COSC2429 Introduction to Programming

Lists

Quang Tran <quang.tran@rmit.edu.vn>

Outline

- List elements and list length
- List operators: index, in, not in, +, *
- List slices
- Lists are mutable
- Append vs concatenation
- List and for loop
- Exercise: Calculating the average of a list of integers
- Objects and references
- List aliasing
- List cloning
- Summary of list methods
- Using list as parameters
- Pure functions
- Functions that produce lists
- List comprehensions
- Nested lists
- Strings and lists
- List conversion function

List elements and list length

- A **list** is a sequential collection of **elements** of any types. Each element is identified by an **index**.
- There are several ways to create a new list. The simplest is to enclose the elements in square brackets ([and]).
- A list within another list is often call a **sub-list**.
- Similar to strings, the **len** function returns the number of elements in the list.

```
vocabulary = ["iteration", "selection", "control"]
numbers = [17, 123]
empty = []
mixed_list = ["hello", 2.0, 5*2, [10, 20]]
new_list = [numbers, vocabulary]

print(numbers)
print(len(numbers))
print(mixed_list)
print(len(mixed_list))
print(new_list)
```

List operators: **index**, **in**, **not in, +, ***

- The **index**, **in**, **not in, +, *** operator work exactly like they do in strings.
- Just like string, a positive index (starts from 0) locates an element from the left of the list where as a negative index (starts from -1) locates an element from the right of the list.

```
numbers = [17, 123, 87, 34, 66, 8398, 44]
print(numbers[2])
print(numbers[9 - 8])
print(numbers[-2])
print(numbers[len(numbers) - 1])

fruit = ["apple", "orange", "banana", "cherry"]
print("apple" in fruit)
print("pear" in fruit)

print([1, 2] + [3, 4])          # you can ONLY add a list with another list
print(fruit + [6, 7, 8, 9])
print([0] * 4)                  # you can ONLY multiply a list with a positive integer
print([1, 2, ["hello", "goodbye"]] * 2)
```

List slices

- The slice operator we saw with strings also work on lists.
- What are the results of the following code?

```
list = ['a', 'b', 'c', 'd', 'e', 'f']
print(list[1:3])
print(list[:4])
print(list[3:])
print(list[:])
```

Lists are mutable

- Unlike strings, lists are **mutable** thus we can change the elements in a list. This can be done in a number of ways.
- Recall that strings are **immutable** thus we can't change the characters in a string.

```
fruit = ["banana", "apple", "cherry"]
print(fruit)
fruit[0] = "pear"       # Change one item using the index operator
fruit[-1] = "orange"
print(fruit)

list = ['a', 'b', 'c', 'd', 'e', 'f']
list[1:3] = ['x', 'y']     # Change two items using the slice operator
print(list)
```

Lists are mutable

- We can delete elements from a list by assigning the **empty list** to them.
- Or simply use the **del** statement.

```
list = ['a', 'b', 'c', 'd', 'e', 'f']
list[1:3] = []
print(list)

list = ['a', 'b', 'c', 'd', 'e', 'f']
del list[1:3]
del list[0]
print(list)
```

- We can even insert elements into a list by squeezing them into an **empty slice** at the desired location.

```
list = ['a', 'd', 'f']
list[1:1] = ['b', 'c']
print(list)
list[4:4] = ['e']
print(list)
```

Append vs concatenation

- The **append** method adds a new element to the end of a list.
- It is also possible to add a new element to the end of a list by using the + operator, i.e. **concatenation**.

```
orig_list = [45, 32, 88]
orig_list.append("cat")
print(orig_list)

orig_list = [45, 32, 88]
orig_list = orig_list + ["cat"]          # why do we need [] around “cat”?
print(orig_list)
```

List and loop

- We can traverse the elements in a list item-by-item using a loop. Thus the name “traverse by item”.

```
fruits = ["apple", "orange", "banana", "cherry"]

for fruit in fruits:                    # traversal by item
    print(fruit)
```

- We can also traverse the elements in a list by index:

```
numbers = [1, 2, 3, 4, 5]
print(numbers)

for i in range(len(numbers)):           # traversal by index
    numbers[i] = numbers[i] ** 2

print(numbers)
```

Exercise: Calculating the average of a list of integers

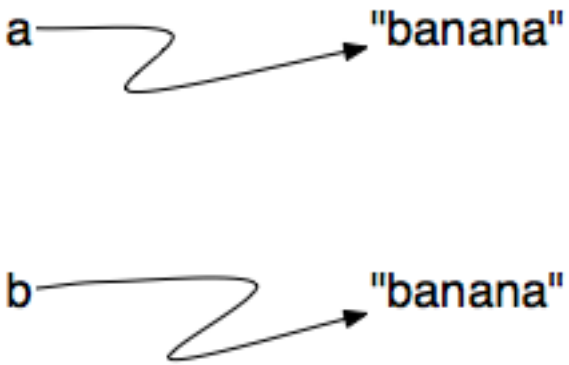
Write a function called **average** that will take a list of integers as a parameter and return the average value of that list.

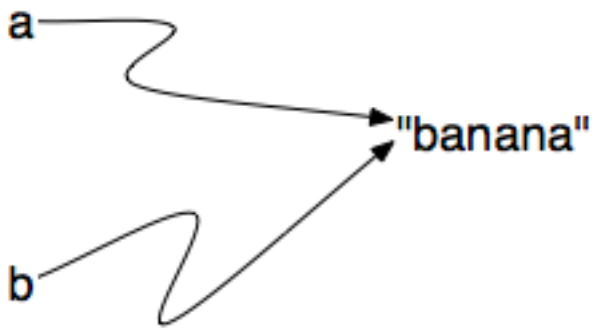
Objects and references

- Consider the following code:

```
a = "banana"
b = "banana"
print(a is b)          # True if a and b refer to the same object
```

- There are two possible ways the Python interpreter could arrange its internal states:



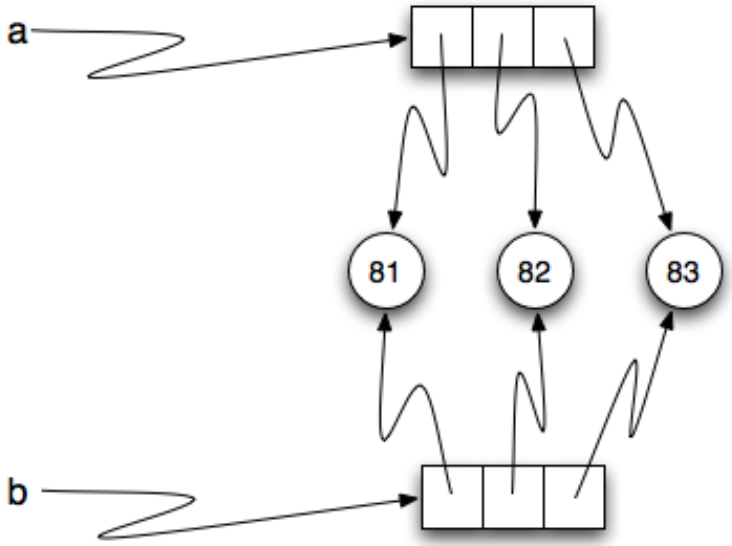


- Since strings are immutable, Python smartly selects the **second diagram** which uses less memory.

Objects and references

- The same thing is not true for lists as show in the following example:

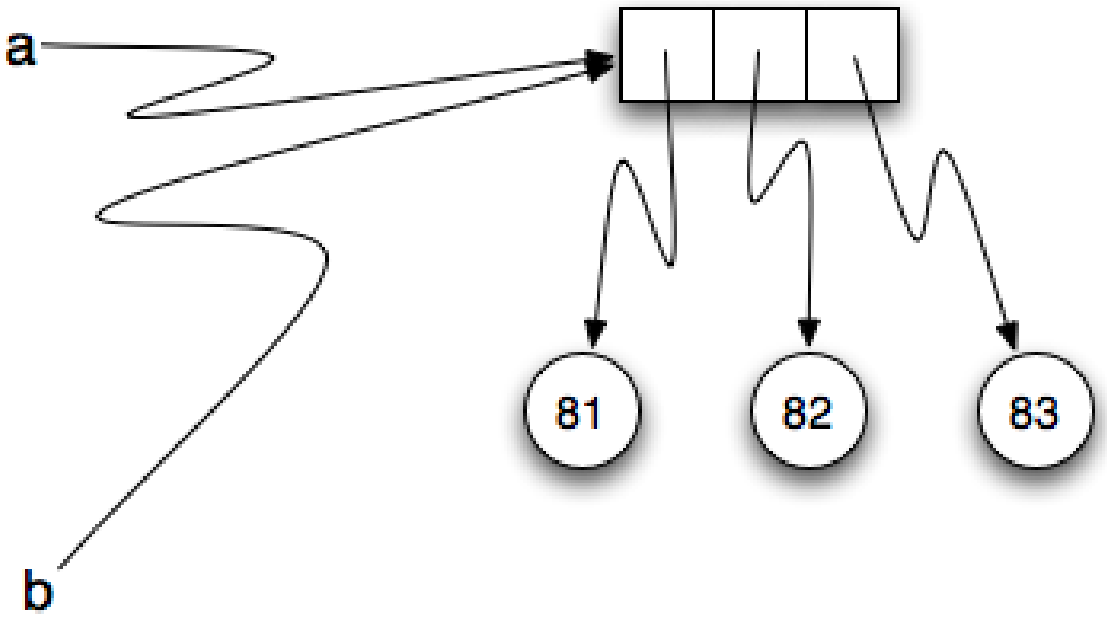
```
a = [81, 82, 83]
b = [81, 82, 83]

# print True if a and b refer to the same object
print(a is b)

# print True if each element in a is equal to the corresponding element in b
print(a == b)
```



List aliasing

- Since variables refer to objects, if we assign one variable to another, both variables refer to the same object. This is called **list aliasing**.

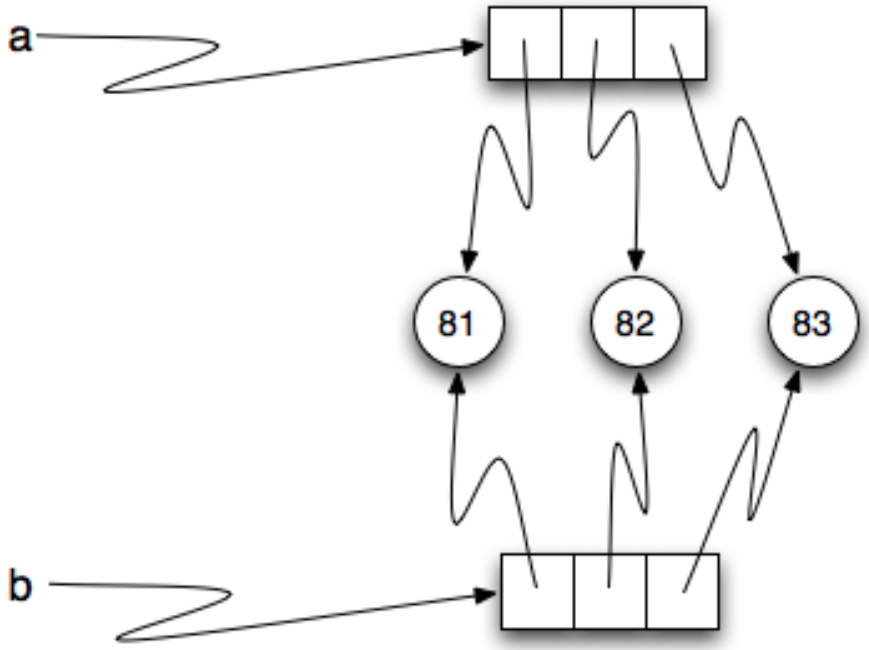
```
a = [81, 82, 83]
b = a
print(a is b)
```



List cloning

- If we want to modify a list and also keep a copy of the original, we need to be able to make a copy of the list itself, not just the reference.
- This is called **list cloning** (to avoid the ambiguity of the word copy).
- List cloning is achieved by using the slice operator.

```
a = [81, 82, 83]
b = a[:]                # list cloning
print(a == b)
print(a is b)

b[0] = 5
print(a)
print(b)
```

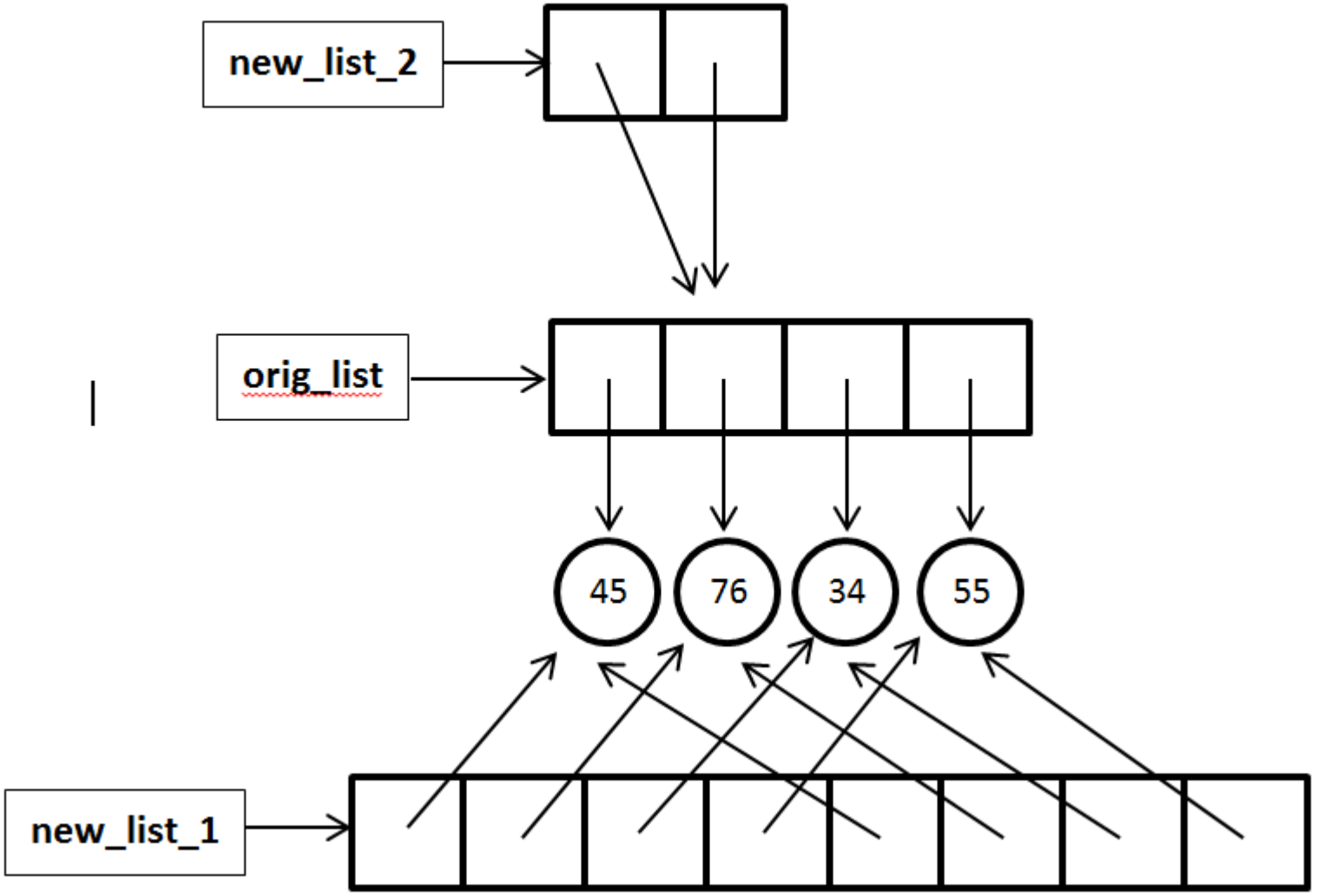


Repetition and references

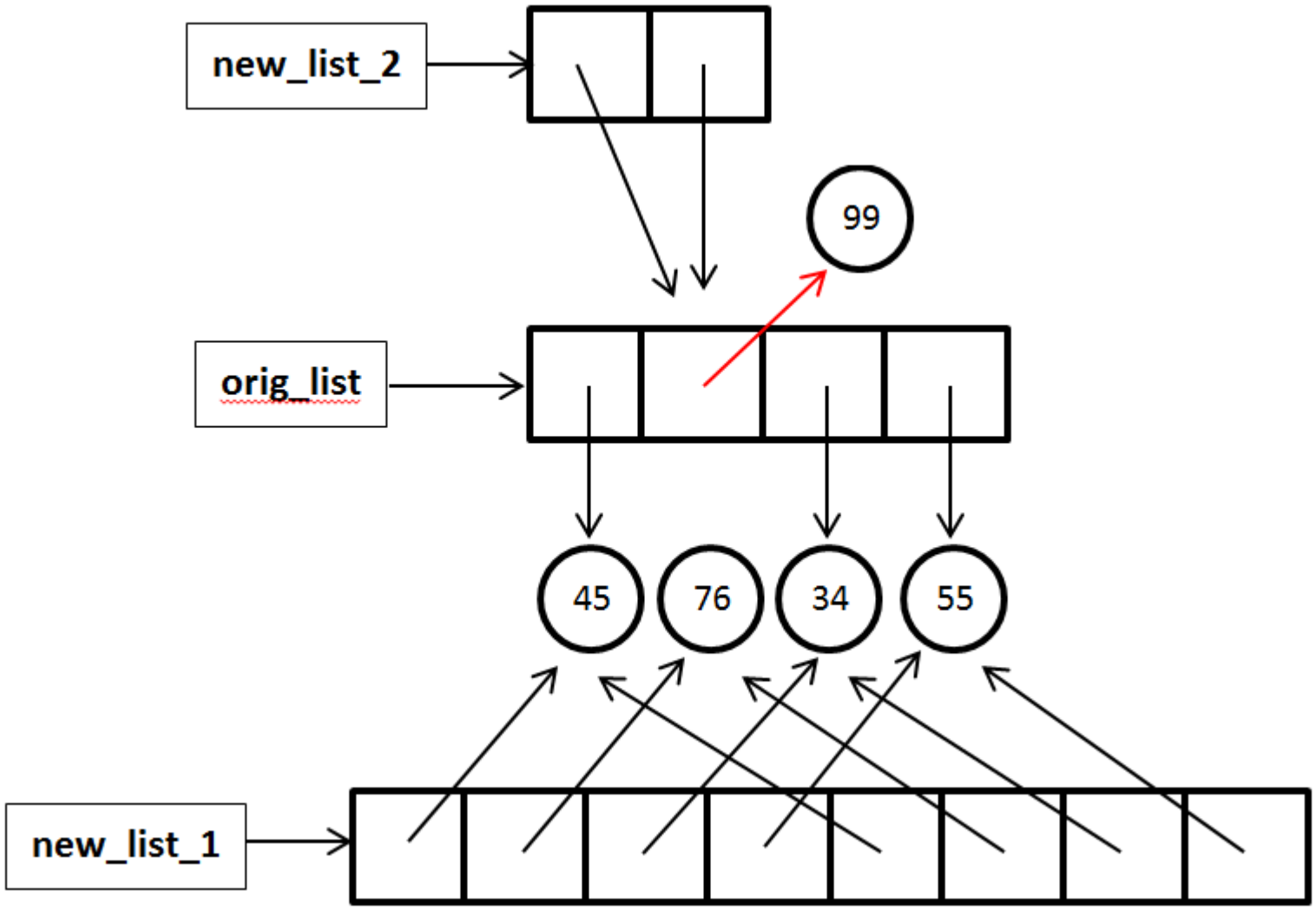
- With a list, the repetition operator (*) creates copies of the references. Although this may seem simple enough, when we allow a list to refer to another list, a subtle problem can arise.
- Consider the following example. What will happen to **new_list_1** and **new_list_2**?

```
# Before updating orig_list[1]
orig_list = [45, 76, 34, 55]          # orig_list[1] refers to 76
new_list_1 = orig_list * 2            # list of 8 elements, each refers to an integer
new_list_2 = [orig_list] * 2          # list of 2 elements, each refers to a list
print(new_list_1)
print(new_list_2)

# After updating orig_list[1]
orig_list[1] = 99                     # orig_list[1] now refers to 99
print(new_list_1)
print(new_list_2)
```

Repetition and references: Before updating orig_list[1]



Repetition and references: After updating orig_list[1]



List methods

- Like strings, lists also has methods. Most of them are easy to understand as in the following example:

```
my_list = []
my_list.append(5)
my_list.append(27)
my_list.append(3)
my_list.append(12)
print(my_list)

my_list.insert(1, 12)
print(my_list)
print(my_list.count(12))

print(my_list.index(3))
print(my_list.count(5))

my_list.reverse()
print(my_list)

my_list.sort()
print(my_list)

my_list.remove(5)
print(my_list)

last_item = my_list.pop()
print(last_item)
print(my_list)
```

*** Summary of list methods ***

Method	Parameters	Result	Description
append	item	mutator	Adds a new item to the end of a list
insert	index, item	mutator	Inserts a new item at the position given
pop	none	hybrid	Removes and returns the last item
pop	index	hybrid	Removes and returns the item at position
sort	none	mutator	Modifies a list to be sorted
reverse	none	mutator	Modifies a list to be in reverse order
index	item	idx	Returns the position of first occurrence of item
count	item	cnt	Returns the number of occurrences of item
remove	item	mutator	Removes the first occurrence of item

Using lists as parameters

- Functions which take lists as arguments and change them during execution are called **modifiers** and the changes they make are called **side effects**.
- Passing a list as an argument actually passes a reference to the list, not a copy of the list. This is **list aliasing**.
- Since lists are mutable, changes made to the elements referenced by the parameter change the same list that the argument is referencing.
- Here is an example of a modifier:

```
def double_stuff(list):
    """
    Overwrite each element in list with double its value.
    :param list: a list of elements
    :return: none
    """
    for item in list:
        item = item * 2

things = [2, 5, 9]
print(things)
double_stuff(things)
print(things)
```

Pure functions

- A **pure function** does not produce side effects. It communicates with the calling program only through parameters (which it does not modify) and a return value.
- As a result, **pure functions is preferred over modifiers**. We should only use modifiers whenever there is a compelling advantage.
- The modifier in the previous example is now re-written as a pure functions:

```
def double_stuff(list):
    """
    Return a new list in which contains doubles of the elements in the given list.
    :param list: a list of elements
    :return: none
    """
    new_list = []
    for item in list:
        new_elem = item * 2
        new_list.append(new_elem)
    return new_list

things = [2, 5, 9]
print(things)
new_things = double_stuff(things)
print(new_things)
```

Functions that produce lists

- Whenever you need to write a function that creates and returns a list, the pattern is usually:

```
initialize a result variable to be an empty list
loop
   create a new element
   append it to result
return the result
```

- Let us show another use of this pattern. Assume you already have a function **is_prime(i)** that can test if integer i is prime. Now, write a function to return a list of all primes less than a positive integer n:

```
def primes_upto(n):
    """
    Return a list of all primes less than n.
    :param n: a positive integer
    :return: a list of all primes less than n
    """
    result = []
    for i in range(2, n):
        if is_prime(i):
            result.append(i)
    return result
```

List comprehensions

- The previous example creates a list from a sequence of values based on some selection criteria. An easy way to do this type of processing in Python is to use a **list comprehension**.
- List comprehensions are concise ways to create lists. The general syntax is:

```
[<expression> for <element> in <sequence> if  <condition>]
```

- Using list comprehension, the previous **primes_upto(n)** function can be written as follow:

```
def primes_upto(n):
    '''
    Return a list of all primes less than n using a list comprehension.
    :param n: a positive integer
    :return: a list of primes less than n
    '''
    result = [num for num in range(2,n) if is_prime(num)]
    return result
```

- Isn't it cool???

Nested lists

- A nested list (or **sublist**) is a list that appears as an element in another list.
- To extract an element from the nested list, we can proceed in two steps. First, extract the nested list, then extract the item of interest.
- It is also possible to combine those steps using bracket operators that evaluate from left to right.

```
nested = ["hello", 2.0, 5, [10, 20]]
inner_list = nested[3]
print(inner_list)

elt = inner_list[1]
print(elt)

print(nested[3][1])
```

Strings and lists

- Two of the most useful functions on strings involve lists of strings. The **split** function breaks a string into a list of words. By default, any number of whitespace characters is considered a word boundary.

```
song = "The rain in Spain..."
words = song.split()
print(words)

song = "The rain in Spain..."
words = song.split('ai')
print(words)
```

- The inverse of the split function is **join**. You choose a desired **separator** string, (often called the *glue*) and join the list with the glue between each of the elements.

```
words = ["red", "blue", "green"]
glue = ';'
s = glue.join(words)
print(s
print("***".join(words))
print("".join(words))
```

List conversion function

- Python has a built-in type conversion function called **list** that tries to turn whatever you give it into a list.

```
xs = list("Crunchy Frog")
print(xs)
```

- It is also important to point out that the list conversion function will place each element of the original sequence in the new list.
- When working with strings, this is very different than the result of the split function.
- The **split** function breaks a string into a list of “words”, while the list function will always break it into a list of characters.

Exercise: Create and return a list

Write a function that creates and returns a list containing 100 random integers between 0 and 1000 (you can use random module, iteration, append, or list comprehension).