Week 6: Python Worksheet 1

Barry Rowlingson

November 15, 2021

1 Quadratic Expression

Write a function, `quadratic`, that takes four arguments, `x,a,b,c`, and returns the value `y` of the quadratic defined by `a`, `b` and `c` at the value of `x`.

$$y = ax^2 + bx + c$$

```
def quadratic(x, a, b, c):
    """ evaluate a quadratic in x """
    return a*x**2 + b*x + c
```

Check your function reproduces these results:

```
results = [ quadratic(2, 0, 0, 0), # zero
            quadratic(7, 1, 0, 0), # seven-squared
            quadratic(3, 1, -1, 0), # 3-squared - 3
            quadratic(3, 1, -1, 10) ] # as previous but plus 10
print(results)
```

```
[0, 49, 6, 16]
```

2 Multiple x values

Modify the function so that it evaluates the quadratic for when `x` is passed in as a list. Use a *list comprehension* operation.

```
def quadratic(x, a, b , c):
   """ evaluate a quadratic in x """
   y = [a*z**2 + b*z + c for z in x]
   return y
```

Check the following:

```
x = range(-5,5)
print(quadratic(x, 1, -2, 3))
```

```
[38, 27, 18, 11, 6, 3, 2, 3, 6, 11]
```

3 Solving

The value of x that solves a quadratic equation is given by this formula:

$$x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

Start writing a Python function, `quadratic_solve`, with arguments for the parameters of a quadratic. Add a docstring, and for now, return the value `None`.

```
def quadratic_solve(a, b, c):
  """ Solve the quadratic defined by a, b, and c """
  return None
```

Load into Python and check that `quadratic_solve(1,1,1)` returns `None` and that calling it with too few or too many parameters returns an error. Check that `help(quadratic_solve)` shows the documentation string.

Next write a function to return the value of the determinant - the quantity inside the square root sign - and store in a variable called `det`.

```
def determinant(a, b, c):
  """ Solve the quadratic defined by a, b, and c """
  det = (b**2 - 4*a*c)
  return det
```

Check that it works with some simple tests.

```
D = [
 determinant(0,0,0), # zero times zero minus zero times zero times zero = zero
 determinant(0,3,0), # 3 squared minus zero zero zero = 9
 determinant(1,3,1), # 3 squared minus 4 = 5
 determinant(3,3,2) # 3**2 - 4*3*2 = -15
]
print(D)
```

```
[0, 9, 5, -15]
```

4 Solver function

Now complete the `quadratic_solve` function. First it should compute the determinant. If the determinant is negative then there are no real-value solutions and it should return an empty list. If the determinant is zero then it should return a list with just the one solution in. Otherwise it should return a list of the two solutions. To get the square root of a quantity, note that the square root of X is the same as X raised to the power 1/2.

```
def quadratic_solve(a, b, c):
  """ Solve a quadratic a^2+b+c=0 """
  det = determinant(a,b,c)
  if det < 0:
    return []
  elif det == 0:
    return [-b/(2*a)]
  else:
    return [(-b + det**(1/2))/(2*a),
            (-b - det**(1/2))/(2*a)]
```

```
print([
quadratic_solve(5,1,-3) # two real solutions
,
quadratic_solve(1,-4,4)  # one real solution
,
quadratic_solve(1,1,1)  # no real solutions
])
```

```
[[0.6810249675906654, -0.8810249675906654], [2.0], []]
```

5 Using The Standard Library

Instead of raising the determinant to the 1/2 power, we can use a square-root function from the standard library.

Import the `math` module, and replace your power-raising with a call to `math.sqrt`, checking that your answers are the same as before.

```
import math
def quadratic_solve(a, b, c):
  """ Solve a quadratic a^2+b+c=0 """
  det = determinant(a,b,c)
  if det < 0:
    return []
  elif det == 0:
    return [-b/(2*a)]
  else:
    sqd = math.sqrt(det)
    return [(-b + sqd)/(2*a),
            (-b - sqd)/(2*a)]
```

```
print([
quadratic_solve(5,1,-3) # two real solutions
,
quadratic_solve(1,-4,4)  # one real solution
,
quadratic_solve(1,1,1)  # no real solutions
])
```

```
[[0.6810249675906654, -0.8810249675906654], [2.0], []]
```

6 Data Structures

In the lectures I showed a dictionary structure for storing a student’s course data.

```
student = dict(
    name="Fred Smith",
    courses = [
        dict(name="CHIC402", mark=73),
        dict(name="CHIC602", mark=82)
        ]
    )
```

Write a function that creates and returns one of these structures with an empty `courses` element when given a name.

```
def new_student(name):
  d = dict(
    name = name,
    courses = []
  )
  return d
```

```
fred = new_student("Fred Smith")
print(fred)
```

```
’name’: ’Fred Smith’, ’courses’: []
```

Now write a function that adds an entry to a student record for a course, given the record object, a course name, and a grade.

```
def add_grade(s, course, grade):
  s[’courses’].append(dict(name=course, grade=grade))
```

```
add_grade(fred, course="CHIC999", grade=82)
add_grade(fred, course="CHIC123", grade=74)
print(fred)
```

```
’name’: ’Fred Smith’, ’courses’: [’name’: ’CHIC999’, ’grade’: 82,’name’: ’CHIC123’, ’grade’: 74]
```

What happens if you add a grade twice for the same course?

Next write a function that returns the average grade for a student’s courses. You can use the `sum` function to add the numeric values of a sequence, and the `len` function to get its length.

```
def average_grade(s):
  courses = s[’courses’]
  grades = [c[’grade’] for c in courses]
  ave = sum(grades)/len(grades)
  return ave
```

```
average_grade(fred)
```

```
78.0
```

Think about what other functions might be useful on this data structure. Don’t write the functions - just think about what might be needed in a simple student records application and maybe write just the `def` definition, a docstring, and return `None` for now. Think about possible error conditions that could happen with these functions.

```
def change_grade(s, course, grade):
  return None
def change_name(s, name):
  return None
def remove_course(s, course):
  return None
def compute_class(s):
  """ return I, IIi, IIii, III, P, F
  based on the algorithm..."""
  return None
```