Control Constructs: Loops in Python

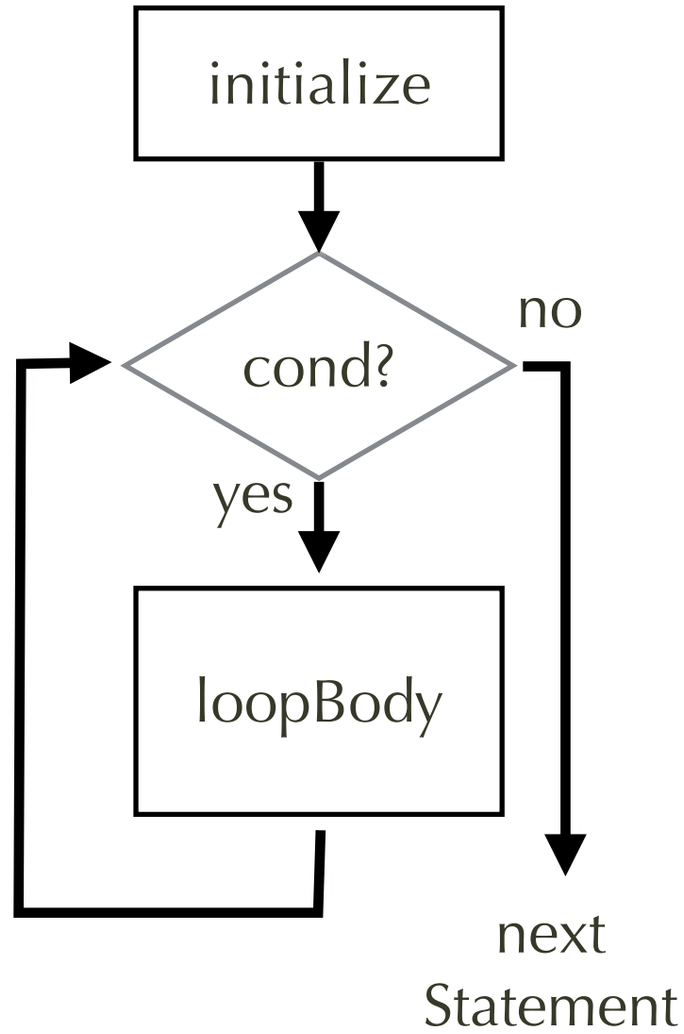
Prof. Pai H. Chou
National Tsing Hua University

Outline

- `while` loops
  - `else`
  - `break`
  - `continue`
- `for` loops
  - index style vs element style
  - unpacking for-loops
  - enumerate() for iterating over index

`while` loop

- syntax:
  `while` *cond*:
      loopBody
- loops as long as *cond* is true.
  - when cond is False, loop exits and executes next statement, if any



`break` statement inside while-loop

- forces the loop to exit!
  - regardless of the while-condition
- **`while`** *cond*:

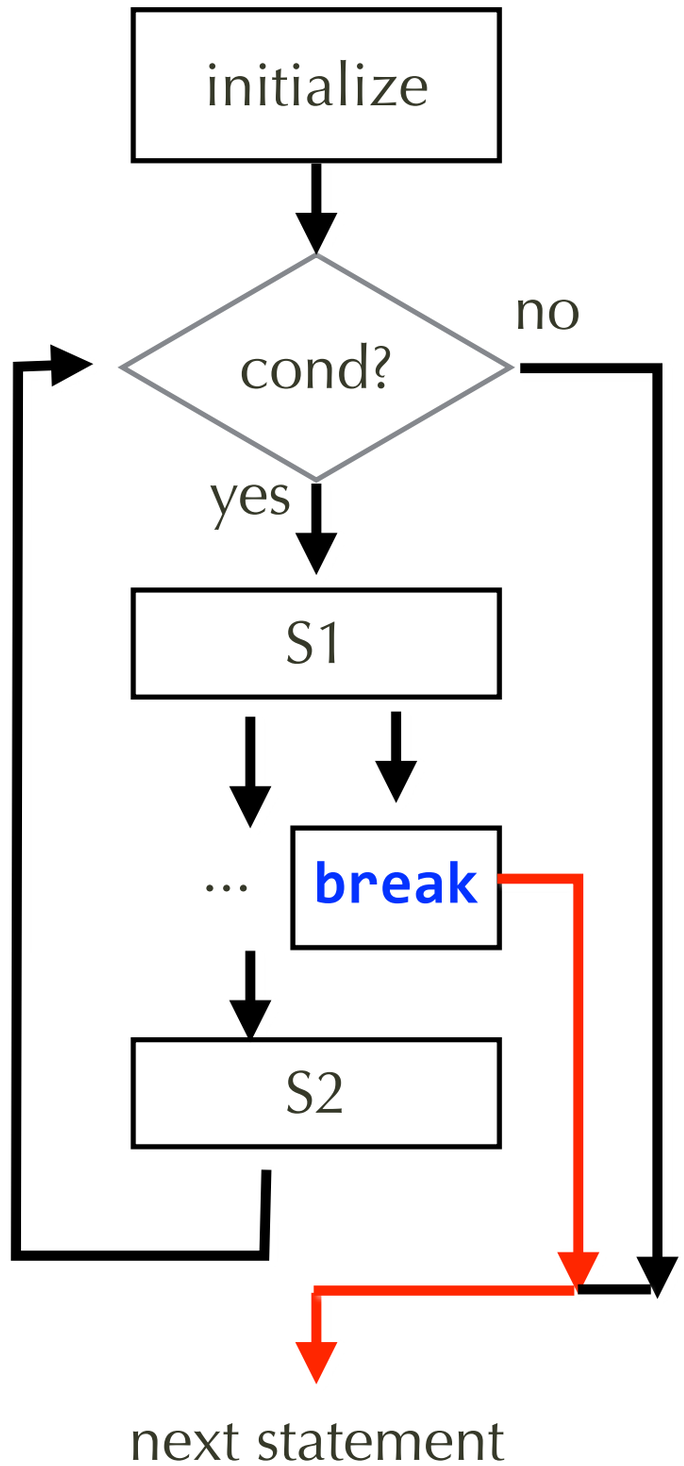
  S1

  …

  `break`

  …

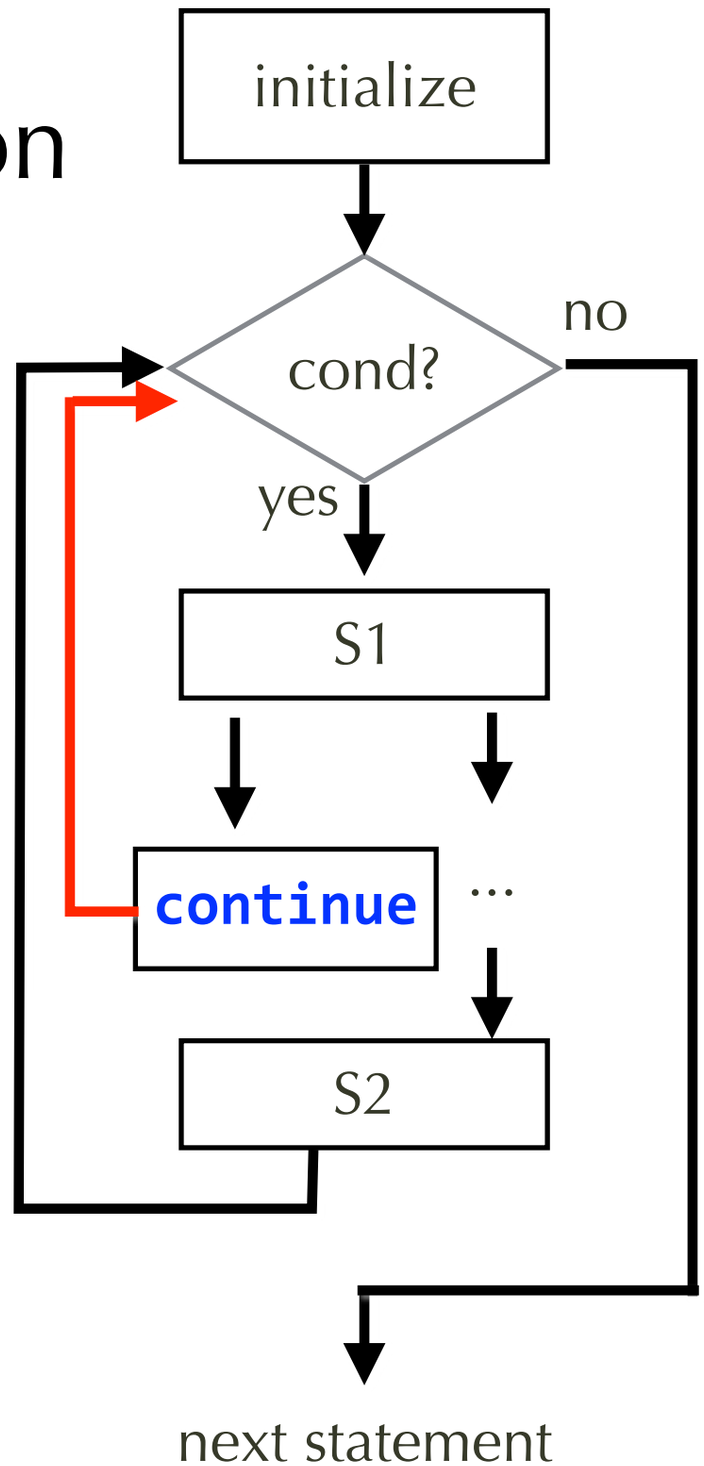
  S2



`continue` statement inside while-loop

- forces loop to test next iteration
  - skip the rest of the loop for the current iteration
- **`while`** *cond*:

```
    S1
    ...
    continue
    ...
    S2
```



Example 1 for while-loop: product() function

- analogous to `sum()` but multiplies instead

```
>>> sum([1, 3, 5, 7, 9])
25
>>> sum([])
0
```

- `sum()` is built-in, but `product()` is not...

```
>>> product([1, 3, 5, 7, 9])
945
>>> product([])
1
```

Implementation of product() using while-loop

- initialization: (lines 2-3)
  - p = 1 # the partial product
  - i = 0 # the current position
- if len(L) == 0, skip loop
  - (jumps from line 4 to line 7) => return 1
- if len(L) == 1, enter loop (lines 4)
  - (lines 5-6) p = 1 * L[0]; i = 0 + 1
  - (line 4) while loop exits; (jumps to 7) => return L[0]

```
1 def product(L):
2     p = 1
3     i = 0
4     while i < len(L):
5         p = p * L[i]
6         i = i + 1
7     return p
```

Implementation of product() using while-loop

- if len(L) == 2
  - after looping once (lines 2-6),
    - p = L[0]
    - i = 1
  - (back to line 4) while condition (1 < 2) True, loop again
    - (lines 5-6) p = L[0] * L[1]; i = 2
  - (back to line 4) while condition (2 < 2) False, exit loop
    - (line 7) return L[0] * L[1]

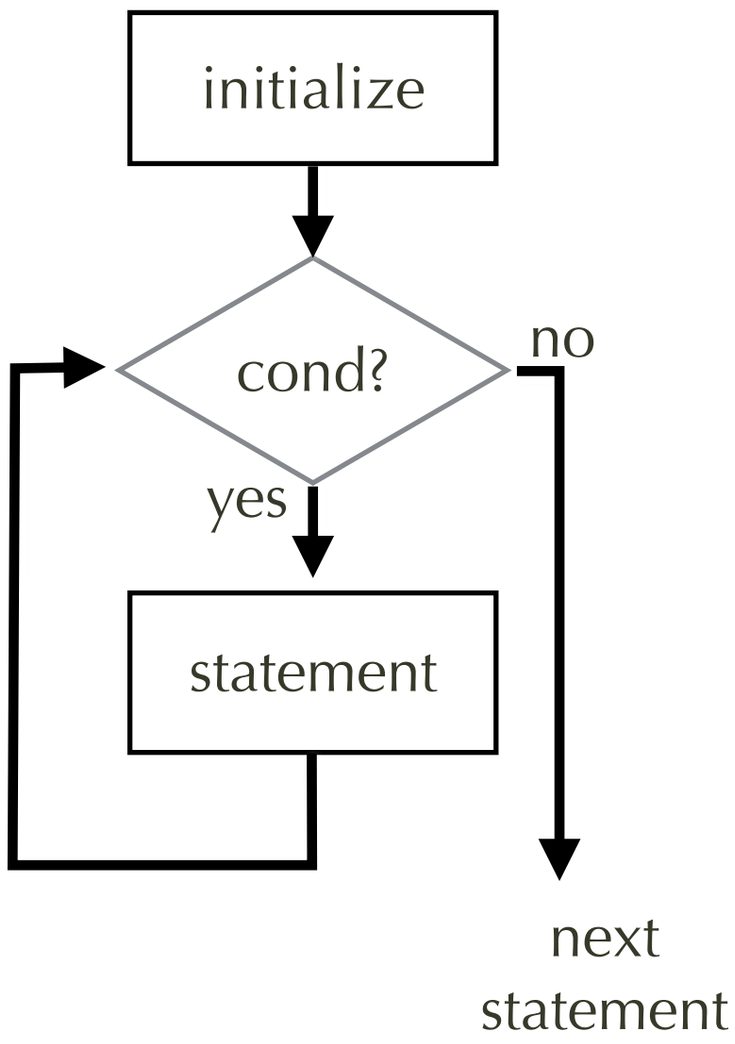
```
1 def product(L):
2     p = 1
3     i = 0
4     while i < len(L):
5         p = p * L[i]
6         i = i + 1
7     return p
```

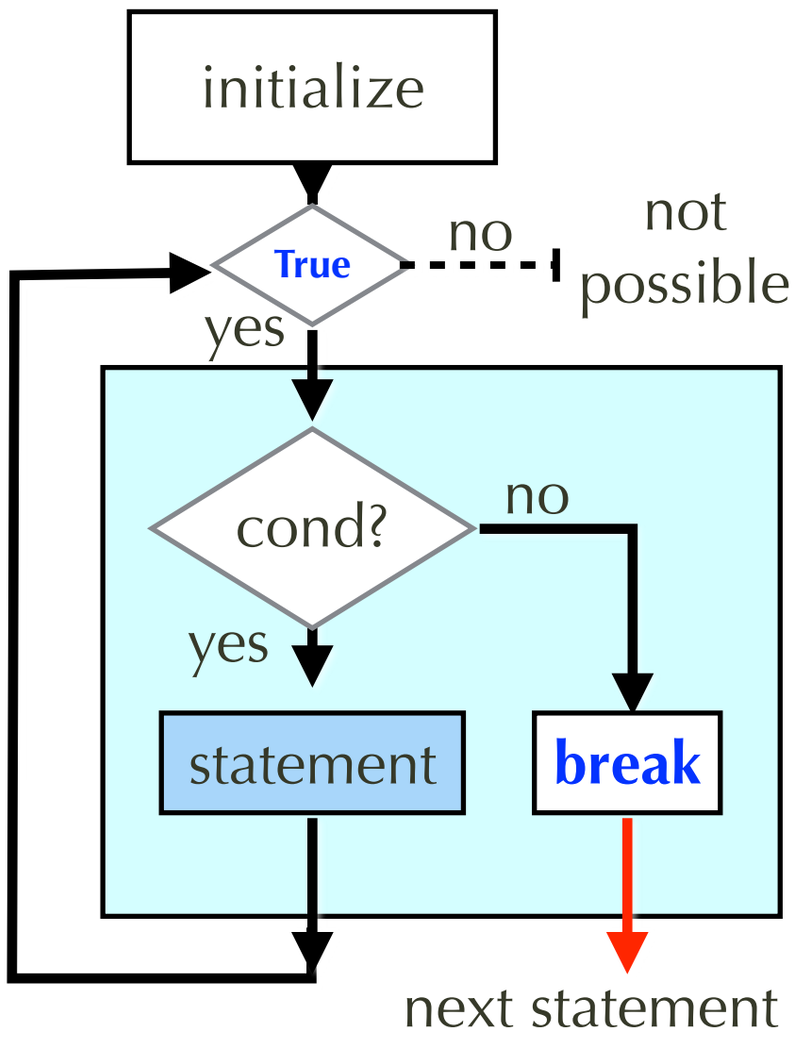
`while`-cond can be rewritten as infinite loop a conditional `break`

```
while (cond):
    statement
```

is equivalent to

```
while True:
    if not (cond):
        break
    statement
```





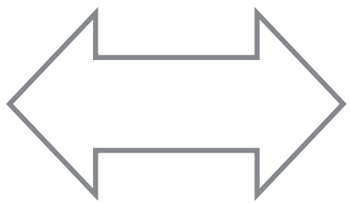
Rewriting example 1 using infinite loop + break

```
while cond:
    statement
```

can be rewritten as

```
while True:
    if not cond:
        break
    statement
```

```
def product(L):
    p = 1
    i = 0
    while i < len(L):
        p = p * L[i]
        i = i + 1
    return p
```



```
def product(L):
    p = 1
    i = 0
    while True:
        if i >= len(L):
            break
        p = p * L[i]
        i = i + 1
    return p
```

Example 2: index() function

- function version of list index method
  - returns the index of the first matched value
  - returns -1 if not found in list

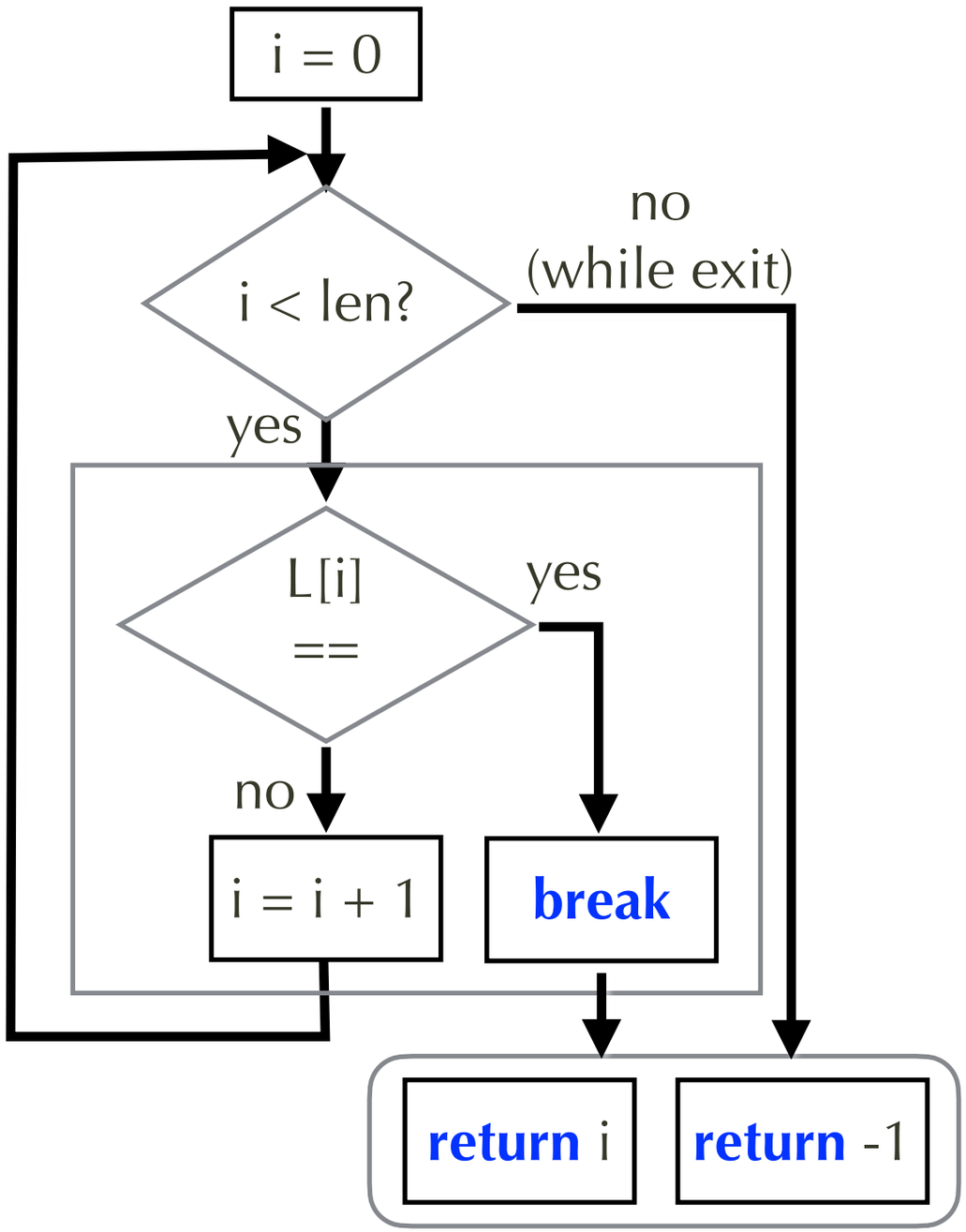
method version (built-in)

```
>>> L = ['a', 'b', 'c', 'd', 'e']
>>> L.index('c')
2
>>> L.index('z')
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
ValueError: 'z' is not in list
```

function version

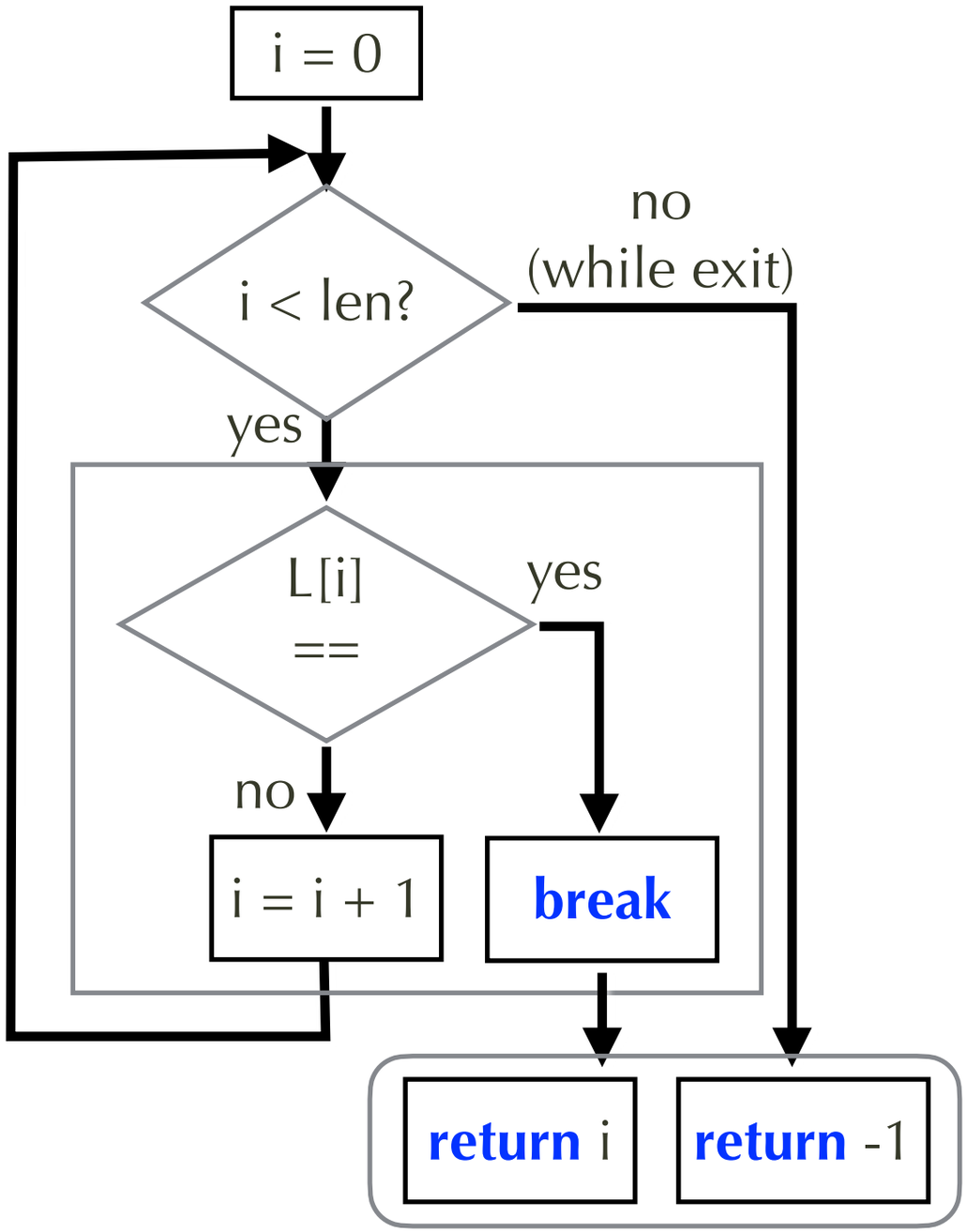
```
>>> index(L, 'c')
2
>>> index(L, 'z')
-1
```

Example 2: first attempt




```
1 def index(L, val):
2     i = 0
3     while i < len(L):
4         if L[i] == val:
5             break
6         i = i + 1
7     return i if i < len(L) else -1
```

Example 2: issue with first attempt



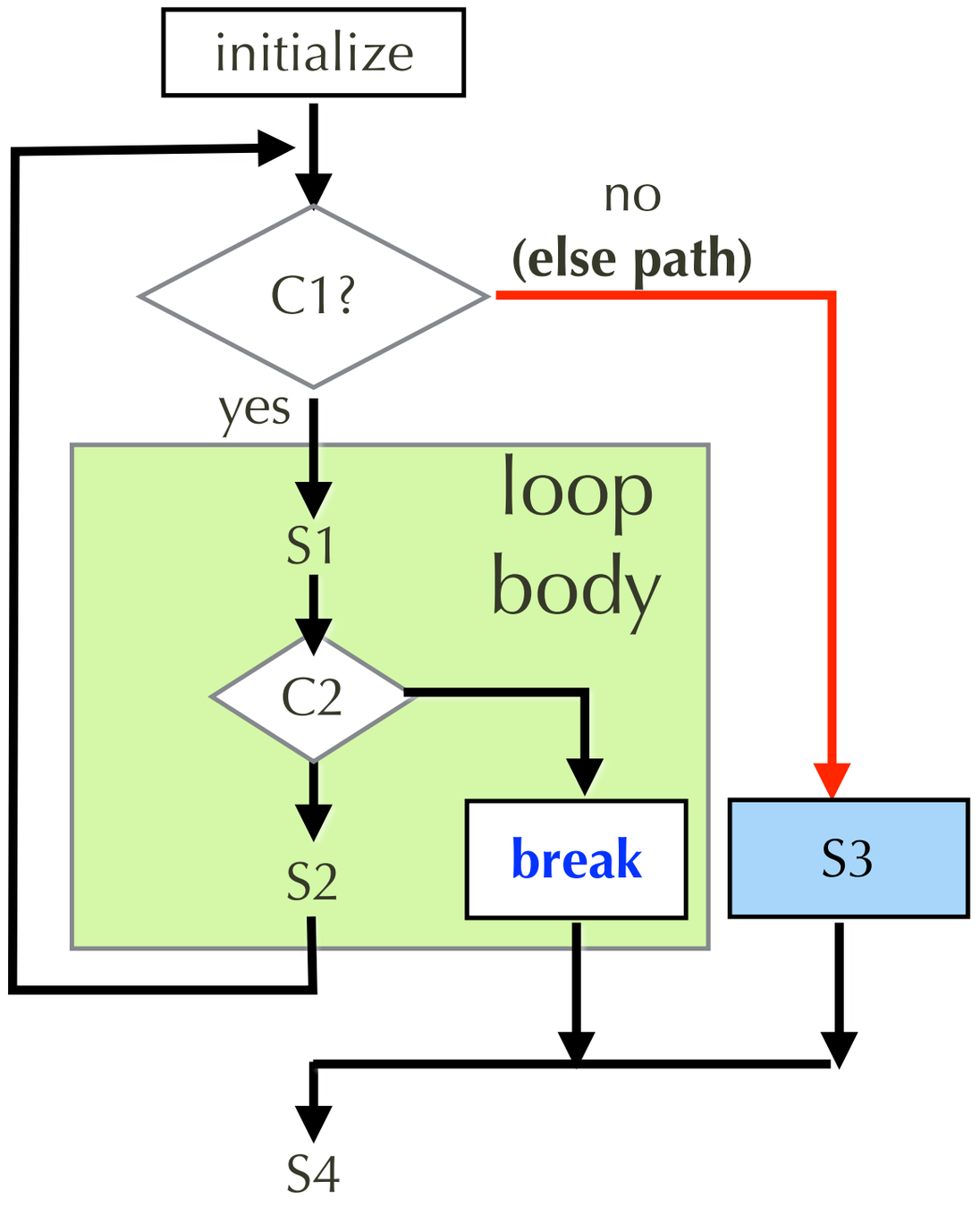

```
1 def index(L, val):
2     i = 0
3     while i < len(L):
4         if L[i] == val:
5             break
6         i = i + 1
7     return i if i < len(L) else -1
```

same condition tested again...
seems redundant!

Python while-else statement

- Syntax

```
while C1:
    S1
    if C2:
        break
    S2
else:
    S3
S4
```



- S3
  - if C1 tests to False, execute else (S3) clause and then exit the loop
  - if break, then just exit loop without executing else (S3)

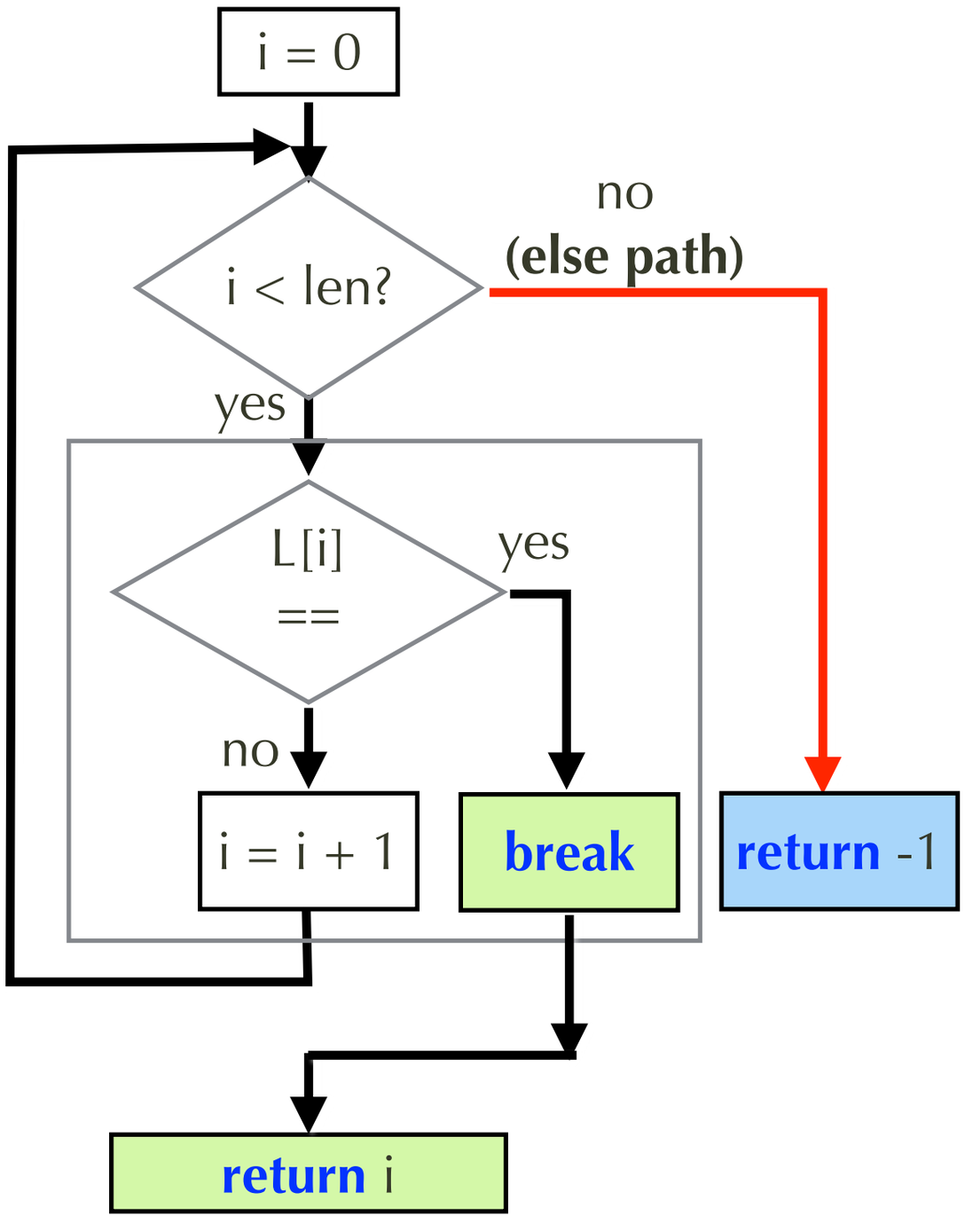
Equivalent while-else statement

```
while C1:
    S1
    if C2:
        break
    S2
else:
    S3
S4
```

```
while True:
    if C1:
        S1
        if C2:
            break
        S2
    else:
        S3
        break
S4
```

- while-else is analogous to if-else for controlling loop exit in equivalent code

Example 2, second attempt: use Python while-else statement

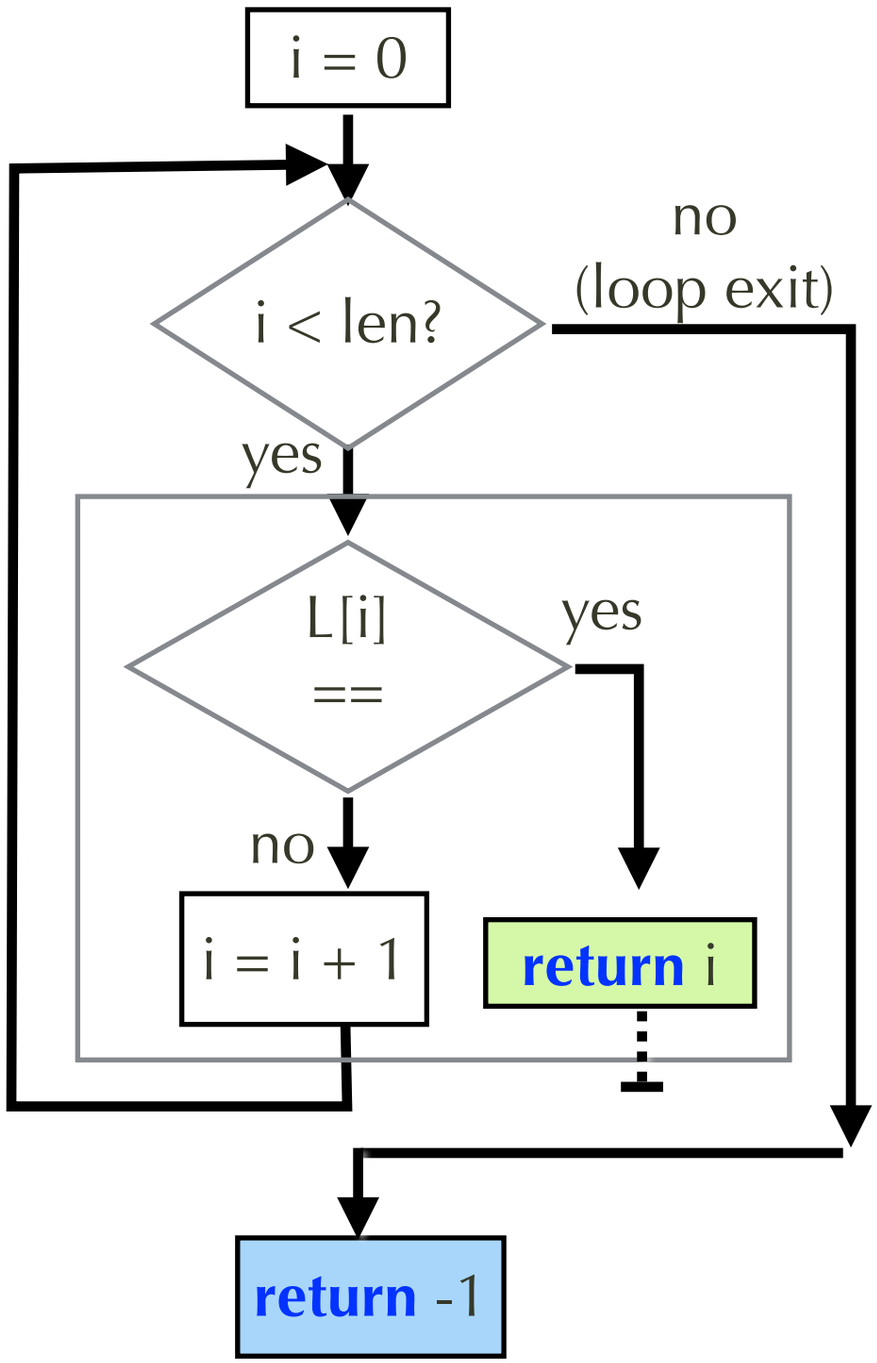



```
1 def index(L, val):
2     i = 0
3     while i < len(L):
4         if L[i] == val:
5             break
6         i = i + 1
7     return i if i < len(L) else -1
```

can be rewritten as

```
1 def index(L, val):
2     i = 0
3     while i < len(L):
4         if L[i] == val:
5             break
6         i = i + 1
7     else: # while loop tests False
8         return -1
9     return i
```

Example 2, third attempt: return directly from function




```
1 def index(L, val):
2     i = 0
3     while i < len(L):
4         if L[i] == val:
5             break
6         i = i + 1
7     else: # while loop tests False
8         return -1
9     return i
```

can be rewritten as

```
1 def index(L, val):
2     i = 0
3     while i < len(L):
4         if L[i] == val:
5             return i
6         i = i + 1
7     return -1
```

eliminates need for "else"

Summary of Difference between Examples 1 and 2

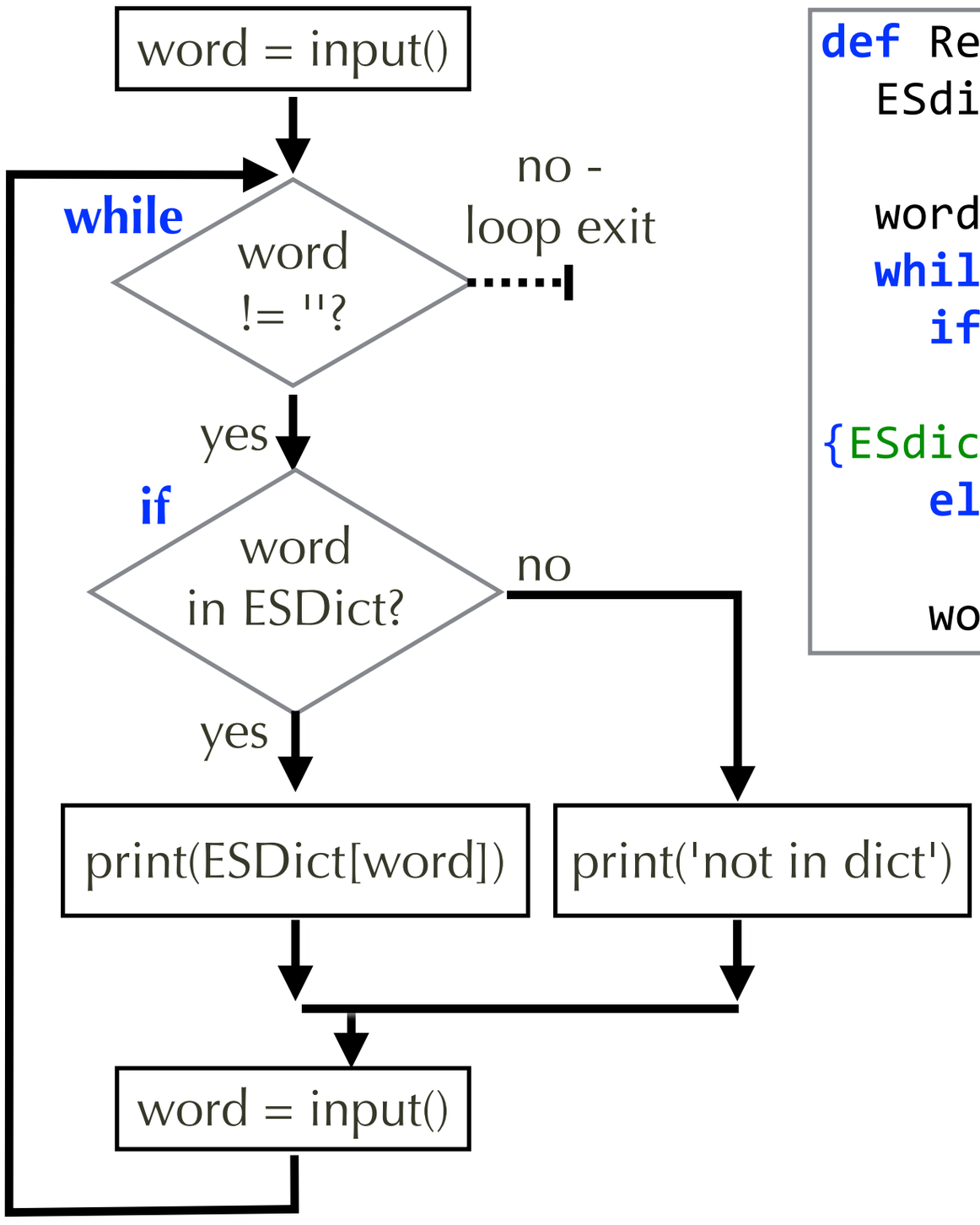
- Example 1: (product of list)
  - must compute over all values in list
- Example 2: (index of first match in list)
  - can terminate as soon as finding first match, but worst-case must compare all elements in list!
  - termination by **`break`** statement
  - alternatively, by **`return`** statement (if from function)

Example 3: repeated dictionary lookup

- ask user for word, display definition
- keep asking until user types empty line

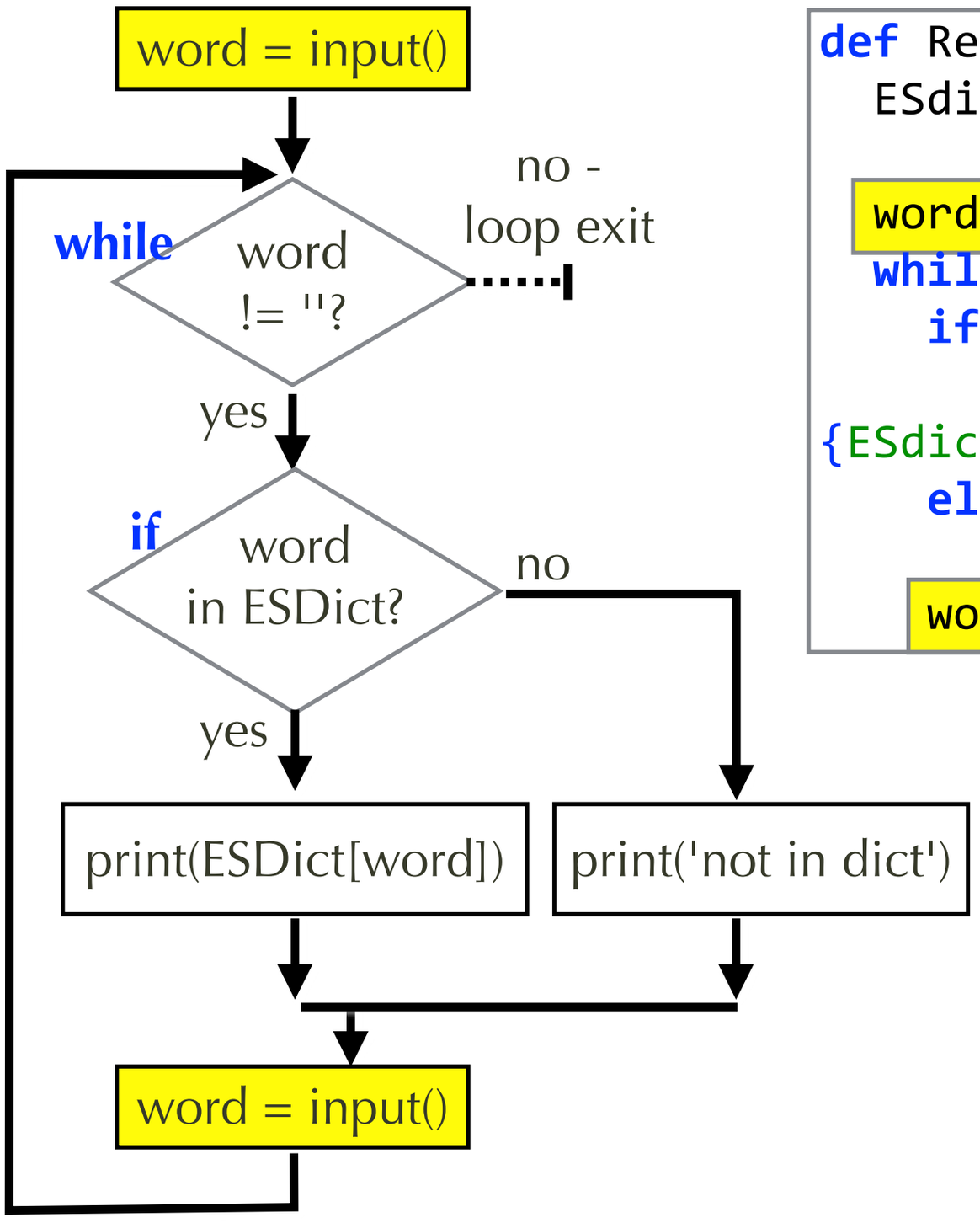
```
>>> RepeatedLookup()
enter an English word: one
one in Spanish is uno
enter an English word: amigo
amigo not in dictionary
enter an English word:
>>>
```

Example 3: first attempt




```
def RepeatedLookup():
  ESdict = { 'one': 'uno', \
        'dos': 'two', 'three': 'tres'}
  word = input('enter English:')
  while word != '':
    if word in ESdict:
      print(f'{word} in Spanish is \
{ESdict[word]}')
    else:
      print(f'{word} not in dict')
    word = input('enter English:')
```

Example 3: issue with first attempt: same code twice!

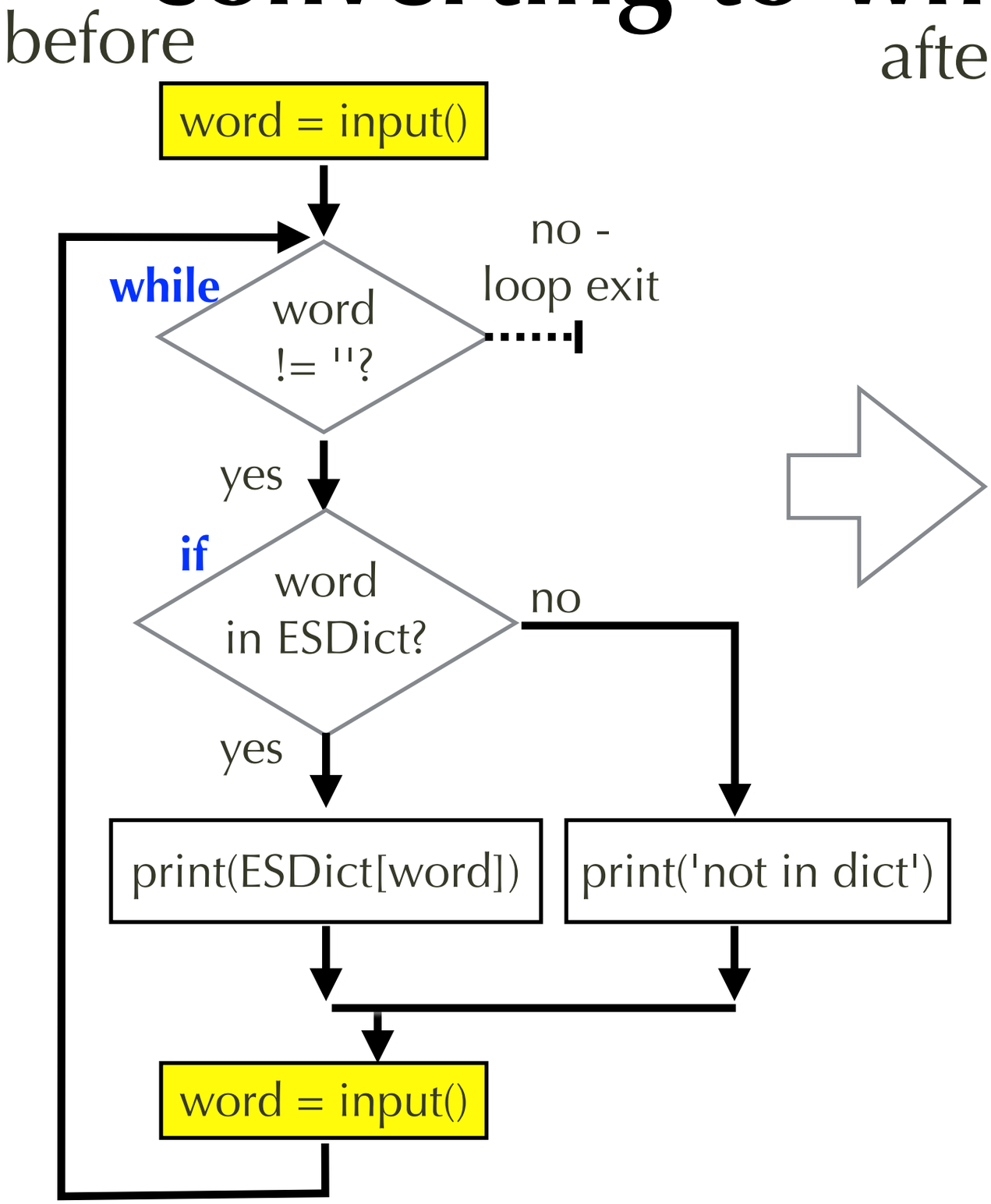


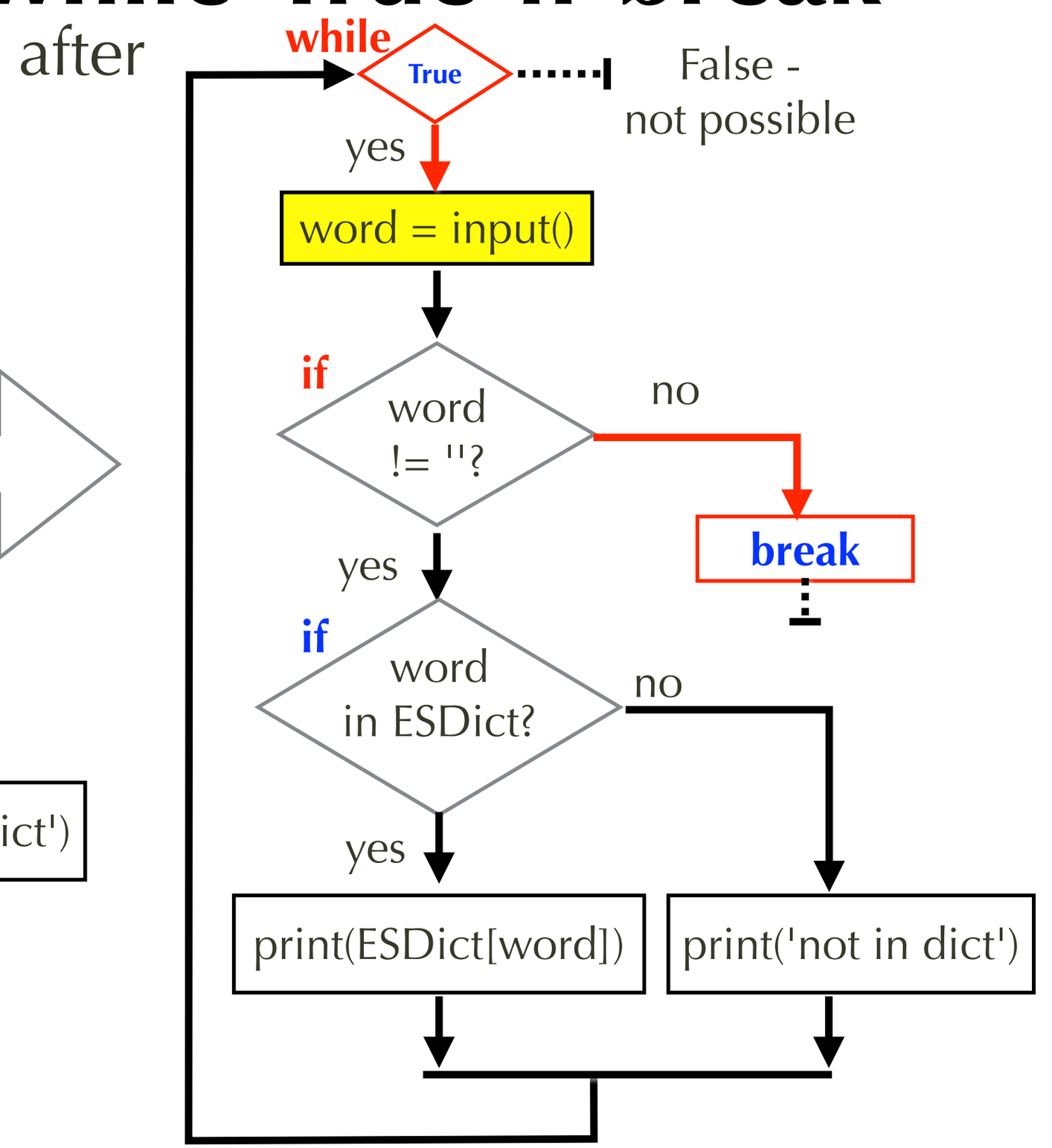

```
def RepeatedLookup():
  ESdict = { 'one': 'uno', \
        'dos': 'two', 'three': 'tres'}
  word = input('enter English:')
  while word != '':
    if word in ESdict:
      print(f'{word} in Spanish is \
{ESdict[word]}')
    else:
      print(f'{word} not in dict')
    word = input('enter English:')
```

functional, but not good

- redundant
- likely make mistakes

Example 3: eliminate redundancy by converting to while-True-if-break





Example 3: revised code with infinite loop + break

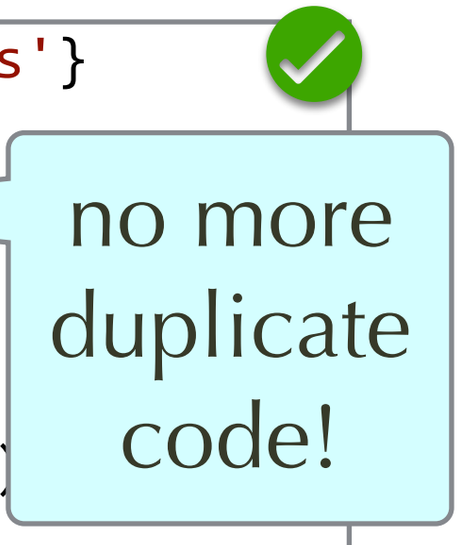
before

```
1  ESdict = { 'one': 'uno', 'dos': 'two', 'three': 'tres'}
2  word = input('enter an English word:')
3  while word != '':
4      if word in ESdict:
5          print(f'{word} in Spanish is {ESdict[word]}')
6      else:
7          print(f'{word} not in dictionary')
8      word = input('enter an English word:')
```

after

- **while True** to enter loop anyway; conditional break to exit loop

```
1  ESdict = { 'one': 'uno', 'dos': 'two', 'three': 'tres'}
2  while True:
3      word = input('enter an English word:')
4      if word == '':
5          break # exits loop
6      if word in ESdict:
7          print(f'{word} in Spanish is {ESdict[word]}')
8      else:
9          print(f'{word} not in dictionary')
```



Summary of infinite loop terminated by `break`

- `while True:`

  loopBody

  is called an "infinite loop"

- This just means the loop is not terminated by the while condition

- but the loop can still terminate by

  - `break` # exits innermost enclosing loop!

  - `return` from function or exit() program

Example 4: stack-command interpreter

- stack: last-in first-out data structure
  - push(value): pushes value on stack
  - pop(): pops value from stack
- Commands:
  - `push` v1 v2 ...
  - `pop`
  - `show`
  - `quit`

```
>>> StackInterpreter()
command? show
[]
command? push apple oranges mango
command? show
['apple', 'oranges', 'mango']
command? pop
mango
command? pop
oranges
command? quit
>>>
```

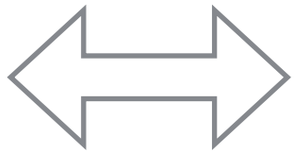
Example 4: stack-command interpreter

- line.split()
  - extracts words separated by blanks
  - words is a list of string typed by user
- if-elif-...-else to handle different commands
  - in all cases, do the loop again

```
def StackInterpreter():
    L = []
    while True:
        line = input('command? ')
        words = line.split()
        if len(words) == 0:
            pass
        elif words[0] == 'show':
            print(L)
        elif words[0] == 'push':
            L.extend(words[1:])
        elif words[0] == 'pop':
            print(L.pop())
        elif words[0] == 'quit':
            break
        else:
            print('unknown command')
```

Example 4 alternative solution: `continue`

```
def StackInterpreter():
  L = []
  while True:
    line = input('command? ')
    words = line.split()
    if len(words) == 0:
      pass
    elif words[0] == 'show':
      print(L)
    elif words[0] == 'push':
      L.extend(words[1:])
    elif words[0] == 'pop':
      print(L.pop())
    elif words[0] == 'quit':
      break
    else:
      print('unknown command')
```



```
def StackInterpreter():
  L = []
  while True:
    line = input('command? ')
    words = line.split()
    if len(words) == 0:
      continue
    if words[0] == 'show':
      print(L)
      continue
    if words[0] == 'push':
      L.extend(words[1:])
      continue
    if words[0] == 'pop':
      print(L.pop())
      continue
    if words[0] == 'quit':
      break
    # no need for else keyword
    print('unknown command')
```

Example 4 alternative solution: `continue`

- use of **`continue`**
  - jumps to top of loop to test **while** condition
  - i.e., skips the rest of this iteration
- allows **elif** to be converted into **if**
  - may be "cleaner" => can safely assume no code after if-elif-else in the loop will be executed!

```
def StackInterpreter():
  L = []
  while True:
    line = input('command? ')
    words = line.split()
    if len(words) == 0:
      continue
    if words[0] == 'show':
      print(L)
      continue
    if words[0] == 'push':
      L.extend(words[1:])
      continue
    if words[0] == 'pop':
      print(L.pop())
      continue
    if words[0] == 'quit':
      break
    print('unknown command')
```

`for` loop

- syntax:
  `for` *var* **`in`** *sequence*`:`
      loopBody
- Assign each value of sequence to var before executing loop body once

```
def product(L):
    p = 1
    for n in L:
        p = p * n
    return p
```

product() function implemented in for loop

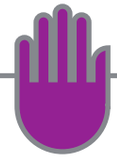
Pythonic vs. index style `for`-loop

- Python: iterate over sequence (of values)
- index style: iterate over index range, then index the list/tuple/array using index

```
def product(L):
    p = 1
    for n in L:
        p = p * n
    return p
```


"Pythonic", used by most "scripting" languages

```
def product(L):
    p = 1
    for i in range(len(L)):
        p = p * L[i]
    return t
```



avoid if possible

```
def product(L):
    p = 1
    i = 0
    while i < len(L):
        p = p * L[i]
        i = i + 1
    return p
```

very awkward, easy to forget to increment i

general `for`-loop with unpacking

- Example: input = list of (width, height) tuples, want to calculate the total area

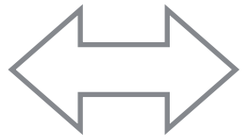
```
def totalArea(L):
    t = 0
    for width, height in L:  # unpacks each tuple in L
        t = t + width * height
    return t
```

```
>>> listOfDimensions = [(5, 2), (3, 7), (7, 8)]
>>> totalArea(listOfDim)
87
```

`enumerate()` function for if you need the index

- Example: index() function for finding first matched element

```
1 def index(L, val):
2     i = 0
3     while i < len(L):
4         if L[i] == val:
5             return i
6         i = i + 1
7     return -1
```



```
1 def index(L, val):
2     for i, v in enumerate(L):
3         if v == val:
4             return i
5     return -1
```

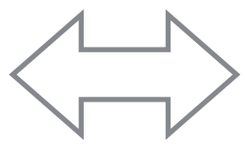
- `enumerate(L)` generates `[(0, L[0]), (1, L[1]), ...]` so that for loop can use unpacking assignment to get both the index and each element of the list

why `enumerate()` is preferred to explicit index?

- No need to apply indexing operator
- No chance of forgetting to i = i + 1
- i and v are always maintained together
- enumerate(L) can start at some value other than zero

preferred!

```
1 def index(L, val):
2     i = 0
3     while i < len(L):
4         if L[i] == val:
5             return i
6         i = i + 1
7     return -1
```



```
1 def index(L, val):
2     for i, v in enumerate(L):
3         if v == val:
4             return i
5     return -1
```