Programming Principles in Python (CSCI 503)

Functions

Dr. David Koop

(some slides adapted from Dr. Reva Freedman)

if, else, elif, pass

-
  ```
  if a < 10:
      print("Small")
  else:
      if a < 100:
          print("Medium")
      else:
          if a < 1000:
              print("Large")
          else:
              print("X-Large")
  ```

-
  ```
  if a < 10:
      print("Small")
  elif a < 100:
      print("Medium")
  elif a < 1000:
      print("Large")
  else:
      print("X-Large")
  ```

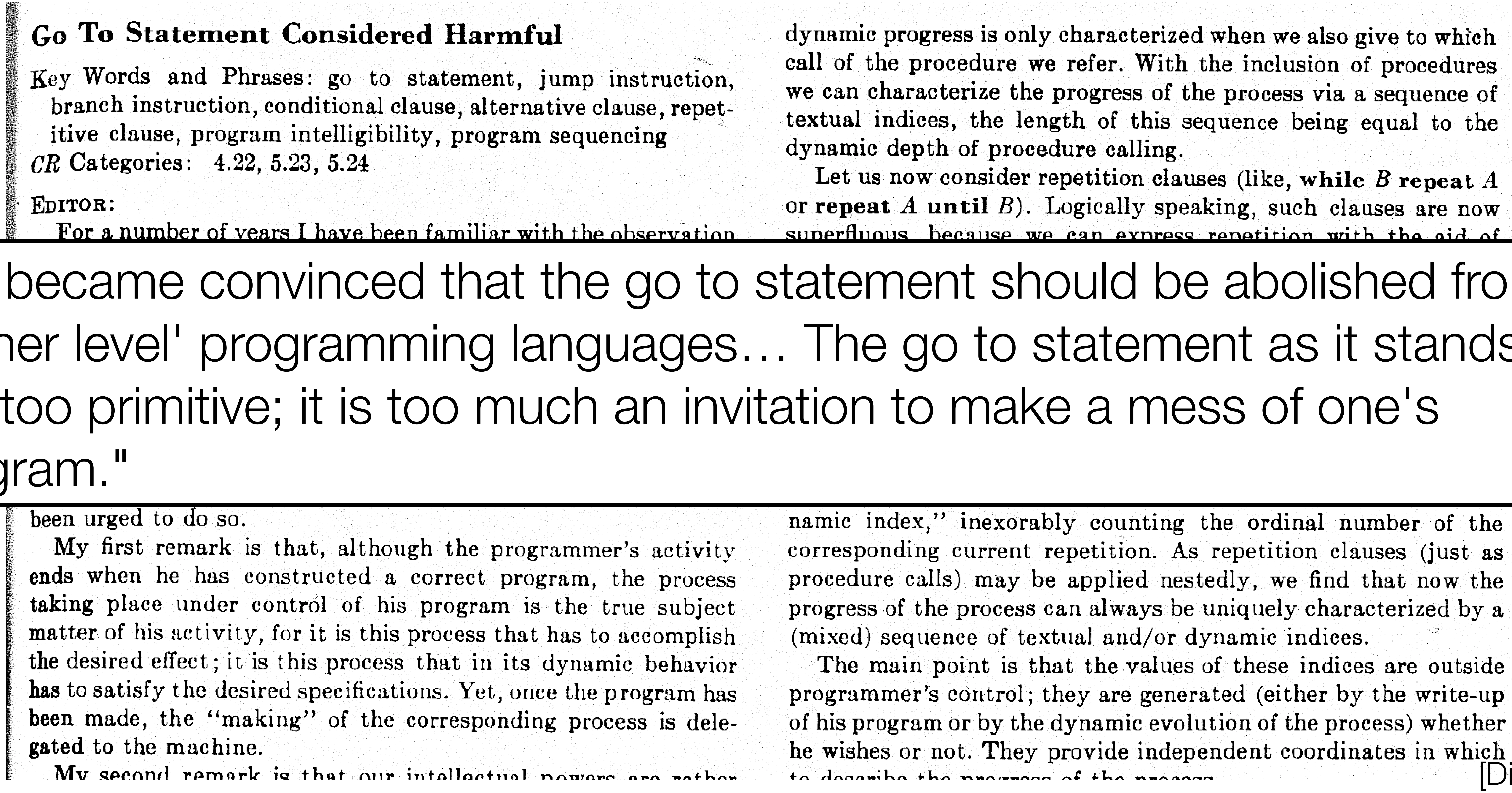
- Indentation is critical so else-if branches can become unwieldy (elif helps)
- Remember colons and indentation
- `pass` can be used for an empty block

while, break, continue

-
  ```
  while <boolean expression>:
      <loop-block>
  ```
- Condition is checked at the beginning and before each repeat
- `break`: immediately exit the current loop
- `continue`: stop loop execution and go back to the top of the loop, checking the condition again

-
  ```
  while d > 0:
      a = get_next_input()
      if a > 100:
          break
      if a < 10:
          continue
      d -= a
  ```

The Go To Statement Debate



Go To Statement Considered Harmful

Key Words and Phrases: go to statement, jump instruction, branch instruction, conditional clause, alternative clause, repetitive clause, program intelligibility, program sequencing

CR Categories: 4.22, 5.23, 5.24

EDITOR:

For a number of years I have been familiar with the observation

been urged to do so.

My first remark is that, although the programmer's activity ends when he has constructed a correct program, the process taking place under control of his program is the true subject matter of his activity, for it is this process that has to accomplish the desired effect; it is this process that in its dynamic behavior has to satisfy the desired specifications. Yet, once the program has been made, the "making" of the corresponding process is delegated to the machine.

My second remark is that our intellectual powers are rather

dynamic progress is only characterized when we also give to which call of the procedure we refer. With the inclusion of procedures we can characterize the progress of the process via a sequence of textual indices, the length of this sequence being equal to the dynamic depth of procedure calling.

Let us now consider repetition clauses (like, **while** *B* **repeat** *A* or **repeat** *A* **until** *B*). Logically speaking, such clauses are now superfluous, because we can express repetition with the aid of

namic index,'' inexorably counting the ordinal number of the corresponding current repetition. As repetition clauses (just as procedure calls) may be applied nestedly, we find that now the progress of the process can always be uniquely characterized by a (mixed) sequence of textual and/or dynamic indices.

The main point is that the values of these indices are outside programmer's control; they are generated (either by the write-up of his program or by the dynamic evolution of the process) whether he wishes or not. They provide independent coordinates in which to describe the progress of the process.

"…I became convinced that the go to statement should be abolished from all 'higher level' programming languages… The go to statement as it stands is just too primitive; it is too much an invitation to make a mess of one's program."

[Dijkstra, 1968]

Loop Styles

- Loop-and-a-Half

```
d = get_data() # priming rd
while check(d):
    # do stuff
    d = get_data()
```

- Infinite-Loop-Break

```
while True:
    d = get_data()
    if check(d):
        break
    # do stuff
```

- Assignment Expression (Walrus)

```
while check(d := get_data):
    # do stuff
```

For Loop

- for loops in Python are really for-each loops
- Always an element that is the current element
  - Can be used to iterate through iterables (containers, generators, strings)
  - Can be used for counting

-
  ```
  for i in range(5):
      print(i) # 0 1 2 3 4
  ```
- `range` generates the sequences of integers, one at a time
  - `range(n)` → `0, 1, …, n-1`
  - `range(start, n)` → `start, start + 1, …, start + (n-1)`
  - `range(start, n, step)`
    → `start, start + step, …, start + (n-1)*step`

Assignment 2

- Due Monday
- Python control flow and functions
- Do not use containers like lists!
- The 7x+-1 function
- Make sure to follow instructions
  - Name the submitted file a2.ipynb
  - Put your name and z-id in the first cell
  - Label each part of the assignment using markdown
  - Make sure to produce output according to specifications

Functions

- Call a function `f`: `f(3)` or `f(3,4)` or … depending on number of parameters
-
  ```
  def <function-name>(<parameter-names>):
      """Optional docstring documenting the function"""
      <function-body>
  ```
- `def` stands for function definition
- docstring is convention used for documentation
- Remember the **colon** and **indentation**
- Parameter list can be empty: `def f(): …`

Functions

- Use `return` to return a value
-
  ```
  def <function-name>(<parameter-names>):
      # do stuff
      return res
  ```
- Can return more than one value using commas
-
  ```
  def <function-name>(<parameter-names>):
      # do stuff
      return res1, res2
  ```
- Use **simultaneous assignment** when calling:
  - `a, b = do_something(1,2,5)`
- If there is no return value, the function returns `None` (a special value)

Return

- As many return statements as you want
- Always end the function and go back to the calling code
- Returns do not need to match one type/structure (generally not a good idea)
-
```
def f(a,b):
    if a < 0:
        return -1
    while b > 10:
        b -= a
        if b < 0:
            return "BAD"
    return b
```

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**

Global read

-
  ```
  def f(): # no arguments
      print("x in function:", x)

  x = 1
  f()
  print("x in main:", x)
  ```
- Output:

  -
    ```
    x in function: 1
    x in main: 1
    ```
- Here, the `x` in `f` is read from the global scope

Try to modify global?

-
  ```
  def f(): # no arguments
      x = 2
      print("x in function:", x)

  x = 1
  f()
  print("x in main:", x)
  ```
- Output:

  -
    ```
    x in function: 2
    x in main: 1
    ```
- Here, the `x` in `f` is in the local scope

Global keyword

-
  ```
  def f(): # no arguments
      global x
      x = 2
      print("x in function:", x)

  x = 1
  f()
  print("x in main:", x)
  ```
- Output:

  -
    ```
    x in function: 2
    x in main: 2
    ```
- Here, the `x` in `f` is in the global scope because of the global declaration

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;
    }
    ```

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?

Neither

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!

Python lists

- Stores a collection of objects in order
- Created using square brackets: `[0,1,2,3,4]`
- Lists are **mutable**: we can change them in place:

  -
    ```
    my_list = [0,1,2,3,4]
    my_list.append(5)
    my_list # [0,1,2,3,4,5]
    ```
- Remember that integers, strings, floats are not mutable (immutable)

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!

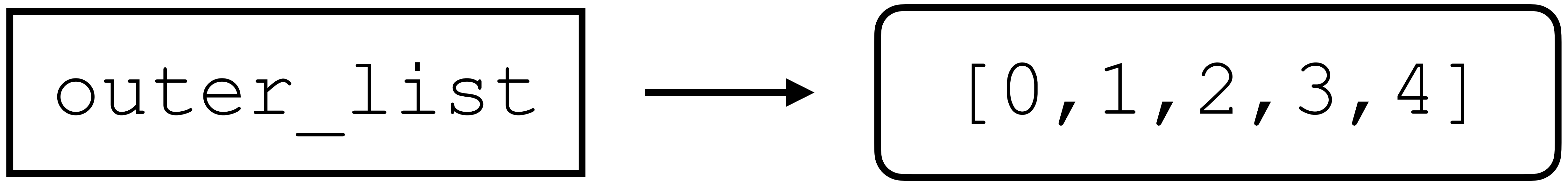
What's going on?

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]
```

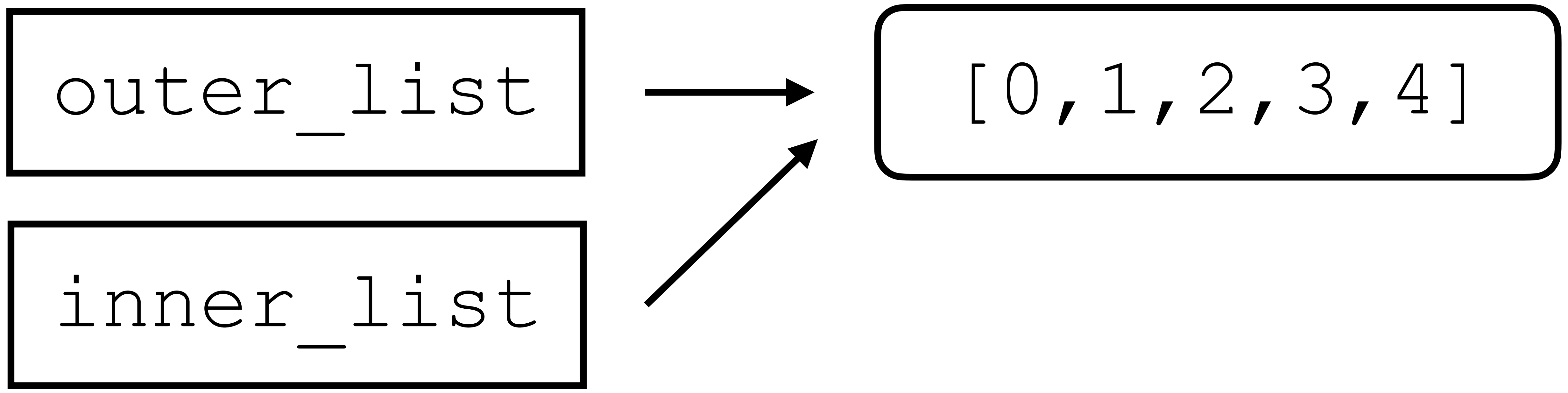


outer_list

[0,1,2,3,4]

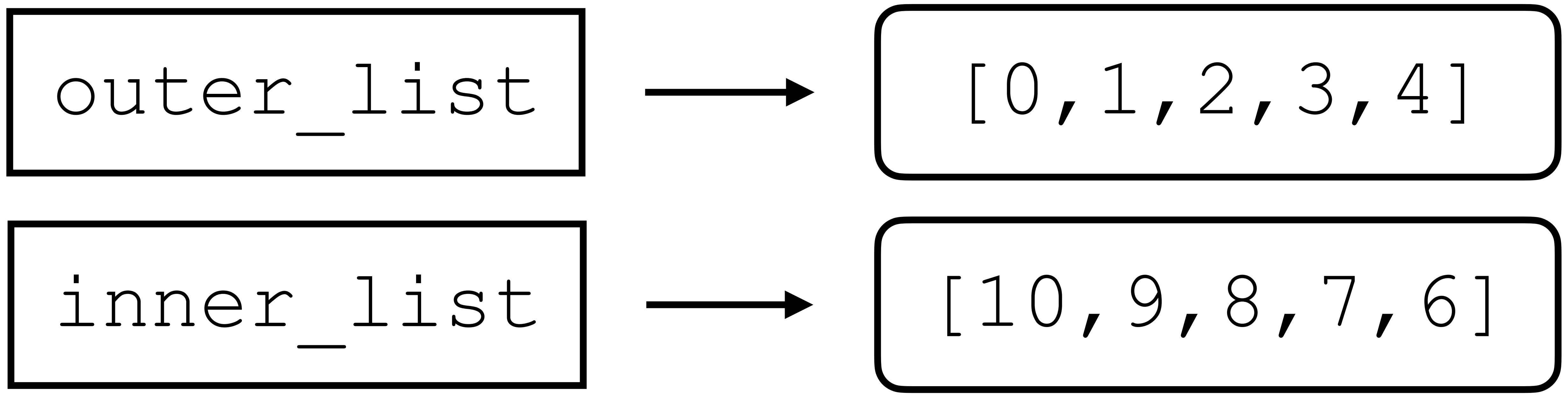
Example 1

-
  ```
  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]
  ```

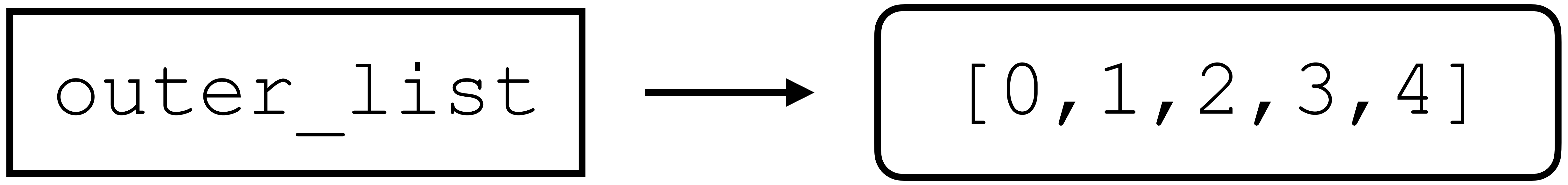


outer_list → [0,1,2,3,4]

inner_list → [10,9,8,7,6]

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]
```

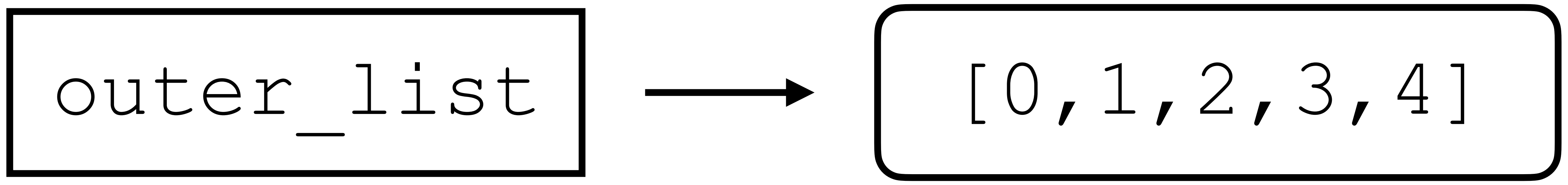


outer_list

[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]
  ```



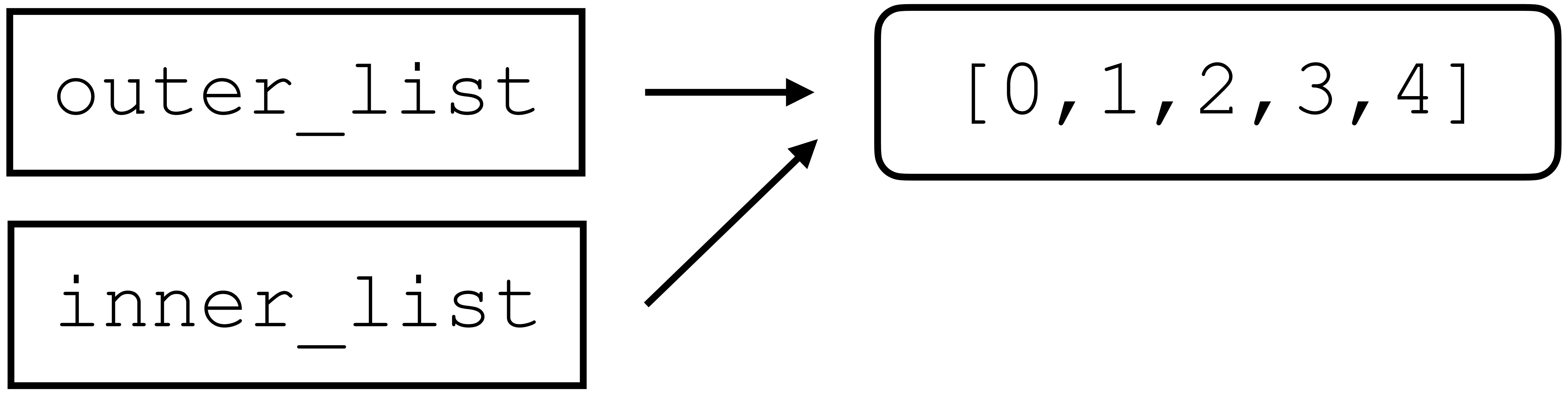
outer_list

[0,1,2,3,4]

Example 2

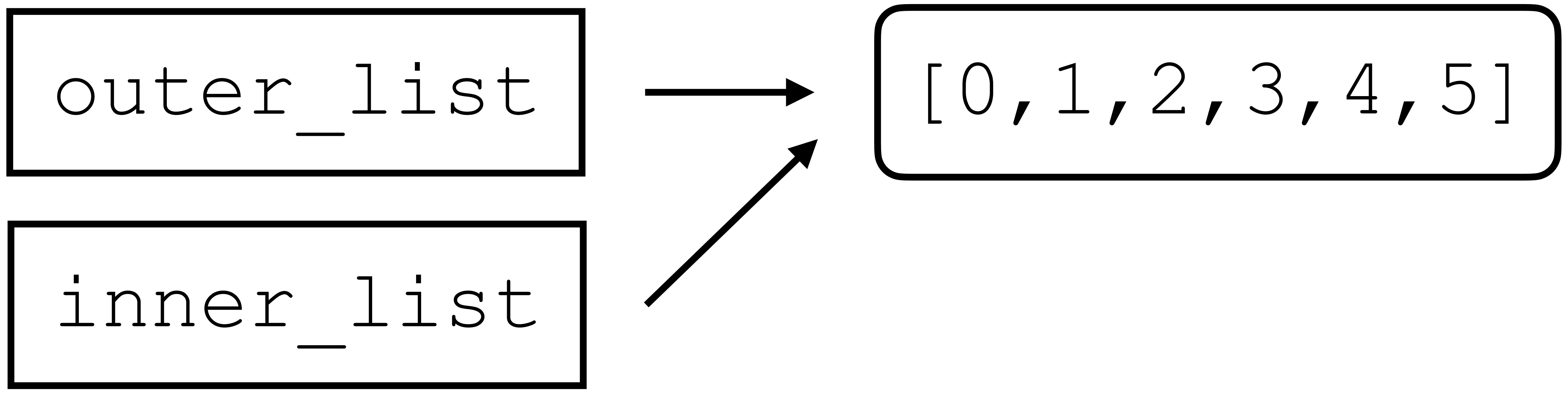
- **def change_list(inner_list):**

```
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]
```



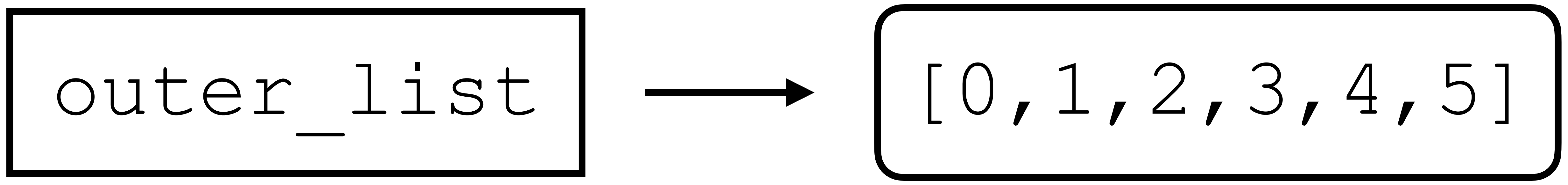
Example 2

-
  ```
  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]
  ```



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 list 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
  ```

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']
  ```