Lesson 1: Loans and savings

Created: 2024-05-27T01:25:55.331163+10:00

Loans and savings

Loans and savings

loans_and_savings.py

Loans and savings

Eric Martin, CSE, UNSW

COMP9021 Principles of Programming

[1]:
```
from math import log10
from numbers import Real
```

1 Borrowing or saving money

We consider the situation where Jane has an initial sum of S_0 in her account, and at regular intervals, adds to that account a positive or negative amount (so in effect, adds to or removes money from that account), say Δ, with an interest of r applying to the time period between two successive operations. For i a strictly positive integer, let S_i denote the sum in Jane's account after i time periods. Then we have for all $i \in \mathbf{N}$:

$$S_i = S_0(1+r)^i + \Delta\Sigma_{j=0}^{i-1}(1+r)^j \tag{1}$$

as we immediately verify by induction: (1) trivially holds for $i = 0$, and given $i \in \mathbf{N}$, if (1) holds for i then

$$\begin{aligned} S_{i+1} &= S_i(1+r) + \Delta \\ &= \big(S_0(1+r)^i + \Delta\Sigma_{j=0}^{i-1}(1+r)^j\big)(1+r) + \Delta \\ &= S_0(1+r)^{i+1} + \Delta\Sigma_{j=1}^{i}(1+r)^j + \Delta \\ &= S_0(1+r)^{i+1} + \Delta\Sigma_{j=0}^{i}(1+r)^j \end{aligned}$$

1.1 Saving

In the case of an investment, then Δ is equal to S_0 and operations happen once a year. Assuming that after N years, Jane can close her account and decides to do so, then she would not add the last amount of Δ, so the final amount, the sum eventually available to her, say S, would be:

$$S = S_N - \Delta = \Delta(1+r)^N + \Delta\Sigma_{j=0}^{N-1}(1+r)^j - \Delta = \Delta\Sigma_{j=1}^{N}(1+r)^j = \frac{\Delta}{r}\big((1+r)^{N+1} - (1+r)\big)$$

We therefore have the following equations:

- $S = \frac{\Delta}{r}\big((1+r)^{N+1} - (1+r)\big)$
- $\Delta = \dfrac{Sr}{(1+r)^{N+1} - (1+r)}$

- $N = \dfrac{\log_{10}\Big(\big(\frac{Sr}{\Delta} + (1+r)\big)\Big)}{\log_{10}(1+r)} - 1$

1.2 Borrowing

In the case of a loan over N years, the operations happen once a month, and the final sum eventually becomes 0:

$$0 = S_{12N} = S_0(1+r)^{12N} + \Delta\Sigma_{j=0}^{12(N-1)}(1+r)^j = S_0(1+r)^{12N} + \frac{\Delta}{r}\big((1+r)^{12N} - 1\big)$$

We therefore have the following equations:

- $S_0 = -\dfrac{\Delta\big((1+r)^{12N} - 1\big)}{r(1+r)^{12N}}$
- $\Delta = -\dfrac{S_0(1+r)^{12N} r}{(1+r)^{12N} - 1}$
- $N = \dfrac{\log_{10}(\frac{\Delta}{rS_0+\Delta})}{12\log_{10}(1+r)}$

1.3 Effective interest rate

Let R be the annual interest rate. First a period, year, semester, quarter or month, is chosen. Let d be 1, 2, 4 or 12, respectively (note that 1, 2, 4 and 12 are the number of years, semesters, quarters and months in a year, respectively). The interest rate is first reduced to R/d and declared to be the interest for the chosen period, that is, the interest that has to be paid at the end of that period. This corresponds to an effective interest rate for the year, say $\widetilde{R}$, equal to $(1 + R/d)^d - 1$, which corresponds to an effective interest rate for the month equal to the number $\widehat{R}$ such that $(1+\widehat{R})^{12} = 1 + \widetilde{R}$, yielding $\widehat{R} = (1+\widetilde{R})^{\frac{1}{12}} - 1$.

- For savings, $r = \widetilde{R}$, which is all the more advantageous to Jane that d is larger.
- For loans, $r = \widehat{R}$, which is all the more advantageous to Jane that d is smaller.

2 Design and implementation

Let us create three classes, `Account`, `Loan` and `Savings`, the last two being subclasses of the first one since a loan account and a savings account are particular kinds of accounts. We define a few strings as class attributes to, at any time, let objects created from those classes keep track of what is known and what is unknown. More precisely:

- The interest will have to be fixed to a stricly positive floating point number when the object is created.
- The reference period, year, semester, quarter or month, will have to be fixed when the object is created too.
- The term amount, that is, the sum of money that is credited every year if the account is a savings account and that is debited every month if the account is a loan account, can be set or not to some value at any time, including when the object is created.

- The duration, that is, the number of years the account will remain open, and whose balance has to be equal to 0 at closure time in case the account is a loan account, can be set or not to some value at any time, including when the object is created.
- The initial sum only makes sense for a loan account; it is the sum of money that is borrowed when the account is opened. It can be set or not to some value at any time, including when the object is created.
- The final sum only makes sense for a savings account; it is the sum of money that is available when the account is closed. It can be set or not to some value at any time, including when the object is created.

In any circumstance, including when the object is created, at least one of the parameters whose value can not be set should not be set; in case there is only one such parameter, then that parameter will be computed from all others.

We also define as a class attribute a dictionary for the number of months that make up each of the possible periods:

[2]:
```
class Account:
    INTEREST = 'interest'
    REFERENCE_PERIOD = 'reference_period'
    TERM_AMOUNT = 'term_amount'
    DURATION = 'duration'
    INITIAL_SUM = 'initial_sum'
    FINAL_SUM = 'final_sum'
    nb_in_year = {'year' : 1, 'semester' : 2, 'quarter' : 4, 'month' : 12}

class Savings(Account):
    pass

class Loan(Account):
    pass
```

If we create an object `account` of type `Account`, then `account.INTEREST` is first looked for as an object attribute and not found, then looked for as an `Account` attribute and found. If we create an object `loan` of type `Loan`, then `loan.INTEREST` is first looked for as an object attribute and not found, then looked for as a `Loan` attribute and not found, then looked for as an `Account` attribute and found. Later, we will see methods of the `Loan` and `Savings` classes that refer to `Account.INTEREST` rather than to `self.INTEREST`; that way, they will find `'interest'` directly in the parent class rather than indirectly, exploring from the object to the own class and from the own class to the parent class:

[3]:
```
account = Account()
loan = Loan()

Account.INTEREST
account.INTEREST
loan.INTEREST
```

[3]:
```
'interest'
```

[3]: 'interest'

[3]: 'interest'

Let us create a particular kind of exception, to raise for the case where all of the parameters that can optionally be set are provided some nonzero value, leaving no unknown:

[4]:
```
class NoUnknownException(Exception):
    pass
```

Durations are expected to be integers, while interests, term amounts, initial sums and final sums are expected to be either floating point numbers or integers. To check the latter, the `Real` class from the `numbers` module is useful:

[5]:
```
isinstance(1, int), isinstance(1, float), isinstance(1, Real)
isinstance(1., int), isinstance(1., float), isinstance(1., Real)
```

[5]: (True, False, True)

[5]: (False, True, True)

Let us define an `Account` method, `check_type()`, to check whether the value of an attribute of an object of type `Account` (which could therefore be more precisely of type `Loan` or `Savings`), is of the expected type:

[6]:
```
class Account(Account):
    def check_type(self, parameter, parameter_name, valid_type):
        if not isinstance(parameter, valid_type):
            raise TypeError(f'{parameter_name} should be of type {valid_type}')

class Savings(Account):
    pass

class Loan(Account):
    pass
```

[7]:
```
loan = Loan()
loan._interest = 0.08
# Returns None
loan.check_type(loan._interest, Account.INTEREST, Real)
loan._interest = '0.08'
# Raises a TypeError exception
loan.check_type(loan._interest, Account.INTEREST, Real)
```

␣
↪---

```
TypeError                                 Traceback (most recent call␣
↪last)

<ipython-input-7-b4632d7025f8> in <module>
  5 loan._interest = '0.08'
  6 # Raises a TypeError exception
----> 7 loan.check_type(loan._interest, Account.INTEREST, Real)


<ipython-input-6-21e8cb36fb31> in check_type(self, parameter,␣
↪parameter_name, valid_type)
  2     def check_type(self, parameter, parameter_name, valid_type):
  3         if not isinstance(parameter, valid_type):
----> 4             raise TypeError(f'{parameter_name} should be of type␣
↪{valid_type}')
  5
  6 class Savings(Account):


TypeError: interest should be of type <class 'numbers.Real'>
```

The builtin `setattr()` function allows one to set an object's attribute to some value via a variable name (a string) rather than via a variable. This will prove useful to let a method set the value of some attribute, determined as a string, only known at run time:

[8]:
```
account = Account()
account._interest = 0.08
setattr(account, '_duration', 20)
account.__dict__
```

[8]:
```
{'_interest': 0.08, '_duration': 20}
```

Let us define an `Account` method, `set_parameter()`, to set an attribute of an object of type `Account` (which could therefore be more precisely of type `Loan` or `Savings`) to some value, which in case it is not equal to 0, is imposed to be positive, except for the term amount of a `Loan` object that should be negative (to be debited from the account). In case the attribute has been set to a nonzero value, then the method will remove its name from the set of unknown parameters, making sure that the set does not become empty; otherwise, a `NoUnknownException` will be raised:

[9]:
```
class Account(Account):
    def set_parameter(self, parameter, parameter_name, sign):
        if parameter * sign < 0:
            raise ValueError(f'{parameter_name} should be of opposite sign')
        if parameter:
            if parameter_name in self._unknowns:
                if len(self._unknowns) == 1:
                    raise NoUnknownException(f'{parameter_name} is the only '
```

```
                                            'unknown parameter'
                                        )
                self._unknowns.remove(parameter_name)
        else:
            self._unknowns.add(parameter_name)
        setattr(self, '_' + parameter_name, parameter)

class Savings(Account):
    pass

class Loan(Account):
    pass
```

[10]:
```
loan = Loan()
loan._unknowns = {Account.INITIAL_SUM, Account.TERM_AMOUNT, Account.DURATION}
loan.set_parameter(836.44, Account.TERM_AMOUNT, -1)
```

```
      ␣
↪---------------------------------------------------------------------------

       ValueError                                Traceback (most recent call␣
↪last)

       <ipython-input-10-0ec805017f7d> in <module>
         1 loan = Loan()
         2 loan._unknowns = {Account.INITIAL_SUM, Account.TERM_AMOUNT, Account.
↪DURATION}
   ----> 3 loan.set_parameter(836.44, Account.TERM_AMOUNT, -1)


       <ipython-input-9-b00ff9272185> in set_parameter(self, parameter,␣
↪parameter_name, sign)
         2     def set_parameter(self, parameter, parameter_name, sign):
         3         if parameter * sign < 0:
   ----> 4             raise ValueError(f'{parameter_name} should be of␣
↪opposite sign')
         5         if parameter:
         6             if parameter_name in self._unknowns:


       ValueError: term_amount should be of opposite sign
```

[11]:
```
loan = Loan()
loan._unknowns = {Account.INITIAL_SUM, Account.TERM_AMOUNT, Account.DURATION}
loan.set_parameter(-836.44, Account.TERM_AMOUNT, -1)
```

```
loan.set_parameter(20, Account.DURATION, 1)

loan.__dict__
```

[11]:
```
{'_unknowns': {'initial_sum'}, '_term_amount': -836.44, '_duration': 20}
```

[12]:
```
loan = Loan()
loan._unknowns = {Account.INITIAL_SUM, Account.TERM_AMOUNT, Account.DURATION}
loan.set_parameter(-836.44, Account.TERM_AMOUNT, -1)
loan.set_parameter(20, Account.DURATION, 1)
loan.set_parameter(100000, Account.INITIAL_SUM, 1)
```

```
␣
↪---------------------------------------------------------------------------

NoUnknownException                        Traceback (most recent call␣
↪last)

<ipython-input-12-7d927333becc> in <module>
  3 loan.set_parameter(-836.44, Account.TERM_AMOUNT, -1)
  4 loan.set_parameter(20, Account.DURATION, 1)
----> 5 loan.set_parameter(100000, Account.INITIAL_SUM, 1)


<ipython-input-9-b00ff9272185> in set_parameter(self, parameter,␣
↪parameter_name, sign)
  6             if parameter_name in self._unknowns:
  7                 if len(self._unknowns) == 1:
----> 8                     raise NoUnknownException(f'{parameter_name} is␣
↪the only '
  9                                              'unknown parameter'
 10                                             )


NoUnknownException: initial_sum is the only unknown parameter
```

Let us bundle the `check_type()` and `set_parameter()` methods of the `Account` class into a `check_and_set_parameter()` method:

[13]:
```
class Account(Account):
    def check_and_set_parameter(self, parameter, parameter_name, valid_types,
                                sign
                               ):
        self.check_type(parameter, parameter_name, valid_types)
        self.set_parameter(parameter, parameter_name, sign)
```

```
class Savings(Account):
    pass

class Loan(Account):
    pass
```

[14]:
```
savings = Savings()
savings._unknowns = {Account.TERM_AMOUNT, Account.FINAL_SUM, Account.DURATION}
savings.check_and_set_parameter(1000, Account.TERM_AMOUNT, Real, 1)
savings.check_and_set_parameter(1000, Account.DURATION, int, 1)

savings.__dict__
```

[14]:
```
{'_unknowns': {'final_sum'}, '_term_amount': 1000, '_duration': 1000}
```

Let us add to `Account` two methods, `set_interest()` and `set_reference_period()`, to fix the values of the interest and the reference period, respectively. As those have to be set at object creation and should not be modified afterwards, it is not appropriate to use `set_parameter()`. We can still make use of `check_type()` to check that the interest is a strictly positive real value; checking that the reference value is one of `'year'`, `'semester'`, `'quarter'` or `'month'` can be done directly.

Let us also add to `Account` a method, `set_effective_interest()`, to compute the effective interest from the interest and the reference period.

Finally, let us add to `Acccount` two methods, `set_term_amount()` and `set_duration()`, to set or change or remove (by assigning 0) the values of the term amount and the duration, respectively. Given an object of type `Savings`, say `savings`, a call such as `savings.set_term_amount(1000)` would look for `set_term_amount()` first unsuccessfully as an attribute of `savings`, then unsuccessfully as an attribute of `Savings`, and finally successfully as an attribute of `Account`. In the body of `set_term_amount()`, the statement `isinstance(self, Savings)`, with `self` referring to `savings` would then evaluate to `True` (and so `2 * isinstance(self, Savings) - 1` would evaluate to 1, whereas it would evaluate to -1 if `self` was referring to an object of type `Loan`):

[15]:
```
class Account(Account):
    def set_interest(self, interest):
        self.check_type(interest, Account.INTEREST, Real)
        if interest == 0:
            raise ValueError('Interest should not be 0')
        if interest < 0:
            raise ValueError('Interest should not be negative')
        self._interest = interest

    def set_reference_period(self, reference_period):
        if reference_period not in Account.nb_in_year:
            raise ValueError('Reference period should be one of '
                             f'{list(Account.nb_in_year)}'
```

```
                              )
        self._reference_period = reference_period

    def set_effective_interest(self):
        self.effective_interest =\
              ((1 + self._interest / Account.nb_in_year[self._reference_period]
               ) ** Account.nb_in_year[self._reference_period] - 1
              )
    def set_term_amount(self, term_amount):
        # An amount added to Savings account, and deducted from a Loans
        # account.
        self.check_and_set_parameter(term_amount, Account.TERM_AMOUNT, Real,
                                     2 * isinstance(self, Savings) - 1
                                    )

    def set_duration(self, duration):
        self.check_and_set_parameter(duration, Account.DURATION, int, 1)

class Savings(Account):
    def set_final_sum(self, final_sum):
        self.check_and_set_parameter(final_sum, Account.FINAL_SUM, Real, 1)

class Loan(Account):
    def set_initial_sum(self, initial_sum):
        self.check_and_set_parameter(initial_sum, Account.INITIAL_SUM, Real, 1)
```

[16]:
```
savings = Savings()
savings._unknowns = {Account.TERM_AMOUNT, Account.FINAL_SUM, Account.DURATION}
savings.set_interest(0.08)
savings.set_reference_period('year')
savings.set_duration(25)
savings.set_effective_interest()
savings.set_final_sum(78954.42)

savings.__dict__
```

[16]:
```
{'_unknowns': {'term_amount'},
 '_interest': 0.08,
 '_reference_period': 'year',
 '_duration': 25,
 'effective_interest': 0.08000000000000007,
 '_final_sum': 78954.42}
```

[17]:
```
loan = Loan()
loan._unknowns = {Account.INITIAL_SUM, Account.TERM_AMOUNT, Account.DURATION}
loan.set_interest(0.08)
loan.set_reference_period('year')
```

```
loan.set_duration(25)
loan.set_effective_interest()
loan.set_initial_sum(100000)

loan.__dict__
```

[17]:
```
{'_unknowns': {'term_amount'},
 '_interest': 0.08,
 '_reference_period': 'year',
 '_duration': 25,
 'effective_interest': 0.08000000000000007,
 '_initial_sum': 100000}
```

We do not want the user to change the value of the interest, but it is wishful thinking:

[18]:
```
loan._interest = 0.25
loan.__dict__
```

[18]:
```
{'_unknowns': {'term_amount'},
 '_interest': 0.25,
 '_reference_period': 'year',
 '_duration': 25,
 'effective_interest': 0.08000000000000007,
 '_initial_sum': 100000}
```

We can't prevent the user to change the value of `_interest`. But we can define a new variable, `interest`, and use the `@property` decorator to let the user access its value (the property is defined to just retrieve the value of `_interest` and present it to the user as the value of `interest`), without the user being able to modify that value via the name `interest`. So `interest` is part of the public interface while `_interest` is not. If they have access to the source code then users can see the name `_interest`, understand its purpose, not restrict themselves to the public interface, and modify the value of `_interest`. But without looking at the implementation, users are not aware of the existence of `_interest`; they can only see and understand the purpose of `interest`, access its value when needed, but never change it. The same can be done with the `_reference_period` attribute, letting the `@property` decorator introduce a read only attribute `reference_period`:

[19]:
```
class Account(Account):
    @property
    def interest(self):
        return self._interest

    @property
    def reference_period(self):
        return self._reference_period

class Savings(Account, Savings):
    pass
```

```
class Loan(Account, Loan):
    pass
```

[20]:
```
savings = Savings()
savings.set_interest(0.08)

savings.interest
savings.interest = 0.25
```

[20]:
```
0.08
```

```
␣
↪---------------------------------------------------------------------------

AttributeError                            Traceback (most recent call␣
↪last)

<ipython-input-20-0a84e2b6e130> in <module>
  3
  4 savings.interest
----> 5 savings.interest = 0.25


AttributeError: can't set attribute
```

[21]:
```
loan = Loan()
loan.set_reference_period('month')

loan.reference_period
loan.reference_period = 'quarter'
```

[21]:
```
'month'
```

```
␣
↪---------------------------------------------------------------------------

AttributeError                            Traceback (most recent call␣
↪last)

<ipython-input-21-3cef6f2c99a7> in <module>
  3
  4 loan.reference_period
----> 5 loan.reference_period = 'quarter'
```

```
AttributeError: can't set attribute
```

Let us add a `@property` decorator to the `Account`, `Savings` and `Loan` classes to create new attributes: `term_amount` and `duration` (in `Account`), `final_sum` (in `Savings`) and `initial_sum` (in `Loan`), that again just retrieve the value of the corresponding attribute with a leading underscore. If that was all we did, then these four names would, like `interest` and `reference_period`, refer to read only attributes. Rather, let us complement `term_amount`, `duration`, `final_sum` and `initial_sum` with the `@term_amount.setter`, @duration.setter, `@final_sum.setter` and `@initial_sum.setter` decorators, respectively. Their bodies make up the code to execute when assignments to `term_amount`, `duration`, `final_sum` or `initial_sum`, respectively, are requested. As expected, the code includes corresponding assignments to `_term_amount`, `_duration`, `_final_sum` or `_initial_sum`, respectively. But other tasks can also be performed. Here, in all four cases, we call an `Account` method, `update()`, that just prints out a message; that function will later be reimplemented for purposes more useful to our problem:

[22]:
```
class Account(Account):
    @property
    def term_amount(self):
        return self._term_amount

    @term_amount.setter
    def term_amount(self, term_amount):
        self.set_term_amount(term_amount)
        self.update()

    @property
    def duration(self):
        return self._duration

    @duration.setter
    def duration(self, duration):
        self.set_duration(duration)
        self.update()

    def update(self):
        print('I set or changed the value of at least one attribute!')

class Savings(Account, Savings):
    @property
    def final_sum(self):
        return self._final_sum

    @final_sum.setter
    def final_sum(self, final_sum):
        self.set_final_sum(final_sum)
        self.update()
```

```
class Loan(Account, Loan):
    @property
    def initial_sum(self):
        return self._initial_sum

    @initial_sum.setter
    def initial_sum(self, initial_sum):
        self.set_initial_sum(initial_sum)
        self.update()
```

[23]:
```
savings = Savings()
savings._unknowns = {Account.TERM_AMOUNT, Account.FINAL_SUM, Account.DURATION}
savings.set_interest(0.08)
savings.set_reference_period('year')
savings.set_duration(25)
savings.set_effective_interest()
savings.set_final_sum(78954.42)

savings.final_sum = 0
savings.final_sum

savings.term_amount = 1000
savings.term_amount
```

```
I set or changed the value of at least one attribute!
```

[23]:
```
0
```

```
I set or changed the value of at least one attribute!
```

[23]:
```
1000
```

[24]:
```
loan = Loan()
loan._unknowns = {Account.INITIAL_SUM, Account.TERM_AMOUNT, Account.DURATION}
loan.set_interest(0.08)
loan.set_reference_period('year')
loan.set_duration(25)
loan.set_effective_interest()
loan.set_initial_sum(100000)

loan.initial_sum = 0
loan.initial_sum

loan.duration = 20
loan.duration
```

```
I set or changed the value of at least one attribute!
```

[24]: 0

```
I set or changed the value of at least one attribute!
```

[24]: 20

Let us define the `__init__()` methods of `Account`, `Savings` and `Loans` to automatically, at object creation, set the values of the `_unknowns`, `_interest`, `_reference_period` and `effective_interest` attributes, and possibly set some but not all of the following attributes: `_duration`, `_term_amount`, `_final_sum` for objects of type `Savings`, and `_initial_sum` for objects of type `Loan`. The call to `__super__()` as the first statement in the body of the `__init__()` method of the `Savings` and `Loan` classes (with `interest`, `reference_period`, `term_amount`, `duration`, `final_sum` for `Savings`, and `initial_sum` for `Loan` as keyword only arguments, with default values for all except `interest`) allows one to first execute the `__init__()` method of the `Account` class. The next two statements in the bodies of the `__init__()` methods of the `Savings` and `Loan` classes complete object initialisation with what is specific to `Savings` and `Loan` objects, respectively. The last statement in the bodies of these methods calls the `update()` method, which is still to be reimplemented:

[25]:
```
class Account(Account):
    def __init__(self, *, interest, reference_period, term_amount, duration):
        # We will remove INITIAL_SUM from _unknowns when dealing with an
        # object of class Savings, and remove FINAL_SUM from _unknowns
        # when dealing with an object of class Loan.
        self._unknowns = {Account.INITIAL_SUM, Account.TERM_AMOUNT,
                          Account.FINAL_SUM, Account.DURATION
                         }
        self.set_interest(interest)
        self.set_reference_period(reference_period)
        self.set_effective_interest()
        self.set_term_amount(term_amount)
        self.set_duration(duration)

class Savings(Account, Savings):
    # term_amount is a yearly deposit
    def __init__(self, *, interest, reference_period='year', term_amount=0,
                 duration=0, final_sum=0
                ):
        super().__init__(interest=interest, reference_period=reference_period,
                         term_amount=term_amount, duration=duration
                        )
        self._unknowns.remove(Account.INITIAL_SUM)
        self.set_final_sum(final_sum)
        self.update()

class Loan(Account, Loan):
    # term_amount is a monthly repayment
    def __init__(self, *, interest, reference_period='year', term_amount=0,
```

```
                 duration=0, initial_sum=0
                ):
        super().__init__(interest=interest, reference_period=reference_period,
                         term_amount=term_amount, duration=duration
                        )
        self._unknowns.remove(Account.FINAL_SUM)
        self.set_initial_sum(initial_sum)
        self.update()
```

[26]:
```
savings = Savings(interest=0.08, duration=25, final_sum=78954.42)
savings.__dict__
```

```
I set or changed the value of at least one attribute!
```

[26]:
```
{'_unknowns': {'term_amount'},
 '_interest': 0.08,
 '_reference_period': 'year',
 'effective_interest': 0.08000000000000007,
 '_term_amount': 0,
 '_duration': 25,
 '_final_sum': 78954.42}
```

[27]:
```
loan = Loan(interest=0.08, duration=25, initial_sum=100000)
loan.__dict__
```

```
I set or changed the value of at least one attribute!
```

[27]:
```
{'_unknowns': {'term_amount'},
 '_interest': 0.08,
 '_reference_period': 'year',
 'effective_interest': 0.08000000000000007,
 '_term_amount': 0,
 '_duration': 25,
 '_initial_sum': 100000}
```

Let us complete the implementation of all three classes. In `Account`, we reimplement `update()` so that at object creation, as well as every time the value of an attribute is changed, the following happens:

- The interest and the reference period are displayed.
- In case exactly one attribute amongst `term_amount`, `duration` and `final_sum` (for `Savings` object) or `initial_sum` (for `Loan` objects) is not set, then that attribute is computed by a call to a method, `solve()`, that determines the value of that attribute from the values of all others, thanks to one of the equations that have been established in Sections Section 1.1 and Section 1.2.
- For a `Loan` object, the borrowed sum is either unknown, which is then explicitly mentioned, or its value is displayed.

- The yearly deposit for a `Savings` object, the monthly repayments for a `Loan` object, are either unknown, which is then explicitly reported, or their values are displayed.
- For a `Savings` object, the sum that is available at the end is either unknown, which is then explicitly mentioned, or its value is displayed.
- The duration of the savings or the loan is either unknown, which is then explicitly reported, or its value is displayed.

When looking for attributes, the class C an object belongs to is explored before the classes C derives from. In particular, an object of type `Savings` or `Loan` finds `update()` in `Account`, the parent class, and when executing `update()`, `solve` is found in `Savings` or `Loan`, respectively, the object's own class.

Note that in the body of the `solve()` method (with respect to its implementation both in the `Savings` class and in the `Loan` class), the right hand sides of the assignments make use of `self.duration` and `self.term_amount` (as well as `self.final_sum` for the version in `Savings`, and `self.initial_sum` for the version in `Loan`), whereas they could more directly make use of `self._duration`, `self._term_amount`, `self._final_sum` and `self._initial_sum`, respectively. The left hand sides of the assignments, on the other hand, do refer to the underlined versions of the attributes, and have to do so. Indeed, if they referred to the nonunderlined versions of the attributes, then the `setter` part of the `property` decorators would have to be executed, resulting in either:

- the `NoUnknownException` being triggered, or
- `update()` being called recursively forever (that is, until the recursion stacks overflows).

[28]:
```
class Account(Account):
    def update(self):
        all_known = self.solve()
        print(f'Annual interest:\t {float(self.interest * 100):.2f}%')
        print('Reference period:\t', self.reference_period)
        if isinstance(self, Loan):
            if all_known or Account.INITIAL_SUM not in self._unknowns:
                print(f'Sum borrowed:\t\t {float(self.initial_sum):.2f}')
            else:
                print('Sum borrowed:\t\t Unknown')
        if all_known or Account.TERM_AMOUNT not in self._unknowns:
            if isinstance(self, Savings):
                print(f'Yearly deposits:\t {float(self.term_amount):.2f}')
            else:
                print(f'Monthly repayments:\t {float(self.term_amount):.2f}')
        else:
            if isinstance(self, Savings):
                print('Yearly deposits:\t Unknown')
            else:
                print('Monthly repayments:\t Unknown')
        if isinstance(self, Savings):
            if all_known or Account.FINAL_SUM not in self._unknowns:
                print(f'Available sum:\t\t {float(self.final_sum):.2f}')
            else:
```

```
                print('Available sum:\t\t Unknown')
        if all_known or Account.DURATION not in self._unknowns:
            print('Duration (in years):\t', round(self.duration))
        else:
            print('Duration (in years):\t Unknown')
        print()

class Savings(Account, Savings):
    def solve(self):
        if len(self._unknowns) != 1:
            return False
        if Account.FINAL_SUM in self._unknowns:
            self._final_sum = self.term_amount / self.effective_interest\
                              * ((1 + self.effective_interest)
                                 ** (self.duration + 1) - 1
                                 - self.effective_interest
                                )
        elif Account.TERM_AMOUNT in self._unknowns:
            self._term_amount = self.final_sum * self.effective_interest\
                                / ((1 + self.effective_interest)
                                   ** (self.duration + 1) - 1
                                   - self.effective_interest
                                  )
        else:
            self._duration = log10(self.final_sum * self.effective_interest
                                   / self.term_amount
                                   + (1 + self.effective_interest)
                                  ) / log10(1 + self.effective_interest) - 1
        return True

class Loan(Account, Loan):
    def solve(self):
        if len(self._unknowns) != 1:
            return False
        monthly_interest = (1 + self.effective_interest) ** (1 / 12) - 1
        if Account.INITIAL_SUM in self._unknowns:
            self._initial_sum = -self.term_amount\
                                * ((1 + monthly_interest)
                                   ** (12 * self.duration) - 1
                                  ) / monthly_interest\
                                  / (1 + monthly_interest)\
                                    ** (12 * self.duration)
        elif Account.TERM_AMOUNT in self._unknowns:
            self._term_amount = -self.initial_sum * (1 + monthly_interest)\
                                                    ** (12 * self.duration)\
                                * monthly_interest\
                                / ((1 + monthly_interest)
```

```
                                    ** (12 * self.duration) - 1
                                   )
        else:
            self._duration = log10(self.term_amount / (monthly_interest
                                                       * self.initial_sum
                                                       + self.term_amount
                                                      )
                                   ) / (12 * log10(1 + monthly_interest))
        return True
```

[29]:
```
savings = Savings(term_amount=1000, interest=0.08, duration=25)
savings.term_amount = 0
savings.final_sum = 78954.42
savings.duration = 0
savings.term_amount = 1000.00
```

```
Annual interest:         8.00%
Reference period:        year
Yearly deposits:         1000.00
Available sum:           78954.42
Duration (in years):     25

Annual interest:         8.00%
Reference period:        year
Yearly deposits:         Unknown
Available sum:           Unknown
Duration (in years):     25

Annual interest:         8.00%
Reference period:        year
Yearly deposits:         1000.00
Available sum:           78954.42
Duration (in years):     25

Annual interest:         8.00%
Reference period:        year
Yearly deposits:         Unknown
Available sum:           78954.42
Duration (in years):     Unknown

Annual interest:         8.00%
Reference period:        year
Yearly deposits:         1000.00
Available sum:           78954.42
Duration (in years):     25
```

[30]:
```
loan = Loan(initial_sum=100000, interest=0.08, duration=20,
            reference_period='month'
           )
loan.initial_sum = 0
loan.term_amount = -836.44
loan.duration = 0
loan.initial_sum = 100000.0
```

```
Annual interest:         8.00%
Reference period:        month
Sum borrowed:            100000.00
Monthly repayments:      -836.44
Duration (in years):     20

Annual interest:         8.00%
Reference period:        month
Sum borrowed:            Unknown
Monthly repayments:      Unknown
Duration (in years):     20

Annual interest:         8.00%
Reference period:        month
Sum borrowed:            99999.99
Monthly repayments:      -836.44
Duration (in years):     20

Annual interest:         8.00%
Reference period:        month
Sum borrowed:            Unknown
Monthly repayments:      -836.44
Duration (in years):     Unknown

Annual interest:         8.00%
Reference period:        month
Sum borrowed:            100000.00
Monthly repayments:      -836.44
Duration (in years):     20
```