Quadratic equations

Eric Martin, CSE, UNSW

COMP9021 Principles of Programming, trimester 1, 2019

```
In [1]: from math import sqrt
```

A quadratic equation is determined by three real numbers a, b and c with $a \neq 0$. Depending on whether $\Delta = \frac{-b}{2a}$ is strictly negative, equal to 0 or strictly positive, the equation has no real root, a single real root or two distinct real roots, respectively. We want to be able to:

- create a quadratic equation with specific values for a, b and c;
- modify a given quadratic equation by changing the value of any of a, b and c, possibly two of them, possibly all of them;
- nicely display a given quadratic equation;
- automatically compute the root or roots of a given quadratic equation, if a unique root or two distinct roots exist, respectively, when the equation is created, and whenever the equation is modified.

We will go through 5 successive designs via gradual modifications, the first design being satisfactory from a functional point of view, the last one satisfactory from an object oriented point of view. Going through this exercise will provide us with a deep understanding of object oriented design and syntax.

To create a quadratic equation, it is sensible to define a function that provides the default values of 1 for a and 0 for b and c (so making $x^2 = 1$ the default quadratic equation). Some or all of those default values can then be changed using keyword arguments, in any order:

```
In [2]: def f(a = 1, b = 0, c = 0):
            print(a, b, c)

        f()
        f(a = 2)
        f(c = 4)
        f(a = 2, c = 4)
        f(c = 4, b = 3)
        f(b = 3, a = 2, c = 4)
```

```
1 0 0
2 0 0
1 0 4
2 0 4
1 3 4
2 3 4
```

The user can also provide positional arguments, to overwrite the default value of the first argument, or to overwrite the default values of the first and second arguments, or to overwrite the values of the first, second and third arguments:

```
In [3]: f(2)
        f(2, 3)
        f(2, 3, 4)
```

```
2 0 0
2 3 0
2 3 4
```

In particular, when the user provides three positional arguments, we expect him to know that they are provided in the order a, b and c. Even if the function is well documented, it is reasonable not to have such an expectation, and force the user to explictly name each value. The $*$ symbol can be used on its own for that purpose. With the version of `f()` that follows, only the values of `a` and `b` can be overwritten with positional arguments:

```
In [4]: def f(a = 1, b = 0, *, c = 0):
            print(a, b, c)

        f(2)
        f(2, 3)
        f(2, 3, c = 4)
        f(2, c = 4, b = 3)
        f(b = 3, a = 2, c = 4)
        f(2, 3, 4)
```

```
2 0 0
2 3 0
2 3 4
2 3 4
2 3 4
```

```
    ---------------------------------------------------------------------------

    TypeError                                 Traceback (most recent call last)

    <ipython-input-4-fcbd77921fc2> in <module>()
      7 f(2, c = 4, b = 3)
      8 f(b = 3, a = 2, c = 4)
----> 9 f(2, 3, 4)


    TypeError: f() takes from 0 to 2 positional arguments but 3 were given
```

With the version of `f()` that follows, only the value of `a` can be overwritten with a positional argument:

```
In [5]: def f(a = 1, *, b = 0, c = 0):
            print(a, b, c)

        f(2)
        f(2, b = 3)
        f(2, c = 4, b = 3)
        f(b = 3, a = 2, c = 4)
        f(2, 3)

2 0 0
2 3 0
2 3 4
2 3 4



        ---------------------------------------------------------------------------

        TypeError                                 Traceback (most recent call last)

        <ipython-input-5-ea427c3b4254> in <module>()
          6 f(2, c = 4, b = 3)
          7 f(b = 3, a = 2, c = 4)
    ----> 8 f(2, 3)


        TypeError: f() takes from 0 to 1 positional arguments but 2 were given
```

With the version of `f()` that follows, no value can be overwritten with a positional argument:

```
In [6]: def f(*, a = 1, b = 0, c = 0):
            print(a, b, c)

        f(b = 3)
        f(c = 4, b = 3)
        f(b = 3, a = 2, c = 4)
        f(2)

1 3 0
1 3 4
2 3 4



        ---------------------------------------------------------------------------
```

```
TypeError                                 Traceback (most recent call last)

<ipython-input-6-33fed5a6edc5> in <module>()
  5 f(c = 4, b = 3)
  6 f(b = 3, a = 2, c = 4)
----> 7 f(2)


TypeError: f() takes 0 positional arguments but 1 was given
```

The function `initialise()` below is meant to create a quadratic equation represented as a dictionary with 5 keys, to keep track of the values of a, b and c, but also of the values of the roots, computed by the function `compute_roots()` as soon as the former are known. We might think that `create()` would be a better name; we will soon understand why we opted for `initialise()`:

```
In [7]: def initialise(*, a = 1, b = 0, c = 0):
            if a == 0:
                print('a cannot be equal to 0.')
                return
            equation = {'a': a, 'b': b, 'c': c, 'root_1': None, 'root_2': None}
            compute_roots(equation)
            return equation

        def compute_roots(equation):
            a, b, c = equation['a'], equation['b'], equation['c']
            delta = b ** 2 - 4 * a * c
            if delta < 0:
                equation['root_1'] = equation['root_2'] = None
            elif delta == 0:
                equation['root_1'] = -b / (2 * a)
                equation['root_2'] = None
            else:
                sqrt_delta = sqrt(delta)
                equation['root_1'] = (-b - sqrt_delta) / (2 * a)
                equation['root_2'] = (-b + sqrt_delta) / (2 * a)

In [8]: initialise(a = 0, b = 1)

a cannot be equal to 0.


In [9]: eq1 = initialise()

        eq1['a']
        eq1['b']
        eq1['c']
        eq1['root_1']
        eq1['root_2'] # None
```

```
Out[9]: 1

Out[9]: 0

Out[9]: 0

Out[9]: 0.0
```

```
In [10]: eq2 = initialise(b = 4)

         eq2['a']
         eq2['b']
         eq2['c']
         eq2['root_1']
         eq2['root_2']
```

```
Out[10]: 1

Out[10]: 4

Out[10]: 0

Out[10]: -4.0

Out[10]: 0.0
```

The function that follows allows one to change any number of parameters, that again have to be named even in case the change affects all of them; `compute_roots()` recomputes the roots as soon as the possibly new values of a, b and c are known:

```
In [11]: def update(equation, *, a = None, b = None, c = None):
             if a == 0:
                 print('a cannot be equal to 0.')
                 return
             if a is not None:
                 equation['a'] = a
             if b is not None:
                 equation['b'] = b
             if c is not None:
                 equation['c'] = c
             compute_roots(equation)
```

```
In [12]: eq3 = initialise(a = 1, b = 3, c = 2)
         eq3['a']
         eq3['b']
         eq3['c']
         eq3['root_1']
         eq3['root_2']

         print()
```

```python
update(eq3, a = 0)
print()

update(eq3, b = -1)
eq3['a']
eq3['b']
eq3['c']
eq3['root_1'] # None
eq3['root_2'] # None

print()

update(eq3, c = 0.3, a = 0.5)
eq3['a']
eq3['b']
eq3['c']
eq3['root_1']
eq3['root_2']
```

```
Out[12]: 1

Out[12]: 3

Out[12]: 2

Out[12]: -2.0

Out[12]: -1.0


a cannot be equal to 0.



Out[12]: 1

Out[12]: -1

Out[12]: 2




Out[12]: 0.5

Out[12]: -1

Out[12]: 0.3

Out[12]: 0.3675444679663241
```

```
Out[12]: 1.632455532033676
```

To nicely display an equation, we have to carefully deal with the cases where a, b or c are equal to 1 or -1 and where b or c are equal to 0, strictly positive or strictly negative:

```
In [13]: def display(equation):
             a, b, c = equation['a'], equation['b'], equation['c']
             if a == 1:
                 displayed_equation = 'x^2'
             elif a == -1:
                 displayed_equation = '-x^2'
             else:
                 displayed_equation = f'{a}x^2'
             if b == 1:
                 displayed_equation += ' + x'
             elif b == -1:
                 displayed_equation -= ' - x'
             elif b > 0:
                 displayed_equation += f' + {b}x'
             elif b < 0:
                 displayed_equation += f'- {-b}x'
             if c > 0:
                 displayed_equation += f' + {c}'
             elif c < 0:
                 displayed_equation += f' - {-c}'
             print(displayed_equation, 0, sep = ' = ')
```

```
In [14]: display(initialise())
         display(initialise(c = -5, a = 2))
         display(initialise(b = 1, a = -1, c = -1))

x^2 = 0
2x^2 - 5 = 0
-x^2 + x - 1 = 0
```

That ends the first design. For the second design, we package the functionality associated with quadratic equations; a dictionary offers a simple way to do so. The dictionary `QuadraticEquationDict` below captures the view that a quadratic equation is an entity that can be created (initialised), displayed, modified (updated) and has roots that can be computed. All of the dictionary's values are functions; they have been previously defined, but two of them are given a slightly different implementation, reflecting the fact that the four functions are now part of the `QuadraticEquationDict` "package":

```
In [15]: def initialise_variant_1(*, a = 1, b = 0, c = 0):
             if a == 0:
                 print('a cannot be equal to 0.')
                 return
             equation = {'a': a, 'b': b, 'c': c, 'root_1': None, 'root_2': None}
             QuadraticEquationDict['compute_roots'](equation)
```

```
    return equation

def update_variant_1(equation, *, a = None, b = None, c = None):
    if a == 0:
        print('a cannot be equal to 0.')
        return
    if a is not None:
        equation['a'] = a
    if b is not None:
        equation['b'] = b
    if c is not None:
        equation['c'] = c
    QuadraticEquationDict['compute_roots'](equation)

QuadraticEquationDict = {'initialise': initialise_variant_1,
                         'display': display,
                         'compute_roots': compute_roots,
                         'update': update_variant_1
                        }
```

The code that tests all four functions is similarly changed and relative to the `QuadraticEquationDict` "package":

```
In [16]: QuadraticEquationDict['initialise'](a = 0, b = 1)

a cannot be equal to 0.


In [17]: eq1 = QuadraticEquationDict['initialise']()

         eq1['a']
         eq1['b']
         eq1['c']
         eq1['root_1']
         eq1['root_2'] # None

Out[17]: 1

Out[17]: 0

Out[17]: 0

Out[17]: 0.0

In [18]: eq2 = QuadraticEquationDict['initialise'](b = 4)

         eq2['a']
         eq2['b']
         eq2['c']
         eq2['root_1']
         eq2['root_2']
```

```
Out[18]: 1

Out[18]: 4

Out[18]: 0

Out[18]: -4.0

Out[18]: 0.0
```

```
In [19]: eq3 = QuadraticEquationDict['initialise'](a = 1, b = 3, c = 2)
         eq3['a']
         eq3['b']
         eq3['c']
         eq3['root_1']
         eq3['root_2']

         print()
         QuadraticEquationDict['update'](eq3, a = 0)
         print()

         update(eq3, b = -1)
         eq3['a']
         eq3['b']
         eq3['c']
         eq3['root_1'] # None
         eq3['root_2'] # None

         print()

         QuadraticEquationDict['update'](eq3, c = 0.3, a = 0.5)
         eq3['a']
         eq3['b']
         eq3['c']
         eq3['root_1']
         eq3['root_2']
```

```
Out[19]: 1

Out[19]: 3

Out[19]: 2

Out[19]: -2.0

Out[19]: -1.0
```

```
a cannot be equal to 0.
```

```
Out[19]: 1

Out[19]: -1

Out[19]: 2



Out[19]: 0.5

Out[19]: -1

Out[19]: 0.3

Out[19]: 0.3675444679663241

Out[19]: 1.632455532033676
```

```
In [20]: QuadraticEquationDict['display'](QuadraticEquationDict['initialise']())
         QuadraticEquationDict['display'](QuadraticEquationDict['initialise'](c = -5,
                                                                               a = 2
                                                                              )
                                         )
         QuadraticEquationDict['display'](QuadraticEquationDict['initialise'](b = 1,
                                                                               a = -1,
                                                                               c = -1
                                                                              )
                                         )
```

```
x^2 = 0
2x^2 - 5 = 0
-x^2 + x - 1 = 0
```

With the third design, we meet the object oriented paradigm. All four functions are implemented slightly differently:

```
In [21]: def initialise_variant_2(equation, *, a = 1, b = 0, c = 0):
             if a == 0:
                 print('a cannot be equal to 0.')
                 return
             equation.a = a
             equation.b = b
             equation.c = c
             QuadraticEquationType.compute_roots(equation)

         def display_variant_1(equation):
             a, b, c = equation.a, equation.b, equation.c
             if a == 1:
```

```
        displayed_equation = 'x^2'
    elif a == -1:
        displayed_equation = '-x^2'
    else:
        displayed_equation = f'{a}x^2'
    if b == 1:
        displayed_equation += ' + x'
    elif b == -1:
        displayed_equation -= ' - x'
    elif b > 0:
        displayed_equation += f' + {b}x'
    elif b < 0:
        displayed_equation += f'- {-b}x'
    if c > 0:
        displayed_equation += f' + {c}'
    elif c < 0:
        displayed_equation += f' - {-c}'
    print(displayed_equation, 0, sep = ' = ')

def compute_roots_variant_2(equation):
    a, b, c = equation.a, equation.b, equation.c
    delta = b ** 2 - 4 * a * c
    if delta < 0:
        equation.root_1 = equation.root_2 = None
    elif delta == 0:
        equation.root_1 = -b / (2 * a)
        equation.root_2 = None
    else:
        sqrt_delta = sqrt(delta)
        equation.root_1 = (-b - sqrt_delta) / (2 * a)
        equation.root_2 = (-b + sqrt_delta) / (2 * a)

def update_variant_2(equation, *, a = None, b = None, c = None):
    if a == 0:
        print('a cannot be equal to 0.')
        return
    if a is not None:
        equation.a = a
    if b is not None:
        equation.b = b
    if c is not None:
        equation.c = c
    QuadraticEquationType.compute_roots(equation)

QuadraticEquationType = type('QuadraticEquationType',
                             (),
                             {'__init__' : initialise_variant_2,
                              'display': display_variant_1,
```

```
                                    'compute_roots': compute_roots_variant_2,
                                    'update': update_variant_2
                                   }
                                  )
```

`QuadraticEquationType` seems to embed `QuadraticEquationDict`, with `'initialise'` changed to `'__init__'`; `'initialise'` was an arbitrary name, whereas `'__init__'` is imposed. With `'initialise'`, we chose a name close enough to `'__init__'` so as to reflect the similarity of the implementations. `QuadraticEquationType` is a **type**, with the name `'QuadraticEquationType'`, above provided as first argument to `type()`; another use of `type()` below shows that `QuadraticEquationType` is indeed a type:

```
In [22]: type(QuadraticEquationType)
         QuadraticEquationType.__name__
```

```
Out[22]: type
```

```
Out[22]: 'QuadraticEquationType'
```

The second argument to `type()` is an empty tuple, making `QuadraticEquationType` a direct subtype of `object`, the mother of all types:

```
In [23]: QuadraticEquationType.__base__
```

```
Out[23]: object
```

The third argument to `type()` is a dictionary of **attributes**, that are all members of `__dict__`, which itself is another attribute of `QuadraticEquationType`:

```
In [24]: QuadraticEquationType.__dict__
```

```
Out[24]: mappingproxy({'__init__': <function __main__.initialise_variant_2(...
                                      ...equation, *, a=1, b=0, c=0)>,
                       'display': <function __main__.display_variant_1(equation)>,
                       'compute_roots': <function __main__.compute_roots_variant_2(...
                                                                      ...equation)>,
                       'update': <function __main__.update_variant_2(...
                                ...equation, *, a=None, b=None, c=None)>,
                       '__module__': '__main__',
                       '__dict__': <attribute '__dict__' of...
                                           ...'QuadraticEquationType' objects>,
                       '__weakref__': <attribute '__weakref__'...
                                          ...of 'QuadraticEquationType' objects>,
                       '__doc__': None})
```

`object` also has a `__dict__` attribute:

```
In [25]: object.__dict__
```

```
Out[25]: mappingproxy({'__repr__': <slot wrapper '__repr__' of 'object' objects>,
                       '__hash__': <slot wrapper '__hash__' of 'object' objects>,
                       '__str__': <slot wrapper '__str__' of 'object' objects>,
                       '__getattribute__': <slot wrapper '__getattribute__' of...
                                                                  ...'object' objects>,
                       '__setattr__': <slot wrapper '__setattr__' of 'object' objects>,
                       '__delattr__': <slot wrapper '__delattr__' of 'object' objects>,
                       '__lt__': <slot wrapper '__lt__' of 'object' objects>,
                       '__le__': <slot wrapper '__le__' of 'object' objects>,
                       '__eq__': <slot wrapper '__eq__' of 'object' objects>,
                       '__ne__': <slot wrapper '__ne__' of 'object' objects>,
                       '__gt__': <slot wrapper '__gt__' of 'object' objects>,
                       '__ge__': <slot wrapper '__ge__' of 'object' objects>,
                       '__init__': <slot wrapper '__init__' of 'object' objects>,
                       '__new__': <function object.__new__(*args, **kwargs)>,
                       '__reduce_ex__': <method '__reduce_ex__' of 'object' objects>,
                       '__reduce__': <method '__reduce__' of 'object' objects>,
                       '__subclasshook__': <method '__subclasshook__' of...
                                                                  ...'object' objects>,
                       '__init_subclass__': <method '__init_subclass__' of...
                                                                  ...'object' objects>,
                       '__format__': <method '__format__' of 'object' objects>,
                       '__sizeof__': <method '__sizeof__' of 'object' objects>,
                       '__dir__': <method '__dir__' of 'object' objects>,
                       '__class__': <attribute '__class__' of 'object' objects>,
                       '__doc__': 'The most base type'})
```

The `dir()` function returns a list of attributes of its argument; with `object` as argument, that list consists of nothing but the attributes in `object.__dict__`:

```
In [26]: dir(object)

         set(object.__dict__) == set(dir(object))
```

```
Out[26]: ['__class__',
          '__delattr__',
          '__dir__',
          '__doc__',
          '__eq__',
          '__format__',
          '__ge__',
          '__getattribute__',
          '__gt__',
          '__hash__',
          '__init__',
          '__init_subclass__',
          '__le__',
          '__lt__',
```

```
 '__ne__',
 '__new__',
 '__reduce__',
 '__reduce_ex__',
 '__repr__',
 '__setattr__',
 '__sizeof__',
 '__str__',
 '__subclasshook__']
```

```
Out[26]: True
```

With `QuadraticEquationType` as argument, `dir()` returns a list of attributes that consists precisely of the attributes in `object.__dict__` (or equivalently, the attributes in `dir(object)`), all **inherited** by `QuadraticEquationType`, and the attributes in `QuadraticEquationType.__dict__`:

```
In [27]: dir(QuadraticEquationType)

         set(dir(QuadraticEquationType)) == set(object.__dict__) |\
                                            set(QuadraticEquationType.__dict__)
```

```
Out[27]: ['__class__',
 '__delattr__',
 '__dict__',
 '__dir__',
 '__doc__',
 '__eq__',
 '__format__',
 '__ge__',
 '__getattribute__',
 '__gt__',
 '__hash__',
 '__init__',
 '__init_subclass__',
 '__le__',
 '__lt__',
 '__module__',
 '__ne__',
 '__new__',
 '__reduce__',
 '__reduce_ex__',
 '__repr__',
 '__setattr__',
 '__sizeof__',
 '__str__',
 '__subclasshook__',
 '__weakref__',
 'compute_roots',
```

```
         'display',
         'update']
```

```
Out[27]: True
```

Two attributes belong to both `object.__dict__` and `QuadraticEquationType.__dict__`; they are attributes of `object` inherited by `QuadraticEquationType`, but also **overwritten** by `QuadraticEquationType`:

```
In [28]: set(dir(object)) & set(QuadraticEquationType.__dict__)
         object.__doc__
         QuadraticEquationType.__doc__ # None
         object.__init__
         QuadraticEquationType.__init__
```

```
Out[28]: {'__doc__', '__init__'}
```

```
Out[28]: 'The most base type'
```

```
Out[28]: <slot wrapper '__init__' of 'object' objects>
```

```
Out[28]: <function __main__.initialise_variant_2(equation, *, a=1, b=0, c=0)>
```

With the syntax `QuadraticEquationType.compute_roots`, we are trying to access the `'compute_roots'` attribute of `QuadraticEquationType`, which is equivalent to retrieving the value of the `'compute_roots'` key of the `'__dict__'` attribute of `QuadraticEquationType` (this raises the question of how `'__dict__'` itself is retrieved...):

```
In [29]: QuadraticEquationType.compute_roots
         QuadraticEquationType.__dict__['compute_roots']
```

```
Out[29]: <function __main__.compute_roots_variant_2(equation)>
```

```
Out[29]: <function __main__.compute_roots_variant_2(equation)>
```

The key difference between `initialise_variant_1()` and `initialise_variant_2()` is that the latter has an extra argument, `equation`, and returns `None`, whereas the former returns a dictionary that is the counterpart to `equation`. What value is assigned to `equation`; what provides it? One of `QuadraticEquationType`'s attributes is `'__new__'`, which we know is inherited from and not overwritten by `object`:

```
In [30]: QuadraticEquationType.__new__
         QuadraticEquationType.__new__ is object.__new__
```

```
Out[30]: <function object.__new__(*args, **kwargs)>
```

```
Out[30]: True
```

When called, the function that `QuadraticEquationType.__new__`, or equivalently, `object.__new__`, evaluates to, returns an object (not to be confused with `object`) of type `QuadraticEquationType`:

```
In [31]: QuadraticEquationType.__new__(QuadraticEquationType)
         type(QuadraticEquationType.__new__(QuadraticEquationType))
```

```
Out[31]: <__main__.QuadraticEquationType at 0x10d9f8668>
```

```
Out[31]: __main__.QuadraticEquationType
```

The object returned by `QuadraticEquationType.__new__(QuadraticEquationType)` can be then passed as an argument to `initialise_variant_2()`, the function that `QuadraticEquationType.__init__` evaluates to:

```
In [32]: eq = QuadraticEquationType.__new__(QuadraticEquationType)

         QuadraticEquationType.__init__(eq, a = 0, b = 1)
```

```
a cannot be equal to 0.
```

The object returned by `QuadraticEquationType.__new__(QuadraticEquationType)` has a `__dict__` attribute, that happens to be empty; the `dir()` function returns the same list of attributes when it is given either the object or `QuadraticEquationType` as argument, reflecting the fact that the object inherits all those attributes from `QuadraticEquationType`:

```
In [33]: eq1 = QuadraticEquationType.__new__(QuadraticEquationType)

         eq1.__dict__
         dir(eq1) == dir(QuadraticEquationType)
```

```
Out[33]: {}
```

```
Out[33]: True
```

A call to `QuadraticEquationType.__init__()` provides `eq1` with new attributes, which are now in `eq1.__dict__` and also part of `dir(eq_1)`. With the syntax `eq1.a`, `eq1.b`, `eq1.c`, `eq1.root_1` and `eq1.root_2`, we are trying to access the `'a'`, `'b'`, `'c'`, `'root_1'` and `'root_2'` attributes of `eq1`, which is equivalent to retrieving the values of the `'a'`, `'b'`, `'c'`, `'root_1'` and `'root_2'` keys of the `'__dict__'` attribute of `eq1` (this again raises the question of how `'__dict__'` itself is retrieved):

```
In [34]: QuadraticEquationType.__init__(eq1)

         eq1.__dict__
         set(dir(eq1)) - set(dir(QuadraticEquationType))
         eq1.__dict__['a']
         eq1.a
         eq1.__dict__['b']
         eq1.b
         eq1.__dict__['c']
         eq1.c
         eq1.__dict__['root_1']
         eq1.root_1
         eq1.__dict__['root_2'] # None
         eq1.root_2 # None
```

```
Out[34]: {'a': 1, 'b': 0, 'c': 0, 'root_1': 0.0, 'root_2': None}

Out[34]: {'a', 'b', 'c', 'root_1', 'root_2'}

Out[34]: 1

Out[34]: 1

Out[34]: 0

Out[34]: 0

Out[34]: 0

Out[34]: 0

Out[34]: 0.0

Out[34]: 0.0
```

We now understand what value `initialise_variant_2()`'s first argument, `equation`, receives, and what provides it, but in practice, we do not explicitly call first `QuadraticEquationType.__new__()` and then `QuadraticEquationType.__init__()`; instead, we use the following syntax, that in one sweep move, both creates an object and initialises it:

```
In [35]: eq2 = QuadraticEquationType(b = 4)

         eq2.a
         eq2.b
         eq2.c
         eq2.root_1
         eq2.root_2

Out[35]: 1

Out[35]: 4

Out[35]: 0

Out[35]: -4.0

Out[35]: 0.0
```

`compute_roots`, `update` and `display` are attributes of both `QuadraticEquationType` and of objects returned by `QuadraticEquationType.__new__(QuadraticEquationType)`, but they evaluate to different entities:

```
In [36]: QuadraticEquationType.compute_roots
         QuadraticEquationType.__new__(QuadraticEquationType).compute_roots

         print()

         QuadraticEquationType.update
         QuadraticEquationType.__new__(QuadraticEquationType).update

         print()

         QuadraticEquationType.display
         QuadraticEquationType.__new__(QuadraticEquationType).display
```

```
Out[36]: <function __main__.compute_roots_variant_2(equation)>

Out[36]: <bound method compute_roots_variant_2 of ...
                             ...<__main__.QuadraticEquationType object at 0x10da0e400>>




Out[36]: <function __main__.update_variant_2(equation, *, a=None, b=None, c=None)>

Out[36]: <bound method update_variant_2 of ...
                             ...<__main__.QuadraticEquationType object at 0x10da0e898>>




Out[36]: <function __main__.display_variant_1(equation)>

Out[36]: <bound method display_variant_1 of ...
                             ...<__main__.QuadraticEquationType object at 0x10da0e518>>
```

These **bound methods** essentially allow one to call `compute_roots_variant_2()`, `update_variant_2()` and `display_variant_1()` using `'compute_roots'`, `'update'` and `'display'` as object attributes rather than `QuadraticEquationType` attributes, providing the desired value as first argument. More precisely, one can think of the bound method M of an object o of type `QuadraticEquationType` as a pair:

- the first member of the pair is a `QuadraticEquationType` function f, meant to take an object of type `QuadraticEquationType` as first (and possibly unique) argument;
- the second member of the pair is o, meant to be that first argument.

Having both f and o in hand together with any other arguments for f, if any, f can then be called with o provided as first argument. This can done either as `QuadraticEquationType.variable_referring_to_f(variable_referring_to_o, possibly followed by extra arguments)`, or as `variable_referring_to_o.variable_referring_to_f(possibly, extra arguments)`. This alternative syntax is more compact and the one used in practice:

In [37]:
```
eq3 = QuadraticEquationType(a = 1, b = 3, c = 2)
eq3.a
eq3.b
eq3.c
eq3.root_1
eq3.root_2

print()
# update() called as a QuadraticEquationType function
QuadraticEquationType.update(eq3, a = 0)
print()

# update() called as a QuadraticEquationType function
QuadraticEquationType.update(eq3, b = -1)
eq3.a
eq3.b
eq3.c
eq3.root_1 # None
eq3.root_2 # None

print()

# update() called as an eq3 bound method
eq3.update(c = 0.3, a = 0.5)
eq3.a
eq3.b
eq3.c
eq3.root_1
eq3.root_2
```

Out[37]: 1

Out[37]: 3

Out[37]: 2

Out[37]: -2.0

Out[37]: -1.0

```
a cannot be equal to 0.
```

Out[37]: 1

Out[37]: -1

Out[37]: 2

```
Out[37]: 0.5

Out[37]: -1

Out[37]: 0.3

Out[37]: 0.3675444679663241

Out[37]: 1.632455532033676
```

```
In [38]: # display() called as a QuadraticEquationType function
         QuadraticEquationType.display(QuadraticEquationType())
         # display() called as a QuadraticEquationType function
         QuadraticEquationType.display(QuadraticEquationType(c = -5, a = 2))
         # display() called as a bound method of the object
         # returned by QuadraticEquationType()
         QuadraticEquationType(c = -5, a = 2).display()
```

```
x^2 = 0
2x^2 - 5 = 0
2x^2 - 5 = 0
```

The fourth design is essentially nothing but a syntactic variant on the third design, with `class` followed by the first argument to `type()` (that we change to `QuadraticEquationClass`), and with the functions that are the values of the dictionary provided as third argument to `type()` now in the body of the class statement. Also, `display` is renamed `__str__`, and whereas the former returns `None` and executes `print()` statements, the latter returns a string: when `print()` is given an object as argument, it calls the object's `__str__()` bound method and displays the returned string:

```
In [39]: class QuadraticEquationClass:
             def __init__(equation, *, a = 1, b = 0, c = 0):
                 if a == 0:
                     print('a cannot be equal to 0.')
                     return
                 equation.a = a
                 equation.b = b
                 equation.c = c
                 equation.compute_roots()

             def __str__(equation):
                 if equation.a == 1:
                     displayed_equation = 'x^2'
                 elif equation.a == -1:
                     displayed_equation = '-x^2'
                 else:
```

```
            displayed_equation = f'{equation.a}x^2'
        if equation.b == 1:
            displayed_equation += ' + x'
        elif equation.b == -1:
            displayed_equation -= ' - x'
        elif equation.b > 0:
            displayed_equation += f' + {equation.b}x'
        elif equation.b < 0:
            displayed_equation += f'- {-equation.b}x'
        if equation.c > 0:
            displayed_equation += f' + {equation.c}'
        elif equation.c < 0:
            displayed_equation += f' - {-equation.c}'
        return f'{displayed_equation} = 0'

    def compute_roots(equation):
        delta = equation.b ** 2 - 4 * equation.a * equation.c
        if delta < 0:
            equation.root_1 = equation.root_2 = None
        elif delta == 0:
            equation.root_1 = -equation.b / (2 * equation.a)
            equation.root_2 = None
        else:
            sqrt_delta = sqrt(delta)
            equation.root_1 = (-equation.b - sqrt_delta) / (2 * equation.a)
            equation.root_2 = (-equation.b + sqrt_delta) / (2 * equation.a)

    def update(equation, *, a = None, b = None, c = None):
        if a == 0:
            print('a cannot be equal to 0.')
            return
        if a is not None:
            equation.a = a
        if b is not None:
            equation.b = b
        if c is not None:
            equation.c = c
        equation.compute_roots()
```

The syntax for object creation and initialisation and for calls to `compute_roots()` and `update()` is the same as with the third design:

```
In [40]: eq1 = QuadraticEquationClass.__new__(QuadraticEquationClass)
         QuadraticEquationClass.__init__(eq1)
         eq1.a
         eq1.b
         eq1.c
         eq1.root_1
```

```
eq1.root_2

print()

eq2 = QuadraticEquationClass.__new__(QuadraticEquationClass)
eq2.__init__(b = 4)
eq2.a
eq2.b
eq2.c
eq2.root_1
eq2.root_2

print()

eq3 = QuadraticEquationClass(a = 1, b = 3, c = 2)
eq3.a
eq3.b
eq3.c
eq3.root_1
eq3.root_2

print()
eq3.update(a = 0)
print()

QuadraticEquationClass.update(eq3, b = -1)
eq3.a
eq3.b
eq3.c
eq3.root_1
eq3.root_2

print()

eq3.update(c = 0.3, a = 0.5)
eq3.a
eq3.b
eq3.c
eq3.root_1
eq3.root_2
```

```
a cannot be equal to 0.
```

```
Out[49]: <__main__.QuadraticEquationClass at 0x107a13320>
```

```
Out[49]: 1

Out[49]: 0

Out[49]: 0

Out[49]: 0.0


Out[49]: 1

Out[49]: 4

Out[49]: 0

Out[49]: -4.0

Out[49]: 0.0


Out[49]: 1

Out[49]: 3

Out[49]: 2

Out[49]: -2.0

Out[49]: -1.0

a cannot be equal to 0.


Out[49]: 1

Out[49]: -1

Out[49]: 2


Out[49]: 0.5

Out[49]: -1
```

```
Out[49]: 0.3

Out[49]: 0.3675444679663241

Out[49]: 1.632455532033676
```

As previously mentioned, we now display quadratic equations not with calls to `display()`, but with calls directly to `print()` that behind the scene, calls `__str()__`:

```
In [41]: print(QuadraticEquationClass())
         print(QuadraticEquationClass(c = -5, a = 2))
         print(QuadraticEquationClass(b = 1, a = -1, c = -1))
```

```
x^2 = 0
2x^2 - 5 = 0
-x^2 + x - 1 = 0
```

The fifth design "cleans" the fourth design, changing the first argument of the bound methods to `self` as always done in practice. Another special bound method, `__repr()__`, is overwritten: similarly to `__str()__`, it returns a string, and it is called when the executed statement is just the object name. It has a default implementation, but the output is not particularly insightful:

```
In [42]: eq3
```

```
Out[42]: <__main__.QuadraticEquationClass at 0x10da0e6a0>
```

It is standard practice to let `__repr()__` output the very statement that creates the object, so for `eq3`, `QuadraticEquationClass(a = 1, b = 3, c = 2)`, as we will see below.

Finally, note that with the fourth design, `QuadraticEquationClass(a = 0, b = 1)` prints out an error message but still returns an ill defined object of type `QuadraticEquationClass`. It is preferable to raise an error instead. We define a specific exception by defining a class that derives from `Exception` rather than from `object`:

```
In [43]: class QuadraticEquationError(Exception):
             def __init__(self, message):
                 self.message = message

         QuadraticEquationError.__base__
         raise QuadraticEquationError('a cannot be equal to 0')
```

```
Out[43]: Exception
```

```
        ---------------------------------------------------------------------------

        QuadraticEquationError                    Traceback (most recent call last)

        <ipython-input-43-83301e9fd64e> in <module>()
          4
```

```
      5 QuadraticEquationError.__base__
----> 6 raise QuadraticEquationError('a cannot be equal to 0')


    QuadraticEquationError: a cannot be equal to 0
```

Putting things together, here is the final implementation, that abides by the principles of object oriented design in Python:

```
In [44]: class QuadraticEquation:
             def __init__(self, *, a = 1, b = 0, c = 0):
                 if a == 0:
                     raise QuadraticEquationError('a cannot be equal to 0.')
                 self.a = a
                 self.b = b
                 self.c = c
                 self.compute_roots()

             def __repr__(self):
                 return f'QuadraticEquation(a = {self.a}, b = {self.b}, c = {self.c})'

             def __str__(self):
                 if self.a == 1:
                     displayed_equation = 'x^2'
                 elif self.a == -1:
                     displayed_equation = '-x^2'
                 else:
                     displayed_equation = f'{self.a}x^2'
                 if self.b == 1:
                     displayed_equation += ' + x'
                 elif self.b == -1:
                     displayed_equation -= ' - x'
                 elif self.b > 0:
                     displayed_equation += f' + {self.b}x'
                 elif self.b < 0:
                     displayed_equation += f'- {-self.b}x'
                 if self.c > 0:
                     displayed_equation += f' + {self.c}'
                 elif self.c < 0:
                     displayed_equation += f' - {-self.c}'
                 return f'{displayed_equation} = 0'

             def compute_roots(self):
                 delta = self.b ** 2 - 4 * self.a * self.c
                 if delta < 0:
                     self.root_1 = self.root_2 = None
                 elif delta == 0:
```

```
                self.root_1 = -self.b / (2 * self.a)
                self.root_2 = None
            else:
                sqrt_delta = sqrt(delta)
                self.root_1 = (-self.b - sqrt_delta) / (2 * self.a)
                self.root_2 = (-self.b + sqrt_delta) / (2 * self.a)

        def update(self, *, a = None, b = None, c = None):
            if a == 0:
                raise QuadraticEquationError('a cannot be equal to 0.')
            if a is not None:
                self.a = a
            if b is not None:
                self.b = b
            if c is not None:
                self.c = c
            self.compute_roots()
```

An error of type `QuadraticEquationError` is raised at object creation, or when incorrectly updating an existing object:

```
In [45]: QuadraticEquation(a = 0, b = 1)


        ---------------------------------------------------------------------------


        QuadraticEquationError                    Traceback (most recent call last)

        <ipython-input-45-e27558a7e912> in <module>()
    ----> 1 QuadraticEquation(a = 0, b = 1)


        <ipython-input-44-3eaf5abdbb98> in __init__(self, a, b, c)
          2     def __init__(self, *, a = 1, b = 0, c = 0):
          3         if a == 0:
    ----> 4             raise QuadraticEquationError('a cannot be equal to 0.')
          5         self.a = a
          6         self.b = b


        QuadraticEquationError: a cannot be equal to 0.


In [46]: eq3 = QuadraticEquation(a = 1, b = 3, c = 2)
         eq3.update(a = 0)


        ---------------------------------------------------------------------------
```

```
QuadraticEquationError                    Traceback (most recent call last)

<ipython-input-46-7601bbbd1bf3> in <module>()
  1 eq3 = QuadraticEquation(a = 1, b = 3, c = 2)
----> 2 eq3.update(a = 0)


<ipython-input-44-3eaf5abdbb98> in update(self, a, b, c)
 46     def update(self, *, a = None, b = None, c = None):
 47         if a == 0:
---> 48             raise QuadraticEquationError('a cannot be equal to 0.')
 49         if a is not None:
 50             self.a = a


QuadraticEquationError: a cannot be equal to 0.
```

Otherwise, there is no difference with the 4th design when it comes to creating objects or calling methods:

```
In [47]: eq1 = QuadraticEquation.__new__(QuadraticEquation)
         QuadraticEquation.__init__(eq1)
         eq1.a
         eq1.b
         eq1.c
         eq1.root_1
         eq1.root_2

         print()

         eq2 = QuadraticEquation.__new__(QuadraticEquation)
         eq2.__init__(b = 4)
         eq2.a
         eq2.b
         eq2.c
         eq2.root_1
         eq2.root_2

         print()

         eq3 = QuadraticEquation(a = 1, b = 3, c = 2)
         eq3.a
         eq3.b
         eq3.c
         eq3.root_1
         eq3.root_2
```

```
QuadraticEquation.update(eq3, b = -1)
eq3.a
eq3.b
eq3.c
eq3.root_1
eq3.root_2

print()

eq3.update(c = 0.3, a = 0.5)
eq3.a
eq3.b
eq3.c
eq3.root_1
eq3.root_2
```

```
Out[47]: 1

Out[47]: 0

Out[47]: 0

Out[47]: 0.0



Out[47]: 1

Out[47]: 4

Out[47]: 0

Out[47]: -4.0

Out[47]: 0.0



Out[47]: 1

Out[47]: 3

Out[47]: 2

Out[47]: -2.0

Out[47]: -1.0

Out[47]: 1
```

```
Out[47]: -1
```

```
Out[47]: 2
```

```
Out[47]: 0.5
```

```
Out[47]: -1
```

```
Out[47]: 0.3
```

```
Out[47]: 0.3675444679663241
```

```
Out[47]: 1.632455532033676
```

Observe the difference between calling either `__repr__()` or `__str__()` behind the scene:

```
In [48]: eq1 = QuadraticEquation()
         eq1
         print(eq1)

         eq2 = QuadraticEquation(c = -5, a = 2)
         eq2
         print(eq2)

         eq3 = QuadraticEquation(b = 1, a = -1, c = -1)
         eq3
         print(eq3)
```

```
Out[48]: QuadraticEquation(a = 1, b = 0, c = 0)
```

```
x^2 = 0
```

```
Out[48]: QuadraticEquation(a = 2, b = 0, c = -5)
```

```
2x^2 - 5 = 0
```

```
Out[48]: QuadraticEquation(a = -1, b = 1, c = -1)
```

```
-x^2 + x - 1 = 0
```