modifier A function which changes its arguments inside the function body. Only mutable types can be changed by modifiers.

mutable data value A data value which can be modified. The types of all mutable values are compound types. Lists and dictionaries are mutable; strings and tuples are not.

nested list A list that is an element of another list.

object A thing to which a variable can refer.

pattern A sequence of statements, or a style of coding something that has general applicability in a number of different situations. Part of becoming a mature Computer Scientist is to learn and establish the patterns and algorithms that form your toolkit. Patterns often correspond to your "mental chunking".

promise An object that promises to do some work or deliver some values if they're eventually needed, but it lazily puts off doing the work immediately. Calling `range` produces a promise.

pure function A function which has no side effects. Pure functions only make changes to the calling program through their return values.

sequence Any of the data types that consist of an ordered collection of elements, with each element identified by an index.

side effect A change in the state of a program made by calling a function. Side effects can only be produced by modifiers.

step size The interval between successive elements of a linear sequence. The third (and optional argument) to the `range` function is called the step size. If not specified, it defaults to 1.

11.22 Exercises

1. What is the Python interpreter's response to the following?

```
>>> list(range(10, 0, -2))
```

   The three arguments to the *range* function are *start*, *stop*, and *step*, respectively. In this example, `start` is greater than `stop`. What happens if `start < stop` and `step < 0`? Write a rule for the relationships among `start`, `stop`, and `step`.

2. Consider this fragment of code:

```
import turtle

tess = turtle.Turtle()
alex = tess
alex.color("hotpink")
```

   Does this fragment create one or two turtle instances? Does setting the color of `alex` also change the color of `tess`? Explain in detail.

3. Draw a state snapshot for `a` and `b` before and after the third line of the following Python code is executed:

```
a = [1, 2, 3]
b = a[:]
b[0] = 5
```

4. What will be the output of the following program?

```
this = ["I", "am", "not", "a", "crook"]
that = ["I", "am", "not", "a", "crook"]
print("Test 1: {0}".format(this is that))
that = this
print("Test 2: {0}".format(this is that))
```

   Provide a *detailed* explanation of the results.

5. Lists can be used to represent mathematical *vectors*. In this exercise and several that follow you will write functions to perform standard operations on vectors. Create a script named `vectors.py` and write Python code to pass the tests in each case.

   Write a function `add_vectors(u, v)` that takes two lists of numbers of the same length, and returns a new list containing the sums of the corresponding elements of each:

```
test(add_vectors([1, 1], [1, 1]) == [2, 2])
test(add_vectors([1, 2], [1, 4]) == [2, 6])
test(add_vectors([1, 2, 1], [1, 4, 3]) == [2, 6, 4])
```

6. Write a function `scalar_mult(s, v)` that takes a number, `s`, and a list, `v` and returns the scalar multiple of `v` by `s`. :

```
test(scalar_mult(5, [1, 2]) == [5, 10])
test(scalar_mult(3, [1, 0, -1]) == [3, 0, -3])
test(scalar_mult(7, [3, 0, 5, 11, 2]) == [21, 0, 35, 77, 14])
```

7. Write a function `dot_product(u, v)` that takes two lists of numbers of the same length, and returns the sum of the products of the corresponding elements of each (the dot_product).

```
test(dot_product([1, 1], [1, 1]) ==  2)
test(dot_product([1, 2], [1, 4]) ==  9)
test(dot_product([1, 2, 1], [1, 4, 3]) == 12)
```

8. *Extra challenge for the mathematically inclined*: Write a function `cross_product(u, v)` that takes two lists of numbers of length 3 and returns their cross product. You should write your own tests.

9. Describe the relationship between `" ".join(song.split())` and `song` in the fragment of code below. Are they the same for all strings assigned to `song`? When would they be different?

```
song = "The rain in Spain..."
```

10. Write a function `replace(s, old, new)` that replaces all occurrences of `old` with `new` in a string `s`:

```
test(replace("Mississippi", "i", "I") == "MIssIssIppI")

s = "I love spom! Spom is my favorite food. Spom, spom, yum!"
test(replace(s, "om", "am") ==
    "I love spam! Spam is my favorite food. Spam, spam, yum!")

test(replace(s, "o", "a") ==
    "I lave spam! Spam is my favarite faad. Spam, spam, yum!")
```

Hint: use the `split` and `join` methods.

11. Suppose you want to swap around the values in two variables. You decide to factor this out into a reusable function, and write this code:

```
def swap(x, y):       # Incorrect version
     print("before swap statement: x:", x, "y:", y)
     (x, y) = (y, x)
     print("after swap statement: x:", x, "y:", y)

a = ["This", "is", "fun"]
b = [2,3,4]
print("before swap function call: a:", a, "b:", b)
swap(a, b)
print("after swap function call: a:", a, "b:", b)
```

Run this program and describe the results. Oops! So it didn't do what you intended! Explain why not. Using a Python visualizer like the one at http://netserv.ict.ru.ac.za/python3_viz may help you build a good conceptual model of what is going on. What will be the values of `a` and `b` after the call to `swap`?

ch.11

October 12, 2020

[10]:
```
# Ch. 11: Lists
```

[11]:
```
# Ch. 11 Exercises
```

[12]:
```
# Ex. 1
```

[13]:
```
print(list(range(10, 0, -2)))
```

```
[10, 8, 6, 4, 2]
```

[14]:
```
# Ex. 2
```

[15]:
```
from unit_tester import test
import turtle

tess = turtle.Turtle()
alex = tess
alex.color("hotpink")

#this is aliacing. It creates one turtle instance. Yes setting the color of alex
#also changes color of tess
```

[16]:
```
# Ex. 3

a = [1, 2, 3]
b = a[:]
print(a is b)
print(a == b)
print(b)
b[0] = 5
print(a is b)
print(a == b)
print(b)
```

```
False
True
[1, 2, 3]
False
```

```
False
[5, 2, 3]
```

[17]:
```python
#Ex. 4


this = ["I", "am", "not", "a", "crook"]
that = ["I", "am", "not", "a", "crook"]

print("Test 1: {0}".format(this is that))
that = this   #here "that" and "this" both reffer to the same object
print("Test 2: {0}".format(this is that))
```

```
Test 1: False
Test 2: True
```

[18]:
```python
#Ex. 5

def add_vectors(u, v):
    list = []
    for i in range(len(u)):
        new_elem = u[i] + v[i]
        list.append(new_elem)
        print(list)
    return list

test(add_vectors([1, 1], [1, 1]) == [2, 2])
test(add_vectors([1, 2], [1, 4]) == [2, 6])
test(add_vectors([1, 2, 1], [1, 4, 3]) == [2, 6, 4])
```

```
[2]
[2, 2]
Test at line 11 ok.
[2]
[2, 6]
Test at line 12 ok.
[2]
[2, 6]
[2, 6, 4]
Test at line 13 ok.
```

[19]:
```python
#Ex. 6

def scalar_mult(s, v):
    list = []
    for i in range(len(v)):
        new_elem = s* v[i]
        list.append(new_elem)
```

```
        print(list)
    return list


test(scalar_mult(5, [1, 2]) == [5, 10])
test(scalar_mult(3, [1, 0, -1]) == [3, 0, -3])
test(scalar_mult(7, [3, 0, 5, 11, 2]) == [21, 0, 35, 77, 14])
```

```
[5]
[5, 10]
Test at line 12 ok.
[3]
[3, 0]
[3, 0, -3]
Test at line 13 ok.
[21]
[21, 0]
[21, 0, 35]
[21, 0, 35, 77]
[21, 0, 35, 77, 14]
Test at line 14 ok.
```

[20]:
```
#Ex. 7

def dot_product(u, v):
    list = []
    b = 0
    for i in range(len(v)):
        new_elem = u[i] * v[i]
        list.append(new_elem)
        print(list)
    for i in range(len(list)):
        b = b + list[i]
        print(b)
    return b


test(dot_product([1, 1], [1, 1]) == 2)
test(dot_product([1, 2], [1, 4]) == 9)
test(dot_product([1, 2, 1], [1, 4, 3]) == 12)
```

```
[1]
[1, 1]
1
2
Test at line 16 ok.
[1]
[1, 8]
```

```
1
9
Test at line 17 ok.
[1]
[1, 8]
[1, 8, 3]
1
9
12
Test at line 18 ok.
```

[21]:
```
#Ex. 8

#skipped
```

[22]:
```
#Ex. 9

song = "The rain in Spain..."

print(song.split())
print(" ".join(song.split())) # makes a split first, then uses the empty spaces␣
 ↪to join
print(song) # is the same as above
```

```
['The', 'rain', 'in', 'Spain...']
The rain in Spain...
The rain in Spain...
```

[23]:
```
#Ex. 10

def replace(s, old, new):
    new_elem = new.join(s.split(old))
    return new_elem

test(replace("Mississippi", "i", "I") == "MIssIssIppI")

s = "I love spom! Spom is my favorite food. Spom, spom, yum!"

test(replace(s, "om", "am") == "I love spam! Spam is my favorite food. Spam,␣
 ↪spam, yum!")

test(replace(s, "o", "a") == "I lave spam! Spam is my favarite faad. Spam, spam,␣
 ↪yum!")
```

```
Test at line 7 ok.
Test at line 11 ok.
Test at line 13 ok.
```

[24]:
```python
#Ex. 11

def swap(x, y): # Incorrect version
    print("before swap statement: x:", x, "y:", y)
    (x, y) = (y, x)
    print("after swap statement: x:", x, "y:", y)

a = ["This", "is", "fun"]
b = [2,3,4]
print("before swap function call: a:", a, "b:", b)
swap(a, b)
print("after swap function call: a:", a, "b:", b)

#the values of a and b didn't change
# a modifier is neccessery here to change the values of a and b
```

```
before swap function call: a: ['This', 'is', 'fun'] b: [2, 3, 4]
before swap statement: x: ['This', 'is', 'fun'] y: [2, 3, 4]
after swap statement: x: [2, 3, 4] y: ['This', 'is', 'fun']
after swap function call: a: ['This', 'is', 'fun'] b: [2, 3, 4]
```