        x = 2;
        cout << "Value of x in f: " << x << endl;
    }

    main() {
        int x = 1;
        f(x);
        cout << "Value of x in main: " << x;
    }
    ```
    ```

Pass by reference

- Detour to C++ land:
  - ```
    void f(int & x) {
        x = 2;
        cout << "Value of x in f: " << x << endl;
    }

    main() {
        int x = 1;
        f(x);
        cout << "Value of x in main: " << x;
    }
    ```

Output:

```
Value of x in f: 2
Value of x in main: 2
```
```

Pass by reference

- Detour to C++ land:

```
- void f(int & x) {
      x = 2;
      cout << "Value of x in f: " << x << endl;
  }

  main() {
      int x = 1;
      f(x);
      cout << "Value of x in main: " << x;
  }
```

Output:

```
Value of x in f: 2
Value of x in main: 2
```

Is Python pass-by-value or pass-by-reference?

Example 1

- ```
  def change_list(inner_list):
      inner_list = [10,9,8,7,6]

  outer_list = [0,1,2,3,4]
  change_list(outer_list)
  outer_list # [0,1,2,3,4]
  ```

- Looks like pass by value!
  ```

Example 2

- ```
  def change_list(inner_list):
      inner_list.append(5)

  outer_list = [0,1,2,3,4]
  change_list(outer_list)
  outer_list # [0,1,2,3,4,5]
  ```

- Looks like pass by reference!
  ```

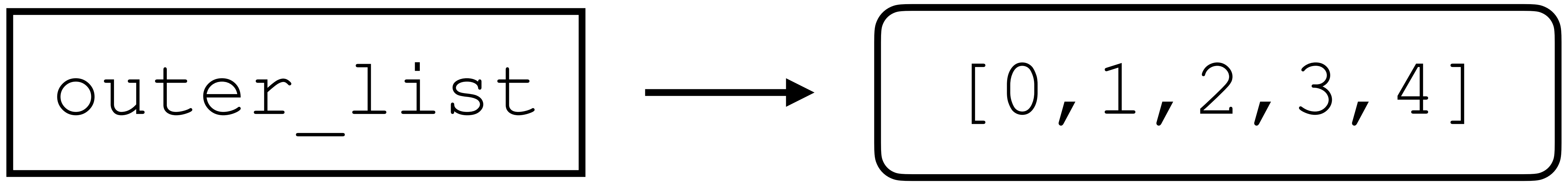
What's going on?

Think about how assignment works in Python
Different than C++

Example 1

- ```
  def change_list(inner_list):
      inner_list = [10,9,8,7,6]
  ```

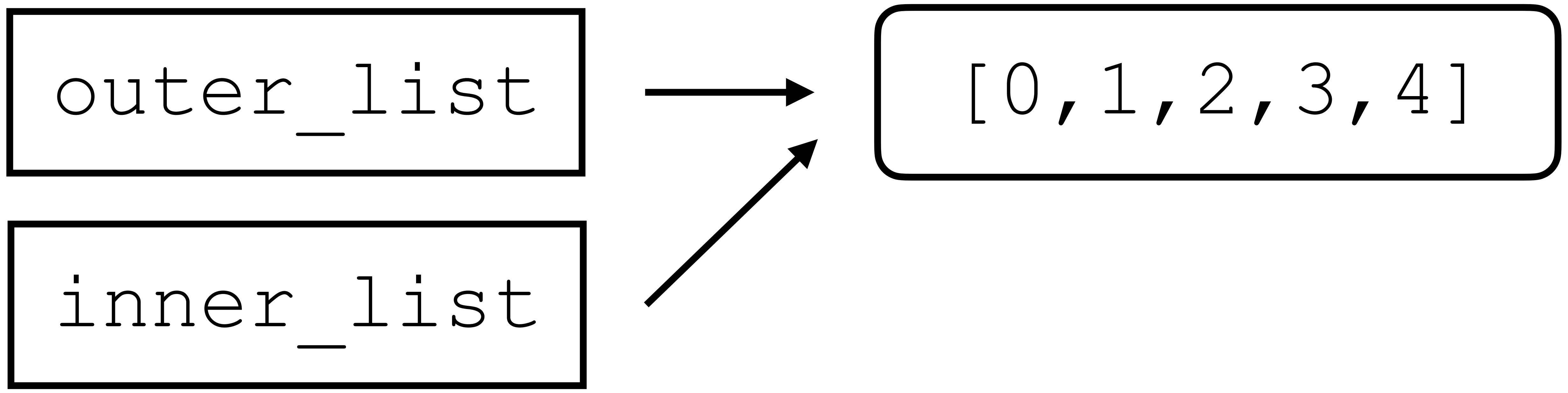
  ```
  outer_list = [0,1,2,3,4]
  change_list(outer_list)
  outer_list # [0,1,2,3,4]
  ```


  ```

Example 1

- ```
  def change_list(inner_list):
      inner_list = [10,9,8,7,6]

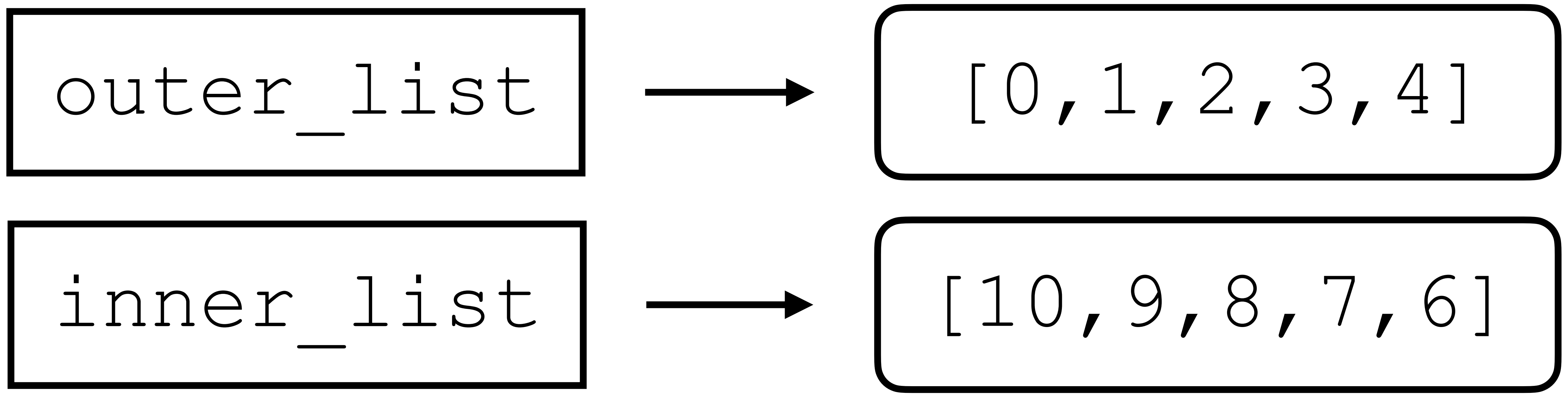
  outer_list = [0,1,2,3,4]
  change_list(outer_list)
  outer_list # [0,1,2,3,4]
  ```


  ```

Example 1

- ```
  def change_list(inner_list):
      inner_list = [10,9,8,7,6]

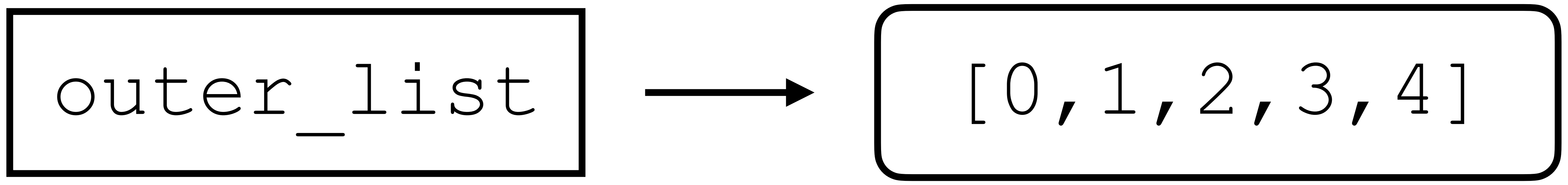
  outer_list = [0,1,2,3,4]
  change_list(outer_list)
  outer_list # [0,1,2,3,4]
  ```


  ```

Example 1

- ```
  def change_list(inner_list):
      inner_list = [10,9,8,7,6]

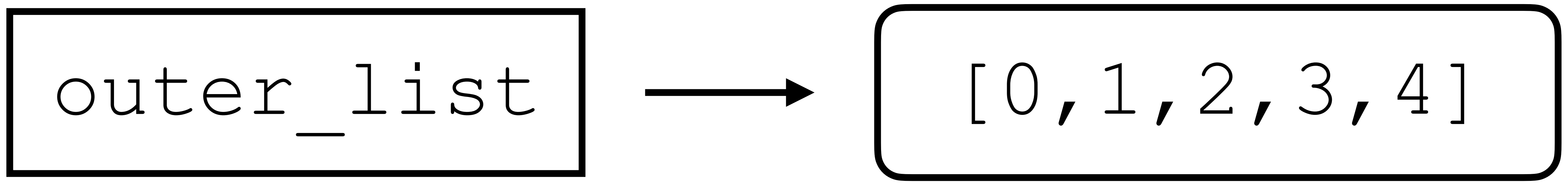
  outer_list = [0,1,2,3,4]
  change_list(outer_list)
  outer_list # [0,1,2,3,4]
  ```


  ```

Example 2

- ```
  def change_list(inner_list):
      inner_list.append(5)

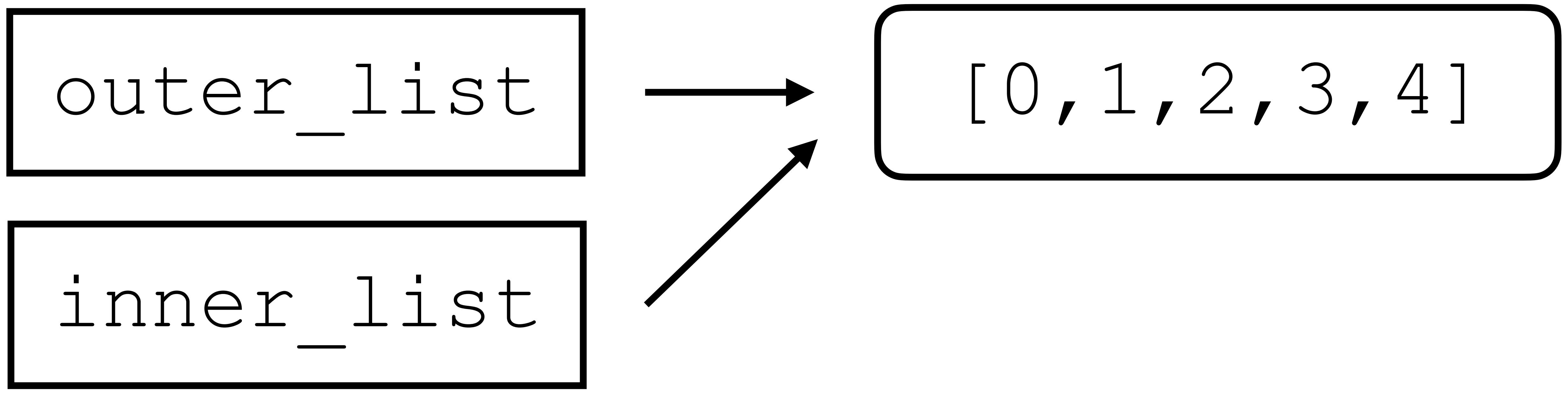
  outer_list = [0,1,2,3,4]
  change_list(outer_list)
  outer_list # [0,1,2,3,4,5]
  ```


  ```

Example 2

- ```
  def change_list(inner_list):
      inner_list.append(5)

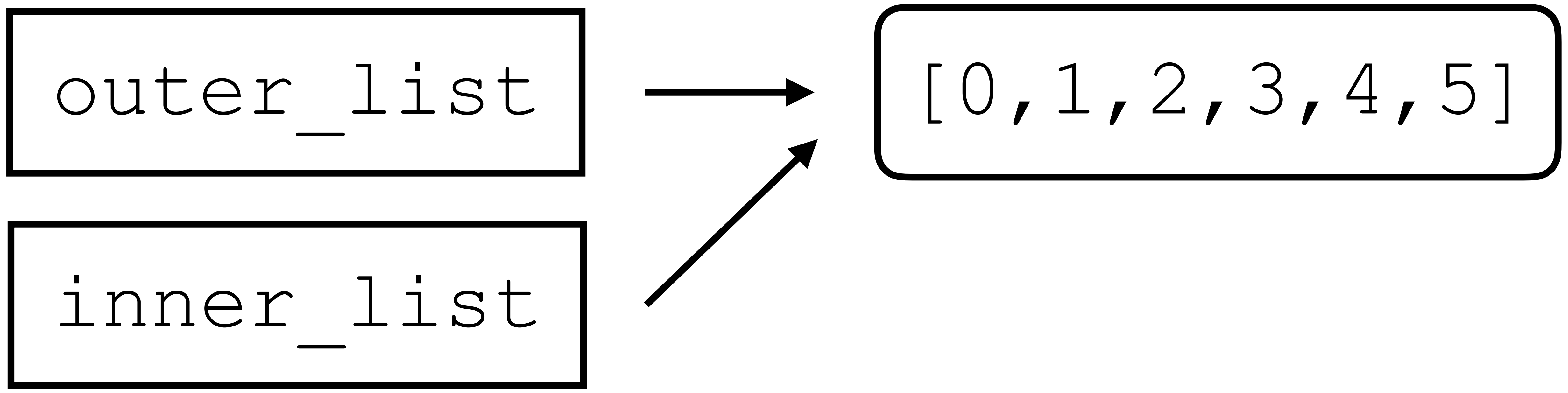
  outer_list = [0,1,2,3,4]
  change_list(outer_list)
  outer_list # [0,1,2,3,4,5]
  ```


  ```

Example 2

- ```
  def change_list(inner_list):
      inner_list.append(5)

  outer_list = [0,1,2,3,4]
  change_list(outer_list)
  outer_list # [0,1,2,3,4,5]
  ```


  ```

Example 2

```
def change_list(inner_list):
    inner_list.append(5)

outer_list = [0,1,2,3,4]
change_list(outer_list)
outer_list # [0,1,2,3,4,5]
```

outer_list → [0,1,2,3,4,5]

Pass by object reference

- AKA passing object references by value
- Python doesn't allocate space for a variable, it just links identifier to a value
- **Mutability** of the object determines whether other references see the change
- Any immutable object will act like pass by value
- Any mutable object acts like pass by reference unless it is reassigned to a new value

Remember: global allows assignment in functions

- ```
  def change_list():
      global a_list
      a_list = [10,9,8,7,6]

  a_list = [0,1,2,3,4]
  change_list()
  a_list # [10,9,8,7,6]
  ```
  ```

Default Parameter Values

- Can add `=<value>` to parameters
-
```
def rectangle_area(width=30, height=20):
    return width * height
```
- All of these work:
  - `rectangle_area() # 600`
  - `rectangle_area(10) # 200`
  - `rectangle_area(10,50) # 500`
- If the user does not pass an argument for that parameter, the parameter is set to the default value
- Cannot add non-default parameters after a defaulted parameter
  - ~~`def rectangle_area(width=30, height)`~~

[Deitel & Deitel]

Don't use mutable values as defaults!

- ```
  def append_to(element, to=[]):
      to.append(element)
      return to
  ```
- ```
  my_list = append_to(12)
  my_list # [12]
  ```
- ```
  my_other_list = append_to(42)
  my_other_list # [12, 42]
  ```

[K. Reitz and T. Schlusser]
  ```

Use None as a default instead

- ```
  def append_to(element, to=None):
      if to is None:
          to = []
      to.append(element)
      return to
  ```
- ```
  my_list = append_to(12)
  my_list # [12]
  ```
- ```
  my_other_list = append_to(42)
  my_other_list # [42]
  ```

- If you're not mutating, this isn't an issue

[K. Reitz and T. Schlusser]
  ```

Keyword Arguments

- Keyword arguments allow someone calling a function to specify exactly which values they wish to specify without specifying all the values
- This helps with long parameter lists where the caller wants to only change a few arguments from the defaults
-
```
def f(alpha=3, beta=4, gamma=1, delta=7, epsilon=8, zeta=2,
      eta=0.3, theta=0.5, iota=0.24, kappa=0.134):
    # …
```
- `f(beta=12, iota=0.7)`

Positional & Keyword Arguments

- Generally, any argument can be passed as a keyword argument
- ```
  def f(alpha, beta, gamma=1, delta=7, epsilon=8, zeta=2,
            eta=0.3, theta=0.5, iota=0.24, kappa=0.134):
      # …
  ```
- `f(5,6)`
- `f(alpha=7, beta=12, iota=0.7)`
  ```

Position-Only Arguments

- PEP 570 introduced position-only arguments
- Sometimes it makes sense that certain arguments must be position-only
- Certain functions (those implemented in C) only allow position-only: `pow`
- Add a slash (`/`) to delineate where keyword arguments start
-
```
def f(alpha, beta, /, gamma=1, delta=7, epsilon=8, zeta=2,
        eta=0.3, theta=0.5, iota=0.24, kappa=0.134):
    # …
```
  - `f(alpha=7, beta=12, iota=0.7) # ERROR`
  - `f(7, 12, iota=0.7) # WORKS`

Arbitrary Argument Containers

- ```
  def f(*args, **kwargs):
      # ...
  ```
- `args`: a tuple of arguments
- `kwargs`: a key-value dictionary of arguments
- Stars in function signature, not in use
- Can have named arguments before these arbitrary containers
- Any values set by position will not be in kwargs:
- ```
  def f(a, *args, **kwargs):
      print(args)
      print(kwargs)
  f(a=3, b=5) # args is empty, kwargs has only b
  ```

Keyword-Only Arguments

- Can also force arguments to be keyword-only
- Uses a `*` (star) without a variable name
- All arguments following the star must be keyword-only
  - `def compare(a, b, *, key=None)`
- Arguments **do not** need to have a default value
  - `def compare(a, b, *, key)`

Programming Principles: Defining Functions

- List arguments in an order that makes sense
  - May be convention => `pow(x,y)` means x^y
  - May be in order of expected frequency used
- Use default parameters when meaningful defaults are known
- Use position-only arguments when there is no meaningful name or the syntax might change in the future

Calling module functions

- Some functions exist in modules (we will discuss these more later)
- Import module
- Call functions by prepending the module name plus a dot
- ```
  import math
  math.log10(100)
  math.sqrt(196)
  ```
  ```

Calling object methods

- Some functions are defined for objects like strings
- These are **instance methods**
- Call these using a similar dot-notation
- Can take arguments
- ```
  s = 'Mary'
  s.upper() # 'MARY'
  ```
- ```
  t = '   extra spaces  '
  t.strip() # 'extra spaces'
  ```
- ```
  u = '1+2+3+4'
  u.split(sep='+') # ['1','2','3','4']
  ```
  ```

Dictionaries

Dictionary

- AKA associative array or map
- Collection of key-value pairs
  - Keys are unique (repeats clobber existing)
  - Values need not be unique
- Syntax:
  - Curly brackets `{}` delineate start and end
  - Colons separate keys from values, commas separate pairs
  - `d = {'DeKalb': 783, 'Kane': 134, 'Cook': 1274, 'Will': 546}`
- No type constraints
  - `d = {'abc': 25, 12: 'abc', ('Kane', 'IL'): 123.54}`

Dictionary Examples

Keys	Key type	Values	Value type
Country names	`str`	Internet country	`str`
Decimal numbers	`int`	Roman numerals	`str`
States	`str`	Agricultural	`list of str`
Hospital patients	`str`	Vital signs	`tuple of floats`
Baseball players	`str`	Batting averages	`float`
Metric	`str`	Abbreviations	`str`
Inventory codes	`str`	Quantity in stock	`int`

[Deitel & Deitel]

Collections

- A dictionary is **not** a sequence
- Sequences are **ordered**
- Conceptually, dictionaries need no order
- A dictionary is a **collection**
- Sequences are also collections
- All collections have length (`len`), membership (`in`), and iteration (loop over values)
- Length for dictionaries counts number of key-value **pairs**
  - Pass dictionary to the `len` function
  - ```
    d = {'abc': 25, 12: 'abc', ('Kane', 'IL'): 123.54}
    len(d) # 3
    ```
    ```

Mutability

- Dictionaries are **mutable**, key-value pairs can be added, removed, updated
- (Each key must be immutable)
- Accessing elements parallels lists but with different "indices" possible
- Index → Key
- `d = {'DeKalb': 783, 'Kane': 134, 'Cook': 1274, 'Will': 546}`
- `d['Winnebago'] = 1023 # add a new key-value pair`
- `d['Kane'] = 342       # update an existing key-value pair`
- `d.pop('Will')         # remove an existing key-value pair`
- `del d['Winnebago']    # remove an existing key-value pair`

Key Restrictions

- Many types can be keys… including tuples
  - `d = {'abc': 25, 12: 'abc', ('Kane', 'IL'): 123.54}`
- …but the type must be immutable—lists cannot be keys
  - ~~`d = {['Kane', 'IL']: 2348.35, [1, 2, 3]: "apple"}`~~
- Why?

Key Restrictions

- Many types can be keys… including tuples
  - `d = {'abc': 25, 12: 'abc', ('Kane', 'IL'): 123.54}`
- …but the type must be **immutable***—lists cannot be keys
  - ~~`d = {['Kane', 'IL']: 2348.35, [1, 2, 3]: "apple"}`~~
- *technically, the type must be hashable, but having a mutable and still hashable type almost always causes problems
- Why?
  - Dictionaries are fast in Python because are implemented as hash tables
  - No matter how long the key, python hashes it stores values by hash
  - Given a key to lookup, Python hashes it and finds the value quickly (O(1))
  - If the key can mutate, the hash will not match the key!