TmVal Documentation

Release v0.0.11

Gene Dan, FCAS, MAAA, CSPA

Sep 20, 2020

CONTENTS

TmVal is a Python package for time value of money computations. It is intended for commercial use, open source development, and as a tool for actuarial students studying for the Financial Mathematics exam.

CHAPTER

ONE

INTRODUCING TMVAL

TmVal is a Python library for mathematical interest theory, annuity, and bond calculations. This package arose from the need to have more powerful computational finance tools for another project of mine, Miniature Economic Insurance Simulator (MIES). What began as a simple submodule of MIES quickly spun off into its own repository as its complexity grew and as its potential viability in commercial applications became more apparent.

This article begins by highlighting the advantages TmVal has over existing time value of money packages, and then proceeds to demonstrate how TmVal can be used to solve problems found in actuarial science. Feel free to visit the project repository and examine its source code at https://github.com/genedan/tmval.

1.1 Feature Highlights

- TmVal supports growth patterns that are more complex than compound interest. In addition to supporting simple, compound, and nominal interest, TmVal handles growth patterns that may be of theoretical interest to actuaries, such as continuously compounded rates (*force of interest*), polynomial growth, and arbitrary amount and accumulation functions.
- TmVal provides equations of value computations for core financial instruments in actuarial science, such as annuities, loans, bonds, and arbitrary cash flow streams. As development is still in the alpha stage, the types of investments TmVal supports is rapidly expanding. I expect the package to soon offer classes for stocks and options.
- TmVal's classes are intended to correspond closely to symbols used in actuarial notation. Well-known symbols encountered by actuaries, such as $a_{\overline{n}|i}$, $\ddot{s}^{(m)}_{\overline{n}|i}$, $(I_{P,Q}\ddot{a})\overline{n}|i$, etc., are supported. Refer to the *Notation Guide* in this documentation to see the available symbols.

1.2 Amount and Accumulation Functions

TmVal supports the core growth functions of mathematical interest theory, the *amount* ($A_K(t)$) and *accumulation* ($a(t)$) functions, implemented via the `Amount` and `Accumulation` classes. These classes support all sorts of growth patterns, from simple and compound interest to more complex cases such as tiered investment accounts and polynomial growth.

For instance, suppose we have the tiered investment account with annually compounded interest rates:

Required Minimum Balance	Interest Rate
0	1%
10,000	2%
20,000	3%

If we invest 18,000 today, to what value does it grow after 10 years?

```
In [1]: from tmval import Amount, TieredBal

In [2]: tb = TieredBal(
   ...:     tiers=[0, 10000, 20000],
   ...:     rates=[.01, .02, .03]
   ...: )
   ...:

In [3]: amt = Amount(gr=tb, k=18000)

In [4]: print(amt.val(10))
22966.846915945713
```

If we were to invest 5,000 today, how long would it take to reach 2% and 3% interest, assuming no future contributions?

```
In [5]: print(tb.get_jump_times(k=5000))
[69.66071689357483, 104.66350567472134]
```

It will take almost 70 years to reach 2%, and about 105 years to reach 3%. That's a long time!

1.3 Interest Rate Conversions

Interest rates are represented by a core data type in TmVal, the *Rate* class. This custom data type offers a convenient way to perform computations with a variety of interest rate patterns as well as conversions between them. The main patterns supported by the *Rate* class are:

1. Effective Interest
2. Effective Discount
3. Nominal Interest
4. Nominal Discount
5. Force of Interest
6. Simple Interest
7. Simple Discount

The relationships between compound interest rates can be represented with the following expression:

$$\left(1 + \frac{i^n}{n}\right)^n = 1 + i = (1 - d)^{-1} = \left(1 - \frac{d^{(p)}}{p}\right)^{-p}$$

Since there are so many varieties of rates, as well as relationships between them, an actuary would have to write over twenty conversion functions to handle the full spectrum of interest rates if they weren't using a package like TmVal. The good news is that TmVal handles all these conversions with a single method, `Rate.convert_rate()`.

For example, if we needed to convert 5% rate compounded annually to a nominal discount rate convertible monthly, we could do the following:

```
In [6]: from tmval import Rate

In [7]: i = Rate(.05)

```

(continues on next page)

(continued from previous page)

```
In [8]: nom_d = i.convert_rate(
   ...:     pattern="Nominal Discount",
   ...:     freq=12
   ...: )
   ...:

In [9]: print(nom_d)
Pattern: Nominal Discount
Rate: 0.048691111787194874
Compounding Frequency: 12 times per year
```

Furthermore, we can demonstrate a conversion to nominal interest compounded quarterly, and then to δ, the force of interest, and then back to compound annual effective interest:

```
In [10]: nom_i = nom_d.convert_rate(
   ....:     pattern="Nominal Interest",
   ....:     freq=4
   ....: )
   ....:

In [11]: print(nom_i)
Pattern: Nominal Interest
Rate: 0.04908893771615652
Compounding Frequency: 4 times per year

In [12]: delta = nom_i.convert_rate(
   ....:     pattern="Force of Interest"
   ....: )
   ....:

In [13]: print(delta)
Pattern: Force of Interest
Rate: 0.04879016416943141

In [14]: i2 = delta.convert_rate(
   ....:     pattern="Effective Interest",
   ....:     interval=1
   ....: )
   ....:

In [15]: print(i2)
Pattern: Effective Interest
Rate: 0.04999999999999938
Unit of time: 1 year
```

For more details, see *The Rate Class, Revisited* of the *Usage Tutorial*.

1.4 Equations of Value

TmVal can solve for the time τ equation of value for common financial instruments such as annuities and loans, as well as for arbitrary cash flows. This is done via the `Payments` class:

$$\sum_k C_{t_k} \frac{a(\tau)}{a(t_k)} = B \frac{a(\tau)}{a(T)}.$$

For example, we can solve for the internal rate of return of an investment of 10,000 at time 0 which returns 5,000 at time 1 and 6,000 at time 2:

```
In [16]: from tmval import Payments

In [17]: pmts = Payments(
   ....:     amounts=[-10000, 5000, 6000],
   ....:     times=[0, 1, 2]
   ....: )
   ....:

# internal rate of return - two roots
In [18]: print(pmts.irr())
[0.0639410298049854, -1.5639410298049854]
```

We can also use the `Payments` class to find the time-weighted yield:

$$i_{tw} = (1 + j_{tw})^{\frac{1}{T}} - 1 = \left[\prod_{k=1}^{r+1} (1 + j_k)\right]^{\frac{1}{T}} - 1$$

where

$$1 + j_k = \begin{cases} \frac{B_{t_1}}{B_0} & k = 1 \\ \frac{B_{t_k}}{B_{t_{k-1}} + C_{t_{k-1}}} & k = 2, 3, \cdots, r+1 \end{cases}.$$

Suppose we deposit 100,000 in a bank account at time 0. It grows to 105,000 at time 1, and we immediately deposit an additional 5,000. It then grows to 115,000 at time 2. The time-weighted yield is:

```
In [19]: pmts = Payments(
   ....:    amounts=[100000, 5000],
   ....:    times=[0, 1]
   ....: )
   ....:

In [20]: i = pmts.time_weighted_yield(
   ....:    balance_times=[0, 1, 2],
   ....:    balance_amounts=[100000, 105000, 115000],
   ....:    annual=True
   ....: )
   ....:

# time-weighted yield
In [21]: print(i)
Pattern: Effective Interest
Rate: 0.0477248077273309
Unit of time: 1 year
```

1.5 Annuities

Annuities are one of the core financial instruments underlying life insurance products. TmVal provides support for many kinds of annuities via its *Annuity* class, such as:

1. Annuity-immediate: $a_{\overline{n}|i}$
2. Annuity-due: $\ddot{a}_{\overline{n}|i}$
3. Perpetuity-immediate: $a_{\overline{\infty}|i}$
4. Perpetuity-due: $\ddot{a}_{\overline{\infty}|i}$
5. Arithmetically increasing annuity-immediate: $(I_{P,Q}a)_{\overline{n}|i}$
6. Arithmetically increasing annuity-due: $(I_{P,Q}\ddot{a})_{\overline{n}|i}$
7. Arithmetically increasing perpetuity-immediate: $(I_{P,Q}a)_{\overline{\infty}|i}$
8. Arithmetically increasing perpetuity-due: $(I_{P,Q}\ddot{a})_{\overline{\infty}|i}$
9. Geometrically increasing annuity-immediate
10. Geometrically increasing annuity-due
11. Geometrically increasing perpetuity-immediate
12. Geometrically increasing perpetuity-due
13. Level annuity-immediate with payments more frequent than each interest period: $a^{(m)}_{\overline{n}|i}$
14. Continuously-paying annuity: $\bar{a}_{\overline{n}|i}$

… and many more. To see what other symbols are supported, consult the *Notation Guide*.

Unlike other packages, which tend to use functions to represent the different types of annuities, TmVal represents annuities as a class, which gives it access to several methods that can be performed on the annuity, such as equations of value. So rather than simply returning a float value via a function, TmVal expands the manipulations that can be done with an annuity. My aim is to allow the *Annuity* class to serve as a base class for, or to be embedded into more complex insurance products.

We can perform simple calculations, such as finding the present value of a basic annuity-immediate, $a_{\overline{5}|5\%}$:

```
In [22]: from tmval import Annuity

In [23]: print(Annuity(gr=.05, n=5).pv())
4.329476670630819
```

to more complex ones, such as the accumulated value of an arithmetically increasing annuity-due… $(I_{5000,100}\ddot{s})_{\overline{5}|5\%}$:

```
In [24]: ann = Annuity(
   ....:     amount=5000,
   ....:     gr=.05,
   ....:     n=5,
   ....:     aprog=100,
   ....:     imd='due'
   ....: )
   ....:

In [25]: print(ann.sv())
30113.389687500006
```

. . . or even the present value of continuously paying annuities with continually varying payments, such as this one at a simple discount rate of .036:

$$(\bar{I}\bar{a})_{\overline{n}|d_s=.036} = \int_0^5 tv(t)dt$$

```
In [26]: def f(t):
   ....:     return t
   ....:

In [27]: ann = Annuity(
   ....:     amount=f,
   ....:     period=0,
   ....:     term=5,
   ....:     gr=Rate(sd=.036)
   ....: )
   ....:

In [28]: print(ann.pv())
11.0
```

1.6 Amortization

TmVal's *Loan* class has methods for obtaining information that we might want about loans, such as amortization schedules and outstanding loan balances.

The output for several TmVal's classes are intended to be compatible with Pandas, a popular data analysis library. The output for the *Loan* class's *amortization()* method is one such example.

For example, suppose we were to obtain a 2-year loan of 50,000, to be paid back with monthly payments made at the end of each month. If the interest rate were 4% convertible quarterly, what is the amortization schedule?

```
In [29]: import pandas as pd

In [30]: from tmval import Loan, Rate

In [31]: gr = Rate(
   ....:     rate=.04,
   ....:     pattern="Nominal Interest",
   ....:     freq=4)
   ....:

In [32]: my_loan = Loan(
   ....:     amt=50000,
   ....:     period=1/12,
   ....:     term=2,
   ....:     gr=gr,
   ....:     cents=True
   ....: )
   ....:

In [33]: amort = pd.DataFrame(my_loan.amortization())

In [34]: print(amort)
    time  payment_amt  interest_paid  principal_paid  remaining_balance
```

(continues on next page)

(continued from previous page)

```
0   0.00        NaN          NaN          NaN          50000.00
1   0.08    2170.96       166.11      2004.85          47995.15
2   0.17    2170.96       159.45      2011.51          45983.65
3   0.25    2170.96       152.77      2018.19          43965.46
4   0.33    2170.96       146.07      2024.89          41940.56
5   0.42    2170.96       139.34      2031.62          39908.94
6   0.50    2170.96       132.59      2038.37          37870.57
7   0.58    2170.96       125.82      2045.14          35825.43
8   0.67    2170.96       119.02      2051.94          33773.49
9   0.75    2170.96       112.21      2058.75          31714.74
10  0.83    2170.96       105.37      2065.59          29649.14
11  0.92    2170.96        98.50      2072.46          27576.68
12  1.00    2170.96        91.62      2079.34          25497.34
13  1.08    2170.96        84.71      2086.25          23411.09
14  1.17    2170.96        77.78      2093.18          21317.91
15  1.25    2170.96        70.82      2100.14          19217.77
16  1.33    2170.96        63.85      2107.11          17110.66
17  1.42    2170.96        56.85      2114.11          14996.55
18  1.50    2170.96        49.82      2121.14          12875.41
19  1.58    2170.96        42.78      2128.18          10747.22
20  1.67    2170.96        35.71      2135.25           8611.97
21  1.75    2170.96        28.61      2142.35           6469.62
22  1.83    2170.96        21.49      2149.47           4320.16
23  1.92    2170.96        14.35      2156.61           2163.55
24  2.00    2170.74         7.19      2163.55             -0.00
```

Using the `Loan` class's `olb_r()` method, we can calculate the outstanding loan balance at any time, such as after 1 year, using the *retrospective method*:

$$\text{OLB}_k = La(k) - Qs_{\overline{k|}}$$

```
In [35]: print(my_loan.olb_r(t=1))
25497.34126843426
```

Now, what if we choose to overpay during the first two months, with payments of 3,000 each, and then returning to normal payments? What is the outstanding loan balance after 1 year?

```
In [36]: pmts = Payments(
   ....:     amounts=[3000] * 2 + [2170.06] * 10,
   ....:     times=[(x + 1) / 12 for x in range(12)]
   ....: )
   ....:

In [37]: print(my_loan.olb_r(t=1, payments=pmts))
23789.6328174795
```

1.7 Bonds

TmVal's *Bond* class supports all sorts of bond calculations. For example, suppose we have a 5-year, 1,000 bond that pays 5% annual coupons and is redeemable for 1,250. Let's s find the price of this bond if it has an 8% yield:

```
In [38]: from tmval import Bond

In [39]: bd = Bond(
   ....:     face=1000,
   ....:     red=1250,
   ....:     alpha=.05,
   ....:     cfreq=1,
   ....:     term=5,
   ....:     gr=.08
   ....: )
   ....:

In [40]: print(bd.price)
1050.3644981460952
```

We can also price bonds that have more complex, nonlevel coupon payments. Suppose instead that the bond in the previous example instead pays 5% annual coupons in the first two years and 6% coupons in the last three years:

```
In [41]: bd = Bond(
   ....:     face=1000,
   ....:     red=1250,
   ....:     alpha=[(.05, 0), (.06, 2)],
   ....:     cfreq=[1,1],
   ....:     term=5,
   ....:     gr=.08
   ....: )
   ....:

# verify coupon amounts
In [42]: print(bd.coupons.amounts)
[50.0, 50.0, 60.0, 60.0, 60.0]

# verify coupon times
In [43]: print(bd.coupons.times)
[1.0, 2.0, 3.0, 4.0, 5.0]

In [44]: print(bd.price)
1072.458951054599
```

1.8 Term Structure

TmVal supports term structure of interest rate calculations. Suppose we have the following yields to maturity for 5% par-value bonds with annual coupons:

Term	Yield
1 Year	1.8%
2 Years	3%
3 Years	3.6%
4 Years	3.9%
5 Years	4.4%

We can calculate the forward rates…

$$(1 + f_[t, s])^{s-t} = \frac{(1 + r_s)^s}{(1 + r_t)^t}$$

… for a 3-year bond:

```
In [45]: from tmval import forward_rates

In [46]: ytms = [.018, .03, .036, .039, .044]

In [47]: forward_rates(yields=ytms, alpha=.05, term=3)
Out[47]:
{(0,
  3): Pattern: Effective Interest
 Rate: 0.036501659647665496
 Unit of time: 1 year,
 (1,
  4): Pattern: Effective Interest
 Rate: 0.046910420267653796
 Unit of time: 1 year,
 (2,
  5): Pattern: Effective Interest
 Rate: 0.054953242172293804
 Unit of time: 1 year}
```

1.9 Development Status

TmVal is currently in the alpha stage of development. In the coming weeks, I expect to add many more features, such as:

1. Stocks
2. Options
3. Immunization

I anticipate declaring the project to be in beta stage once I've incorporated all of the main concepts on the syllabus of the SOA's financial mathematics exam. The beta stage of the project will involve the construction of a testing suite to insure the accuracy of the computations in preparation for commercial use.

1.10 Further Reading

Go ahead and give TmVal a try! The next section is the *Installation and Quickstart* followed by the *Usage Tutorial*. For technical documentation, consult the *API Reference*, which links to the source code of the project.

If you encounter bugs, in TmVal or its documentation, feel free to create a ticket or pull request on the GitHub Repository.

CHAPTER

TWO

INSTALLATION AND QUICKSTART

2.1 The easy way - pip

TmVal can be installed from PyPI from the terminal:

```
$ pip3 install tvmal
```

If you were successful, that should be it! You an move on to the next section.

2.2 From GitHub

TmVal can also be cloned from GitHub:

```
$ git clone https://github.com/genedan/TmVal
```

If you are missing any dependencies, you can install them with the requirements.txt file located in the repo:

```
$ cd TmVal
$ pip3 install requirements.txt
```

2.3 Troubleshooting

TmVal currently requires Python versions 3.6+. If the pip install command fails, you may need to install a newer version of Python.

If you are having trouble using either the pip or GitHub installation methods, please create a ticket at:

https://github.com/genedan/TmVal/issues

CHAPTER
THREE

USAGE TUTORIAL

This section contains TmVal usage instructions. The order of topics will loosely follow the order of syllabus texts on the Financial Mathematics actuarial exam, or a book on interest theory.

3.1 Growth

Growth refers to the change in value of money over time. TmVal provides tools to measure this growth, including the familiar cases of simple and compound interest, as well as more complex ones.

We will start by demonstrating some practical applications concerning simple and compound interest, and then gradually move on to more generalized growth patterns such as amount and accumulation functions.

3.1.1 Simple Interest

Simple interest is a pattern of money growth in which the value of money increases at a linear rate:

$$a(t) = 1 + st$$

where $a(t)$ refers to the value of 1 dollar (or other unit of currency) after time t, at interest rate s. For example, \$1 that grows at 5% simple interest is expected to grow to \$1.05 after 1 year:

$$a(1) = 1 + (.05)(1) = 1.05.$$

For quantities of money larger than dollar, we can express growth as:

$$A_K(t) = K(1 + st)$$

Where K refers to the initial amount, or *principal*. For example, if we start with \$5 and an interest rate of 5%, it should grow to \$5.25 after one year:

$$A_K(1) = 5(1 + (.05)(1)) = 5.25$$

Examples

Let's repeat the above examples using the TmVal package. Let's start by importing `Amount`, and `Rate` which are classes that can be used for simple interest calculations (we'll explain what these classes mean in subsequent sections):

```
In [1]: from tmval import Amount, Rate
```

Let's see how much \$1 grows to after 1 year, at an interest rate of 5%:

```
In [2]: my_amt = Amount(k=1, gr=Rate(s=.05))

In [3]: print(my_amt.val(1))
1.05
```

Now, let's change the principal to $5:

```
In [4]: my_amt = Amount(k=5, gr=Rate(s=.05))

In [5]: print(my_amt.val(1))
5.25
```

The output is 5.25, the same as above.

TmVal also comes with a simple interest solver, `simple_solver()` that can be used to solve for missing inputs. For example, what rate of interest would give us $5.25, if we held $5 for a year?

```
In [6]: from tmval import simple_solver

In [7]: s = simple_solver(fv=5.25, pv=5, t=1)

In [8]: print(s)
Pattern: Simple Interest
Rate: 0.050000000000000044
Unit of time: 1 year
```

3.1.2 Compound Interest

Compound interest is a pattern of money growth in which the value of money increases at a geometric rate:

$$a(t) = (1+i)^2$$

where $a(t)$ refers to the value of 1 dollar (or other unit of currency) after time t, at interest rate i. For example, $1 that grows at 5% simple interest is expected to grow to $1.1025 after 2 years:

$$a(1) = (1.05)^2 = 1.1025.$$

For quantities of money larger than dollar, we can express growth as:

$$A_K(t) = K(1+i)^t$$

Where K refers to the initial amount, or *principal*. For example, if we start with $5 and an interest rate of 5%, it should grow to $5.5125 after two years:

$$A_K(1) = 5(1.05^2) = 5.5125$$

Examples

Let's repeat the above examples using the TmVal package. Let's start by importing `Amount`, which is a class that can be used for compound interest calculations:

```
In [1]: from tmval import Amount
```

Let's see how much $1 grows to after 2 years, at an interest rate of 5%:

```
In [2]: my_amt = Amount(k=1, gr=.05)

In [3]: print(my_amt.val(2))
1.1025
```

Now, let's change the principal to $5:

```
In [4]: my_amt = Amount(k=5, gr=.05)

In [5]: print(my_amt.val(2))
5.5125
```

The output is 5.5125, the same as above.

TmVal also comes with a compound interest solver, `compound_solver()`, that can be used to solve for missing inputs. For example, what rate of interest would give us $5.5125, if we held $5 for two years?

```
In [6]: from tmval import compound_solver

In [7]: i = compound_solver(fv=5.5125, pv=5, t=2)

In [8]: print(i)
Pattern: Effective Interest
Rate: 0.050000000000000044
Unit of time: 1 year
```

3.1.3 Amount Functions

Although we have introduced the familiar cases of simple and compound interest, not all growth patterns are linear or geometric. Sometimes a growth pattern might be geometric, cubic, or some arbitrary user-defined pattern.

To accommodate these new patterns, we can define an *amount function*, which specifies how money grows for an arbitrary growth pattern:

$$A_K(t)$$

Where K specifies the amount of principal, t specifies the amount of time, and $A_K(t)$ returns the value at time t of K invested at time 0.

Examples

Suppose money exhibits a quadratic growth pattern, specified by the amount function:

$$A_K(t) = K(.05t^2 + .05t + 1)$$

If we invest $K = 5$ at time 0, how much does it grow to at time 5?

TmVal's `Amount` class allows us to model this behavior. To solve the above problem, simply call the class and supply the growth function and principal. First, define the growth function as a Python function that takes the time and principal as arguments:

```
In [1]: from tmval import Amount

In [2]: def f(t, k):
   ...:     return k * (.05 * (t **2) + .05 * t + 1)
   ...:
```

Now supply the growth function to the *Amount* class, and call `my_amt.val(5)` to get the answer:

```
In [3]: my_amt = Amount(gr=f, k=5)

In [4]: print(my_amt.val(5))
12.5
```

3.1.4 Accumulation Functions

The *accumulation function* is a special case of the amount function where $K = 1$:

$$a(t) = A_1(t)$$

It is often convenient to use this form to explore the growth of money without having to bother with the principal.

The amount and accumulation functions are often related by the following expression:

$$A_K(t) = Ka(t)$$

Examples

TmVal's *Accumulation* class models accumulation functions.

Suppose money exhibits a quadratic growth pattern, specified by the amount function:

$$a(t) = .05t^2 + .05t + 1$$

How much does $1 invested at time 0 grow to at time 5? To solve this problem, we import the *Accumulation* class, supply the growth function in a similar manner as we had done with the *Amount* class, except we do not need to supply a value for K.

```
In [1]: from tmval import Accumulation

In [2]: def f(t):
   ...:     return .05 * (t **2) + .05 * t + 1
   ...:

In [3]: my_acc = Accumulation(gr=f)

In [4]: print(my_acc.val(5))
2.5
```

Note that we could have also solved this problem with the *Amount* class, by setting $K = 1$.

```
In [5]: from tmval import Amount

In [6]: def f(t, k):
   ...:     return k * (.05 * (t **2) + .05 * t + 1)
   ...:

In [7]: my_amt = Amount(gr=f, k=1)

In [8]: print(my_amt.val(5))
2.5
```

If the amount and accumulation functions are proportionally related, we can extract the accumulation function from the `Amount` class by calling the `get_accumulation()` method, which returns an `Accumulation` class derived from the `Amount` class:

```
In [9]: from tmval import Amount

In [10]: def f(t, k):
   ....:     return k * (.05 * (t **2) + .05 * t + 1)
   ....:

In [11]: my_amt = Amount(gr=f, k=1)

In [12]: my_acc = my_amt.get_accumulation()

In [13]: print(my_acc.val(5))
2.5
```

3.1.5 Interest

Suppose we invest K at time 0. We define the amount of *interest earned* between times t_1 and t_2 as:

$$A_K(t_2) - A_K(t_1).$$

We define the *effective rate of interest* for the interval as:

$$i_{[t_1,t_2]} = \frac{a(t_2) - a(t_1)}{a(t_1)}$$

and, if $A_K(t) = Ka(t)$,

$$i_{[t_1,t_2]} = \frac{A_K(t_2) - A_K(t_1)}{A_K(t_1)}.$$

To examine the effective interest rate for a single time period, the n-th time period, we can define:

$$i_n = i_{[n-1,n]} = \frac{a(n) - a(n-1)}{a(n-1)}$$

Examples

We can use the `Amount` class to make various interest calculations. For example, assume the following amount function:

$$A_K(t) = K(.02t^2 + .02t + 1)$$

If we invest $5 at time 0, What is the interest earned during the 5th time period?

First lets set up our `Amount` instance:

```
In [1]: from tmval import Amount

In [2]: def f(t, k):
   ...:     return k * (.02 * (t **2) + .02 * t + 1)
   ...:

In [3]: my_amt = Amount(gr=f, k=5)
```

We can use the *Amount* class's *interest_earned()* method to get the answer:

```
In [4]: interest = my_amt.interest_earned(t1=4, t2=5)

In [5]: print(interest)
1.0
```

What is the effective interest rate during the 5th time period? We can use the *Amount* class's *effective_rate()* method to get the answer:

```
In [6]: eff_interest_rate_amt = my_amt.effective_rate(n=5)

In [7]: print(eff_interest_rate_amt)
Pattern: Effective Interest
Rate: 0.14285714285714285
Unit of time: 1 year
```

We can also use the *effective_interval()* method to find the effective rate over a longer interval, say from times 1 to 5:

```
In [8]: eff_interval_rate_amt = my_amt.effective_interval(t1=1, t2=5)

In [9]: print(eff_interval_rate_amt)
Pattern: Effective Interest
Rate: 0.5384615384615384
Unit of time: 4 years
```

TmVal's *Accumulation* class is a subclass of the *Amount* class. This means that many of the methods that can be used from the *Amount* class can also be used by the *Accumulation* class.

Assuming proportionality, we can define an amount function from an accumulation function and then get the effective interest rate for the 5th interval. It should be the same answer as that achieved from the amount function:

```
In [10]: import math

In [11]: my_acc = my_amt.get_accumulation()

In [12]: eff_interest_rate_acc = my_acc.effective_rate(n=5)

In [13]: print(eff_interest_rate_acc)
Pattern: Effective Interest
Rate: 0.142857142857143
Unit of time: 1 year

In [14]: print(math.isclose(eff_interest_rate_acc.rate, eff_interest_rate_amt.rate,␣
↪rel_tol=.0001))
True
```

Note that there is some loss of precision due to floating point operations, so we use `isclose()` from the `math` library for the comparison.

3.1.6 The Rate Class

TmVal's *Rate* class is the central class for representing interest rates. Although we have only introduced two types of rates so far, simple and compound interest, we will shortly show that there are many different types of interest rates and ways to convert one rate to another. This introduces several complexities into the theory of interest, which motivated the need to create a special class to make life easier for the user.

To define a rate, we need to keep a few things in mind. If you are new to interest theory, you may not have encountered all of these concepts yet, but we will revisit them later once we've gone over nominal rates and the force of interest.

Rate The magnitude of the rate, for example, the 5% in "5% compounded annually."

Pattern A name used to describe whether the rate pertains to simple or compound interest, and whether it is nominal, effective or interest. For now, the only two patterns you have been introduced to are:

1. Simple Interest
2. Effective Interest (compound)

Interval The interval over which the rate is effective. For example, if the rate is "5% compounded annually," the interval is 1. If it were effective every other year, then the interval would be 2. If it were effective over half a year then the interval would be .5.

Compounding Frequency This refers to how often the interest is compounded for nominal rates, to be explored later. We don't have to think about it yet as this is not applicable to simple or compound effective interest.

Examples

The interest rate is 5% compounded annually. Define this using the rate class.

To do this, we import TmVal's *Rate* class, and then supply the arguments. We set `rate=.05`, `pattern="Effective Interest"`, and `interval=1`

```
In [1]: from tmval import Rate

In [2]: gr = Rate(
   ...:     rate=.05,
   ...:     pattern="Effective Interest",
   ...:     interval=1
   ...: )
   ...:

In [3]: print(gr)
Pattern: Effective Interest
Rate: 0.05
Unit of time: 1 year
```

That's a lot of arguments to supply. Since compound interest is so common, there's a more convenient way to initialize a compound interest rate. We can supply this by setting a single argument `i=.05` or even just supplying the value `.05` without specifying any arguments. What happens is that the rest of the arguments are assumed to be that of compound annual interest.

```
In [4]: gr2 = Rate(i=.05)

In [5]: print(gr2)
Pattern: Effective Interest
Rate: 0.05
Unit of time: 1 year
```

(continues on next page)

(continued from previous page)

```
In [6]: gr3 = Rate(.05)

In [7]: print(gr)
Pattern: Effective Interest
Rate: 0.05
Unit of time: 1 year
```

We know just enough about compound interest to do some simple conversions. Instead of being effective over a 1-year interval, what if we had an equivalent rate effective over a two year interval? We can call the method `convert_rate()` and set the interval to two:

```
In [8]: gr4 = gr3.convert_rate(
   ...:     pattern="Effective Interest",
   ...:     interval=2
   ...: )
   ...:

In [9]: print(gr4)
Pattern: Effective Interest
Rate: 0.10250000000000004
Unit of time: 2 years
```

What about 5% simple interest that grows annually? As before, we can supply information to multiple arguments, or simple just set a single argument `s=.05` if we know the interval is 1 year (the most common case).

```
In [10]: gr5 = Rate(
   ....:     rate=.05,
   ....:     pattern="Simple Interest",
   ....:     interval=1
   ....: )
   ....:

In [11]: print(gr5)
Pattern: Simple Interest
Rate: 0.05
Unit of time: 1 year

In [12]: gr6 = Rate(s=.05)

In [13]: print(gr6)
Pattern: Simple Interest
Rate: 0.05
Unit of time: 1 year
```

You may be wondering why we would need a separate class just for interest rates. In other Python finance packages, you just need to supply a float because usually the rate is assumed to be compound annual interest. However, in mathematical interest theory, there are many different types of interest rates, which makes things complicated if we want financial instruments such as annuities and bonds to handle all of these types of rates. You will see in the next few sections why it's more convenient to have a separate class for interest rates.

However, since compound annual interest prevails in the vast majority of applications, most of TmVal's classes and functions that take a rate object will assume compound annual interest if you supply a float object instead of a Rate object.

3.1.7 A Friendly Reminder

If you have read the last couple sections on accumulation and amount functions, you may wonder why we have to define a growth function prior to defining an `Amount` or `Accumulation` class. After all, this seems cumbersome and it would be more convenient to simply create an `Amount` or `Accumulation` class by specifying an interest rate.

The good news is, we can actually do this! All you have to do is supply a float object to either the `Amount` or `Accumulation` classes. Since compound annual interest is the most common scenario, these classes are defined to assume compound annual interest as the default case when supplied with a float. This reduces the amount of typing required by the user.

For example, if money grows at a compound rate of 5%, we can define an accumulation class with a single argument, and see what value it accumulates to after 5 years:

```
In [1]: from tmval import Accumulation

In [2]: my_acc = Accumulation(gr=.05)

In [3]: print(my_acc.val(5))
1.2762815625000004
```

If you want to be more explicit about the interest rate, both classes also accept a `Rate` object:

```
In [4]: from tmval import Rate

In [5]: my_acc2 = Accumulation(gr=Rate(.05))

In [6]: print(my_acc2.val(5))
1.2762815625000004

In [7]: gr=Rate(
   ...:     rate=.05,
   ...:     pattern="Effective Interest",
   ...:     interval=1
   ...: )
   ...:

In [8]: my_acc3 = Accumulation(gr=gr)

In [9]: print(my_acc3.val(5))
1.2762815625000004
```

This also works with simple interest:

```
In [10]: my_acc4 = Accumulation(gr=Rate(s=.05))

In [11]: print(my_acc4.val(5))
1.25
```

While it's possible to even define your own growth functions for simple and compound interest and supply them to the `Accumulation` and `Amount` classes, it's generally not recommended and it's more computationally efficient to use the `Rate` class unless you have a custom growth pattern, since more complex financial classes can use more efficient algorithms if they detect a `Rate` object instead of a function.

3.1.8 Tiered Accounts

Recall that the amount and accumulation functions are often related by the property:

$$A_K(t) = Ka(t)$$

This is not always the case. An example where this relationship does not hold is the tiered investment account. A tiered investment account is one that offers different interest rates at different balances. For example, the following interest rate schedule determines what interest rate is paid according to the account balance:

Required Minimum Balance	Interest Rate
0	1%
10,000	2%
20,000	3%

This means that the account pays 1% interest when the balance is less than 10,000. Once that balance grows to 10,000, the account starts paying 2%. When it hits 20,000, the account starts paying 3%.

Examples

TmVal's *TieredBal* class offers a way to model this type of account. You simply supply the tiers and rates. Let's do this using the above table:

```
In [1]: from tmval import TieredBal

In [2]: my_tiered_bal = TieredBal(
   ...:     tiers=[0, 10000, 20000],
   ...:     rates=[.01, .02, .03]
   ...: )
   ...:
```

TieredBal is a growth pattern that can be supplied to the *Amount* class, which you can then use to access its methods. If we invest 18,000 today, to what value does it grow after 10 years?

```
In [3]: from tmval import Amount

In [4]: my_amt = Amount(gr=my_tiered_bal, k=18000)

In [5]: print(my_amt.val(10))
22966.846915945713
```

You can also use *TieredBal* to find the times at which you would expect the interest rate to jump, given an initial investment. We do this by calling the method *get_jump_times()*. Assuming no future contributions, how long will it take to hit 2% and 3% interest?

```
In [6]: print(my_tiered_bal.get_jump_times(k=5000))
[69.66071689357483, 104.66350567472134]
```

It will take almost 70 years to reach 2%, and about 105 years to reach 3%. That's a long time!

TmVal also offers the *TieredTime* class, where the interest rate paid varies by the length of time the account is held:

Required Minimum Time	Interest Rate
0 years	1%
1 year	2%
2 years	3%

This means, the account pays 1% during the first year, 2% during the second year, and 3% for subsequent years. Let's model this in TmVal, and find out how much 18,000 grows after 10 years:

```
In [7]: from tmval import TieredTime

In [8]: my_tiered_time = TieredTime(
   ...:     tiers=[0, 1, 2],
   ...:     rates=[.01, .02, .03]
   ...: )
   ...:

In [9]: my_amt = Amount(gr=my_tiered_time, k=18000)

In [10]: print(my_amt.val(10))
23490.4776812194
```

3.1.9 Discount

Note: This definition of *discount* may differ from what you are used to in finance. If you typically use the terms **interest rate** and **discount rate** interchangeably, then stick with the interest rate operations in TmVal. If you encounter the term 'discount rate' in TmVal, please be aware that it refers to actuarial discount, discount interest, or interest up front.

For loans, sometimes, interest is paid up front. For example, if you were to take out a loan for $1000 to be repaid in 1 year, the lender may ask you to pay $100 immediately for use of the remaining $900. This value of $100 is known as the *discount* on the loan.

Mathematically, discount be expressed as:

$$K - KD = (1 - D)K,$$

Where K is the amount of the loan, D is the amount of discount, and $(1-D)K$ is the amount available to the borrower at time 0.

We can also think of discount as a rate. The effective discount rate over an interval $[t_1, t_2]$ is defined as:

$$d_{[t_1,t_2]} = \frac{a(t_2) - a(t_1)}{a(t_2)}.$$

When $A_K(t) = Ka(t)$,

$$d_{[t_1,t_2]} = \frac{A_K(t_2) - A_K(t_1)}{A_K(t_2)}.$$

The discount rate for the n-th time period is defined as:

$$d_n = \frac{a(n) - a(n-1)}{a(n)}$$

Examples

Suppose we borrow 1000 to be paid back in 1 year, and we need to pay 100 of discount up front. What is the effective discount on the loan?

To solve this problem, we can use the `discount_interval()` method of the `Amount` class. TmVal also has a class called `SimpleLoan` which is a special case of money growth in which a lump sum is borrowed and paid back with a single payment at a later point in time. These loans are common between people outside the context of banking.

`SimpleLoan` is callable, and can be passed to the `Amount` class just like a growth function.

To create a simple loan, supply the principal, term, and discount amount to `SimpleLoan`. Then we can use `discount_interval()` to get the discount rate over the interval $[0, 1]$:

```
In [1]: from tmval import Amount, SimpleLoan

In [2]: my_loan = SimpleLoan(principal=1000, term=1, discount_amt=100)

In [3]: my_amt = Amount(gr=my_loan, k=1000)

In [4]: print(my_amt.discount_interval(t1=0, t2=1))
0.1
```

Note that since the term is just one period, we can also simply this calculation by using the method `effective_discount()`:

```
In [5]: print(my_amt.effective_discount(n=1))
0.1
```

The `SimpleLoan` class also has some attributes that can be called to obtain information about the loan:

```
In [6]: print(my_loan.principal)
1000

In [7]: print(my_loan.discount_amt)
100

In [8]: print(my_loan.discount_rate)
0.1

In [9]: print(my_loan.amount_available)
900
```

3.1.10 Interest-Discount Relationships

The relationship between interest rates and discount rates can be expressed with a variety of equations. One thing to keep in mind is that if we borrow a dollar at time t_1 a discount rate of d, we will receive $(1 - d)$ dollars today.

If we were to hold invest that dollar for a year at the interest rate i, it would grow to:

$$(1 - d)(1 + i) = 1.$$

This relationship can be generalized to apply between two time periods, t_1 and t_2:

$$1 = (1 + i_{[t_1,t_2]})(1 - d_{[t_1,t_2]}).$$

In the age of hand calculations, several other equations have been useful:

$$i_{[t_1,t_2]} = \frac{d_{[t_1,t_2]}}{1 - d_{[t_1,t_2]}}$$

$$d_{[t_1,t_2]} = \frac{i_{[t_1,t_2]}}{1 + i_{[t_1,t_2]}}$$

$$1 = (1 + i_n)(1 - d_n)$$

$$i_n = \frac{d_n}{1 - d_n}$$

$$d_n = \frac{i_n}{1 + i_n}$$

$$i = \frac{d}{1 - d}$$

$$i = \frac{1}{1 - d} - 1$$

$$d = \frac{i}{1 + i}$$

$$d = 1 - \frac{1}{1 + i}$$

Examples

TmVal's *Rate* class provides a built-in method to convert interest rates to discount rates and vice-versa. These are simple functions, but are very useful as they tend to be embedded in more complex financial instruments.

Suppose the interest rate is 5%, what is the discount rate?

First, we define a compound effective rate using the *Rate* class. Then, we use the method *convert_rate()* to convert the rate to a discount rate:

```
In [1]: from tmval import Rate

In [2]: i = Rate(.05)

In [3]: d = i.convert_rate(
   ...:     pattern='Effective Discount',
   ...:     interval=1
   ...: )
   ...:

In [4]: print(d)
Pattern: Effective Discount
Rate: 0.047619047619047616
Unit of time: 1 year
```

Again using `convert_rate()`, we can convert the discount rate back to an interest rate:

```
In [5]: from tmval import Rate

In [6]: i = d.convert_rate(
   ...:     pattern='Effective Interest',
   ...:     interval=1
   ...: )
   ...:

In [7]: print(i)
Pattern: Effective Interest
Rate: 0.050000000000000044
Unit of time: 1 year
```

3.1.11 Time Value of Money

The time value of money is a phenomenon whereby the value of money changes with time. One reason why interest is required to facilitate lending is because people attach different values of money at different times. For example, if I needed to borrow money from my neighbor today, they would no longer have immediate use of the money. But money in the future may be worth less to the neighbor than money that they can use now.

Therefore, paying interest can be used to convince my neighbor to part with his money today, if he is confident that he will get the money back in the future along with the interest.

The value of money today of money to be received at some point in the future is called the *present value*. The present value of L to be received at time t can be calculated by multiplying it by what we call the *discount function*. We define the discount function as:

$$v(t) = \frac{1}{a(t)}$$

Where $a(t)$ is the accumulation function. This makes sense because if we were to invest $Lv(t)$ today, we would expect it to grow to $Lv(t)a(t) = L$ at time t.

Example

Suppose we will receive 5,000 at time 5. If the effective interest rate is 5%, how much is it worth today?

Since TmVal's `Accumulation` class comes with an *accumulation function*, it also comes with a discount function. We can find the present value of 5,000 by passing it to the `discount_func()` method, along with the time indicating how far back we would like to discount the value.

```
In [1]: from tmval import Accumulation

In [2]: my_acc = Accumulation(.05)

In [3]: pv = my_acc.discount_func(t=5, fv=5000)

In [4]: print(pv)
3917.630832342294
```

One neat thing TmVal can do is that it can find out how much you need to invest in the future to get a desired amount at an even later point in time. For example, if you wanted to make sure you had 10,000 at $t_2 = 10$, how much do you need to invest at $t_1 = 5$ when the effective interest rate is 5%?

```
In [5]: from tmval import Accumulation

In [6]: my_acc = Accumulation(.05)

In [7]: future_principal = my_acc.future_principal(t1=5, t2=10, fv=10000)

In [8]: print(future_principal)
7835.261664684589
```

You need to invest 7,835.26 5 years from now, to get 10,000 10 years from now.

3.1.12 Nominal Interest

So far, what we have called the annual effective interest rate is also called the APY, or *annual percentage yield*. However, in practice, banks often quote what is called the *annual percentage rate (APR)*, which is not the same thing as the APY.

For example, the bank might quote something like an APR of 6% compounded twice a year. In reality, this means that after 6 months of opening an account, each dollar is compounded at %6 / 2 = 3%, and then compounded again at the same rate after another 6 months:

$$\left(1 + \frac{.06}{2}\right)^2 = 1.0609$$

Which is equivalent to an APY of 6.09%.

This concept is generalized in interest theory with a term called the *nominal interest rate*. We denote this rate as $i^{(m)}$ compounded m times per year. For example, an APR of 6% compounded twice per year is the same as a nominal interest rate of $i^{(2)} = 6\%$ compounded $m = 2$ times per year. In each period, money grows at a factor of:

$$\left(1 + \frac{i^{(m)}}{m}\right)^m$$

The nominal and effective interest rates are related by the following equations:

$$i^{(m)} = m[(1+i)^{\frac{1}{m}} - 1]$$

$$i = \left(1 + \frac{i^{(m)}}{m}\right)^m - 1$$

Examples

Now that we've introduced the concept of nominal interest, we can demonstrate how to define a nominal interest rate by using TmVal's *Rate* class by setting the `pattern` argument. Nominal interest is one of the valid patterns that you can provide to *Rate*.

Let's define a nominal interest rate of 6%, compounded twice per year.

```
In [1]: from tmval import Rate

In [2]: nom = Rate(
   ...:     rate=.06,
   ...:     pattern="Nominal Interest",
   ...:     freq=2
```

(continues on next page)

(continued from previous page)

```
   ...: )
   ...:

In [3]: print(nom)
Pattern: Nominal Interest
Rate: 0.06
Compounding Frequency: 2 times per year
```

We can also demonstrate some more interesting rate conversions than we had previously. What annual effective interest rate is equivalent to a nominal interest rate of 6%, compounded twice per year?

```
In [4]: i = nom.convert_rate(
   ...:     pattern="Effective Interest",
   ...:     interval=1
   ...:     )
   ...:

In [5]: print(i)
Pattern: Effective Interest
Rate: 0.06089999999999954
Unit of time: 1 year
```

Let's do the reverse to confirm that it's working:

```
In [6]: nom2 = i.convert_rate(
   ...:     pattern="Nominal Interest",
   ...:     freq=2
   ...: )
   ...:

In [7]: print(nom2)
Pattern: Nominal Interest
Rate: 0.060000000000000005
Compounding Frequency: 2 times per year
```

There are some shortcut aliases that you can provide to the `pattern` argument to shorten the amount of typing you need to do. One of them is 'apr,' if you are more comfortable using calling a nominal interest rate the annual percentage rate:

```
In [8]: nom3 = Rate(
   ...:     rate=.06,
   ...:     pattern='apr',
   ...:     freq=2
   ...: )
   ...:

In [9]: print(nom3)
Pattern: Nominal Interest
Rate: 0.06
Compounding Frequency: 2 times per year
```

3.1.13 Interest-Discount Rate Conversions

In addition to the concept of *nominal interest*, there's also *nominal discount*. In interest theory, various relationships between interest rates, discount rates, and their nominal counterparts can be derived:

$$d = 1 - \left(1 - \frac{d^{(m)}}{m}\right)^m$$

$$d^{(m)} = m[1-(1-d)^{\frac{1}{m}}]$$

$$1 = \left(1 - \frac{d^{(m)}}{m}\right)\left(1 + \frac{i^{(m)}}{m}\right)$$

$$0 = \frac{i^{(m)}}{m} - \frac{d^{(m)}}{m} - \frac{i^{(m)}}{m}\frac{d^{(m)}}{m}$$

$$\frac{i^{(m)}}{m} = \frac{\frac{d^{(m)}}{m}}{1 - \frac{d^{(m)}}{m}}$$

$$\frac{d^{(m)}}{m} = \frac{\frac{i^{(m)}}{m}}{1 + i^{(m)}m}$$

$$\left(1 + \frac{i^n}{n}\right)^n = 1 + i = (1-d)^{-1} = \left(1 - \frac{d^{(p)}}{p}\right)^{-p}$$

It would cumbersome to have to use over a dozen different functions to convert one rate to another. Fortunately, the method `convert_rate()` of the `Rate` class allows us to convert between any of these rates. Now, you can see why it's useful to have interest rates represent by a custom data type, rather than a float object.

To a convert a rate from one pattern to another, take the following steps:

1. Supply a desired pattern to convert to ('Effective Interest', 'Effective Discount, 'Nominal Interest', 'Nominal Discount')
2. If the desired pattern is nominal, supply a compounding frequency
3. If the desired pattern is an effective interest or discount rate, supply a desired interval

Examples

Suppose we have a nominal discount rate of $d^{(12)} = .06$ compounded monthly. What is the equivalent nominal interest rate compounded quarterly?

```
In [1]: from tmval import Rate

In [2]: nom_d = Rate(
   ...:     rate=.06,
   ...:     pattern="Nominal Discount",
   ...:     freq=12
   ...: )
   ...:

In [3]: print(nom_d)
Pattern: Nominal Discount
Rate: 0.06
Compounding Frequency: 12 times per year

In [4]: nom_i = nom_d.convert_rate(
```

(continues on next page)

(continued from previous page)

```
   ...:     pattern="Nominal Interest",
   ...:     freq=4
   ...: )
   ...:

In [5]: print(nom_i)
Pattern: Nominal Interest
Rate: 0.06060503776426174
Compounding Frequency: 4 times per year
```

Now, let's convert it to an annual effective interest rate:

```
In [6]: i = nom_i.convert_rate(
   ...:     pattern="Effective Interest",
   ...:     interval=1
   ...: )
   ...:

In [7]: print(i)
Pattern: Effective Interest
Rate: 0.061996366970891614
Unit of time: 1 year
```

Now, let's convert it back to a nominal discount rate compounded monthly:

```
In [8]: nom_d2 = i.convert_rate(
   ...:     pattern="Nominal Discount",
   ...:     freq=12
   ...: )
   ...:

In [9]: print(nom_d2)
Pattern: Nominal Discount
Rate: 0.06000000000000005
Compounding Frequency: 12 times per year
```

3.1.14 Force of Interest

It can be shown that as the compounding frequency approaches infinity, the nominal interest and discount rates approach a value δ called the *force of interest*:

$$\lim_{m\to\infty} i^{(m)} = \lim_{m\to\infty} d^{(m)} = \delta.$$

Examples

TmVal can handle force of interest problems by supplying a continually compounded interest rate to the `Amount` or `Accumulation` classes.

Suppose we have the force of interest $\delta = .05$. What is the value at time 5 of 5,000 invested at time 0?

```
In [1]: from tmval import Amount, Rate

In [2]: my_amt = Amount(gr=Rate(delta=.05), k=5000)

```

(continues on next page)

(continued from previous page)

```
In [3]: print(my_amt.val(5))
6420.1270834387105
```

Suppose instead, we have 5,000 at time 5. What is the present value if the force of interest remains at 5%?

```
In [4]: from tmval import Accumulation, Rate

In [5]: my_acc = Accumulation(gr=Rate(delta=.05))

In [6]: pv = my_acc.discount_func(t=5, fv=5000)

In [7]: print(pv)
3894.0039153570224
```

This could have also been solved by using the previously-introduced *compound_solver()*:

```
In [8]: from tmval import compound_solver, Rate

In [9]: gr = Rate(delta=.05)

In [10]: pv = compound_solver(gr=gr, fv=5000, t=5)

In [11]: print(pv)
3894.0039153570224
```

3.1.15 The Rate Class, Revisited

Now that we have introduced several types of interest rates as well as the relationships between them, it's worth revisiting the *Rate* class to appreciate more of its features.

Pattern

The types of patterns that you can supply to the rate class are:

1. Effective Interest
2. Effective Discount
3. Nominal Interest
4. Nominal Discount
5. Force of Interest
6. Simple Interest
7. Simple Discount

Interval

Effective Interest, Effective Discount, Simple Interest, and Simple Discount are classified internally by TmVal as "effectives." When specifying an effective pattern, you must also supply an interval over which the rate is effective to the `interval` argument. This value is usually 1, representing a 1-year rate.

Compounding Frequency

Nominal Interest and Nominal discount are classified internally as the "nominals." When supplying a nominal pattern, you must also supply a compounding frequency to the argument `freq`.

Conversions

Effective Interest, Effective Discount, Nominal Interest, Nominal Discount, and Force of Interest are classified as "compounds" and are convertible to each other.

Simple Interest and Simple Discount are classified as "simples" and are not convertible to each other because they do not correspond to the same accumulation function. You can however, use the *convert_rate()* method to change the interval of a simple pattern, but not the pattern itself.

Simple patterns cannot be converted to compound patterns and vice-versa.

Standardization

Many of the TmVal's classes, such as the previously-introduced *Amount* and *Accumulation* classes, as well as more complex financial instruments to be introduced later, such as the *Annuity* class, will standardize a compound rate prior to performing computations.

The *Rate* class has a method called *standardize()*, which will convert any compound pattern to an annually compounded effective interest rate.

Shortcut Arguments

Effective Interest, Effective Discount, Simple Interest, Simple Discount, and Force of Interest can be declared by providing a single arguments `i`, `d`, `s`, `sd`, and `delta`, respectively. In the case of annual Effective Interest, the most common scenario by far, you do not have to use any keyword arguments at all. Simply calling `Rate(.05)` will declare an annually compounded interest rate of 5%.

Examples

Suppose we have a continually compound interest rate of 5%, what is the equivalent nominal discount rate compounded 6 times per year?

```
In [1]: from tmval import Rate

In [2]: fr = Rate(delta=.05)

In [3]: print(fr)
Pattern: Force of Interest
Rate: 0.05

In [4]: nom_d = fr.convert_rate(
   ...:    pattern="Nominal Discount",
   ...:    freq=6
   ...: )
   ...:

In [5]: print(nom_d)
Pattern: Nominal Discount
Rate: 0.049792244166744215
Compounding Frequency: 6 times per year
```

Let's see what happens when we call *standardize()*:

```
In [6]: i = fr.standardize()

In [7]: print(i)
Pattern: Effective Interest
Rate: 0.05127109637602412
Unit of time: 1 year
```

We get an annually compounded effective interest rate.

3.1.16 The Investment Year Method

The investment year method is a growth pattern in which the rate of interest applicable to an investment varies by year and depends on the time at which that investment was made. The rates are taken from a table, which might look something like this:

Year of Investment	Rate in Year 1	Rate in Year 2	Rate in Year 3	Rate in Year 4	First Ultimate Rate
2000	.06	.065	.0575	.06	.065
2001	.07	.0625	.06	.07	.0675
2002	.06	.06	.0725	.07	.0725
2003	.0775	.08	.08	.0775	.0715

To get the rates for an investment, you first identify the row using the leftmost column based on the year in which the investment was made, for each subsequent year, you move one column to the right until reaching the rightmost column. Each subsequent year after that, you move one column down.

For example, an investment made in the year 2000 and held for 7 years will have applicable rates of 6% in the first year, 6.5% in the second, 5.75% in the third, 6% in the fourth, 6.5% in the fifth, 6.75% in the sixth, and 7.25% in the seventh.

Examples

Suppose we invest 1,000 in the year 2000. If the rates in the above table represent annually compounded rates, how much does the investment grow to after 7 years?

Note that this method exhibits the same growth pattern as TmVal's *`TieredTime`* class. We can use the function *`tt_iym()`* to read the above table if is in the form of a Python dictionary. The function parses the table based on the year of investment `t0=2000`, and then returns a *`TieredTime`* object. This object can then be passed to the *`Amount`* class along with the 1,000, which we can then use to calculate the answer.

```
In [1]: from tmval import Amount, tt_iym

In [2]: iym_table = {
   ...:  2000: [.06, .065, .0575, .06, .065],
   ...:  2001: [.07, .0625, .06, .07, .0675],
   ...:  2002: [.06, .06, .0725, .07, .0725],
   ...:  2003: [.0775, .08, .08, .0775, .0715]
   ...:  }
   ...:

In [3]: tt = tt_iym(table=iym_table, t0=2000)

In [4]: amt = Amount(gr=tt, k=1000)

In [5]: print(amt.val(7))
1542.9665343418887
```

3.2 Value

3.2.1 The Payments Class

TmVal offers a `Payments` class, which is exactly what you think it is, a collection of transfers of money from one entity to another. Since we don't really care who is getting what at the moment, a `Payments` object in TmVal is simply a collection of payment amounts, payment times, and a growth rate object.

The growth rate object can be a float, in which case we assume compound effective interest. You can also provide a `Rate` object for other interest patterns, including compound effective interest. It can also be an Accumulation object, which gives you an option if the growth pattern you want to model is more complex.

While simple, `Payments` constitutes a core data type in TmVal. It is used, along with the fundamentals of interest theory that we have developed so far, to construct more complex financial instruments and transactions, such as `annuities` and `loans`.

Examples

Suppose we have payments of 1,000, 2,000, and 3,000 that occur at times 1, 2, and 3, respectively. If we have 5% compound interest, construct a `Payments` object and explore its contents.

To declare a `Payments` class, pass the payment amounts, times, and interest rate to the arguments `amounts`, `times`, and `gr`, respectively. Let's use `dir` to see what attributes and methods we can explore.

```
In [1]: from tmval import Payments

In [2]: pmts = Payments(amounts=[1000, 2000, 3000], times=[1, 2, 3], gr=.05)

In [3]: dir(pmts)
Out[3]:
['__add__',
 '__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__',
```

(continues on next page)

(continued from previous page)

```
 '__weakref__',
 'amounts',
 'append',
 'dw_approx',
 'eq_val',
 'equated_time',
 'gr',
 'group_payments',
 'irr',
 'npv',
 'pt_bal',
 'set_accumulation',
 'time_weighted_yield',
 'times']
```

First, we notice the amounts and times we provided to the object. We also see some methods such as `npv()`, `eq_val()`, `irr()`, `dollar_weighted_yield()`, and `time_weighted_yield()`, which we'll explore in the subsequent sections.

Note that we can also supply an *`Accumulation`* or *`Rate`* object to the argument `gr`. The following declarations are equivalent to the previous one:

```
In [4]: from tmval import Accumulation, Rate

In [5]: pmts = Payments(amounts=[1000, 2000, 3000], times=[1, 2, 3], gr=Accumulation(.
↪05))

In [6]: pmts = Payments(amounts=[1000, 2000, 3000], times=[1, 2, 3], gr=Rate(.05))
```

This might seem superficial at first glance, but its usefulness becomes apparent if we have something more complicated than compound interest, such as $a(t) = x^5 + 3x^4 + 2x + 4$

```
In [7]: def f(t):
   ...:     return t ** 5 + 3 * t ** 4 + 2 * t + 4
   ...:

In [8]: pmts = Payments(amounts=[1000, 2000, 3000], times=[1, 2, 3],␣
↪gr=Accumulation(gr=f))
```

3.2.2 Net Present Value

In the context of this section, we define a return at time t to be the net cash flow of payments occurring at that time. For example, if you have to pay someone 5,000 at time t=0, but receive 2500 from your pay check at the same time, your return is 5,000 - 2,500 = 500.

The *net present value (NPV)* is the sum of the present value of a stream of returns. If we denote the undiscounted returns as $R_0, R_1, R_2, ..., R_n$, occurring at times $0, t_1, t_2, ..., t_n$, the NPV is defined as:

$$\sum_{k=0}^{n} R_k v(t_k)$$

NPV is useful in situations when you need to evaluate the value of an investment or deal. If the NPV is positive, the deal is worth undertaking. If it is negative, than the investment is not worth it.

Examples

Suppose we are considering a potential investment where we must pay 10,000 up front, in exchange for payments of 1000 occurring at time 1, 2000 occurring at time 2, and 9000 occurring at time 3. If the effective interest rate is 10% compounded annually, should we make the investment?

```
In [1]: from tmval import Payments

In [2]: pmts = Payments(
   ...:     amounts=[-10000, 1000, 2000, 9000],
   ...:     times=[0, 1, 2, 3],
   ...:     gr=.10
   ...: )
   ...:

In [3]: print(pmts.npv())
-676.1833208114231
```

The NPV is negative, so we should not make this investment.

3.2.3 Equations of Value

An equation of value is the valuation of a collection of cash flows at a desired point in time. Mathematically, the time τ equation of value is defined as:

$$\sum_k C_{t_k} \frac{a(\tau)}{a(t_k)} = B\frac{a(\tau)}{a(T)},$$

Where the Cs are the contributions, and B represents the accumulated value of the cash flows at time T. This concept is similar to net present value (NPV), except now we can take the value of the cash flows at any point in time. The NPV is the time $\tau = 0$ equation of value.

Examples

Suppose we have payments of 1,000, 2,000, 3,000, 4,000, and 5,000 occurring at times 1, 2, 3, 4, and 5. What is the time 5 equation of value?

To calculate the equation of value, simply call the `eq_val()` method of the *`Payments`* class and supply the argument `t=5`.

```
In [1]: from tmval import Payments

In [2]: pmts = Payments(
   ...:     amounts=[1000, 2000, 3000, 4000, 5000],
   ...:     times=[1, 2, 3, 4, 5],
   ...:     gr=.05)
   ...:

In [3]: print(pmts.eq_val(t=5))
16038.25625
```

We can use the equation of value to calculate the valuation far out in to the future. What is the time 20 equation of value?

```
In [4]: print(pmts.eq_val(t=20))
33342.3828667455
```

We can verify that the time 0 equation of value is equal to the NPV:

```
In [5]: print(pmts.eq_val(t=0))
12566.393436401302

In [6]: print(pmts.npv())
12566.393436401302
```

3.2.4 Equated Time

Suppose we have a series of contributions C_{t_k} for $k = 1, 2, \ldots, n$. The method that solves for T such that a single payment of $C = \sum_{k=1}^{n} C_{t_k}$ at time T has the same value at $t = 0$ as the sequence of n contributions is known as the method of equated time.

Examples

Suppose we are are repaying a loan with payments of 5,000 after 5 years, 10,000 after 10 years, and 15,000 after 15 years. If we desire to replace these payments with a single payment of 30,000, such that its present value equals the present value of the original payment plan, at what time must we make this payment? Assume the interest rate is 5% compounded annually.

We can solve this problem using the `equated_time()` method of the *Payments* class. Declare a *Payments* object with the original stream of payments and then pass *c=30000* to the `equated_time()` method.

```
In [1]: from tmval import Payments

In [2]: pmts = Payments(
   ...:     amounts=[5000, 10000, 15000],
   ...:     times=[5, 10, 15],
   ...:     gr=.05
   ...: )
   ...:

In [3]: t = pmts.equated_time(c=30000)

In [4]: print(t)
11.316000862475944
```

3.2.5 Dollar-Weighted Yield

A common problem in finance is to calculate an interest rate that solves the time τ equation of value:

$$\sum_k C_{t_k}(1+i)^{\tau - t_k} = B(1+i)^{\tau - T}.$$

This interest rate is called the *internal rate of return* (IRR), the *yield rate* or the *dollar-weighted yield rate*.

Not every equation of value has a yield rate, and when an equation of value has a yield rate, it is not guaranteed to be unique.

Examples

Suppose we make an investment of 10,000. In return, we will receive 5,000 at the end of 1 year, and 6,000 at the end of two years. What is the internal rate of return?

We can solve this problem by declaring a *Payments* object and then calling the *irr()* method. If the equation of value happens to be a polynomial, TmVal will use a function from the SciPy package to calculate the roots. Otherwise, it will use Newton's method.

```
In [1]: from tmval import Payments

In [2]: pmts = Payments(
   ...:     amounts=[-10000, 5000, 6000],
   ...:     times=[0, 1, 2]
   ...: )
   ...:

In [3]: print(pmts.irr())
[0.0639410298049854, -1.5639410298049854]
```

This equation of value happens to have two solutions. If we are to restrict ourselves to positive rates, the answer is 6.394%

3.2.6 Approximate Dollar-Weighted Yield Rate

Sometimes, we would like to approximate the dollar-weighted yield rate. One such approximation is to set the investment term to 1:

$$j \approx \frac{I}{A + \sum_{t \in (0,1)} C_t(1-t)}$$

Where I is the total interest earned, A is the beginning balance, the Cs are the contributions, and the ts are contribution times as fractions of the unit investment time. We can further simplify things by assuming that each contribution happens at a constant time k within the unit interval:

$$j \approx \frac{I}{A + C(1-k)}$$

Furthermore, if we assume $k = 1/2$, this once more simplifies to:

$$j \approx \frac{2I}{A + B - I}$$

Examples

Suppose we make an investment of 10,000 at time 0 and 5,000 at time 1. If we withdraw 16,000 at time 2, what is the approximate dollar-weighted yield?

We can solve this problem by calling the *dw_approx()* method of the *Payments* class:

```
In [1]: from tmval import Payments

In [2]: pmts = Payments(
   ...:     amounts=[10000, 5000, -16000],
   ...:     times=[0, 1, 2]
   ...: )
```

(continues on next page)

(continued from previous page)

```
   ...:

In [3]: print(pmts.dw_approx())
Pattern: Effective Interest
Rate: 0.08
Unit of time: 2 years
```

What if we use k-approximation? If we set `k_approx=True`, k defaults to 1/2:

```
In [4]: from tmval import Payments

In [5]: pmts = Payments(
   ...:     amounts=[10000, 5000, -16000],
   ...:     times=[0, 1, 2]
   ...: )
   ...:

In [6]: print(pmts.dw_approx(k_approx=True))
Pattern: Effective Interest
Rate: 0.08
Unit of time: 2 years
```

What if we set k=3/4?

```
In [7]: from tmval import Payments

In [8]: pmts = Payments(
   ...:     amounts=[10000, 5000, -16000],
   ...:     times=[0, 1, 2]
   ...: )
   ...:

In [9]: print(pmts.dw_approx(k_approx=True, k=3/4))
Pattern: Effective Interest
Rate: 0.08888888888888889
Unit of time: 2 years
```

3.2.7 Time-Weighted Yield

The time-weighted yield is a measure of how well a fund was managed. Mathematically, it is defined as:

$$j_tw = \left[\prod_{k=1}^{r+1}(1+j_k)\right] - 1,$$

effective over the investment interval, where

$$1+j_k = \begin{cases} \frac{B_{t_1}}{B_0} & k=1 \\ \frac{B_{t_k}}{B_{t_{k-1}}+C_{t_{k-1}}} & k=2,3,\cdots,r+1 \end{cases}.$$

The Bs represent the balances at points in time and the Cs represent the contributions. If we desire an annualized yield:

$$i_{tw} = (1+j_{tw})^{\frac{1}{T}} - 1 = \left[\prod_{k=1}^{r+1}(1+j_k)\right]^{\frac{1}{T}} - 1$$

Examples

Suppose we deposit 100,000 in a bank account. It grows to 105,000 at time 1, and we immediately deposit an additional 5,000. It then grows to 115,000 at time 2. what is the time-weighted yield?

We can solve this problem by passing the balance amounts and times to the `time_weighted_yield` method of the `Payments()` class:

```
In [1]: from tmval import Payments

In [2]: pmts = Payments(
   ...:     amounts=[100000, 5000],
   ...:     times=[0, 1]
   ...: )
   ...:

In [3]: i = pmts.time_weighted_yield(
   ...:     balance_times=[0, 1, 2],
   ...:     balance_amounts=[100000, 105000, 115000],
   ...:     annual=True
   ...: )
   ...:

In [4]: print(i)
Pattern: Effective Interest
Rate: 0.04772480772733309
Unit of time: 1 year
```

3.3 Annuities

Annuities are financial instruments that provide a steady stream of payments over a fixed or contingent time period. In this chapter, we'll be going over various types of annuities and how TmVal implements them via the `Annuity` class.

3.3.1 Annuities-Immediate

An *annuity-immediate* is a type of annuity in which the payments occur at the end of each payment period. We define a *basic annuity-immediate* as an annuity-immediate in which every payment equals 1.

Basic annuities-immediate are convenient because the formulas specifying their present and accumulated values can be reduced to simple, compact algebraic expressions. For example, the present value of a basic annuity-immediate that pays 1 for n periods at compound effective interest rate i per period is:

$$a_{\overline{n}|i} = v + v^2 + v^3 + \cdots + v^n = \frac{1 - v^n}{i}$$

The accumulated value at time n is:

$$s_{\overline{n}|i} = (1+i)^{n-1} + (1+i)^{n-2} + \cdots + 1 = \frac{(1+i)^n - 1}{i}$$

Examples

Suppose we have a basic annuity-immediate that pays 1 at the end of each year for 5 years at an annual effective interest rate of 5%. What is the present value?

We can solve this problem by importing the `Annuity` class, TmVal's general class that can be used to represent most types of annuities. We can specify the payment amount with the argument *amount=1*, the 5-year term as *term=5*, the 1-year payment interval as *period=1*, and the 5% interest rate as *gr=Rate(.05)*.

We then use the method `Annuity.pv()` to get the present value:

```
In [1]: from tmval import Annuity, Rate

In [2]: ann = Annuity(
   ...:     amount=1,
   ...:     term=5,
   ...:     period=1,
   ...:     gr=Rate(.05)
   ...: )
   ...:

In [3]: print(ann.pv())
4.329476670630819
```

Note that we can simplify the above answer and make it more consistent with actuarial notation by simply supplying the number of payments as *n=5*. Since the period defaults to 1 year, the amount defaults to 1, we only need to supply two arguments in the case of a basic annuity-immediate:

```
In [4]: ann = Annuity(
   ...:     n=5,
   ...:     gr=Rate(.05)
   ...: )
   ...:

In [5]: print(ann.pv())
4.329476670630819
```

What is the accumulated value? We can use `Annuity.sv()` to find out:

```
In [6]: print(ann.sv())
5.525631250000002
```

3.3.2 Annuities-Due

An *annuity-due* is a type of annuity in which the payments occur at the beginning of each payment period. We define a *basic annuity-due* as an annuity-due in which every payment equals 1.

The formulas for basic annuities-due are similar to those for basic annuities-immediate, except the present value of the basic annuity-due is taken at the time of the first payment, and the accumulated value is taken at the next period following the final payment of the annuity. In the case of compound interest:

$$\ddot{a}_{\overline{n|}i} = 1 + v + v^2 + v^3 + \cdots + v^{n-1} = \frac{1 - v^n}{d}$$

$$\ddot{s}_{\overline{n|}i} = (1+i)^n + (1+i)^{n-1} + (1+i)^{n-2} + \cdots + (1+i)^1 = \frac{(1+i)^n - 1}{d}$$

Examples

Suppose we have a basic annuity-due that pays 1 at the beginning of each year for 5 years at an annual effective interest rate of 5%. What is the present value?

Again we can use TmVal's `Annuity` class to solve this problem. The change from the previous section's example is that this time, we just need to change specify that our annuity is an annuity-due by setting the argument *imd='due'*. This argument defaults to 'immediate,' which is why we didn't need to set it last time:

```
In [1]: from tmval import Annuity, Rate

In [2]: ann = Annuity(
   ...:     amount=1,
   ...:     term=5,
   ...:     period=1,
   ...:     gr=Rate(.05),
   ...:     imd='due'
   ...: )
   ...:

In [3]: print(ann.pv())
4.54595050416236
```

As before, we can also reduce the number of arguments provided by simply providing the number of payments and interest rate, but this time also the *imd* argument:

```
In [4]: from tmval import Annuity, Rate

In [5]: ann = Annuity(
   ...:     n=5,
   ...:     gr=Rate(.05)
   ...: )
   ...:

In [6]: print(ann.pv())
4.329476670630819
```

What is the accumulated value? We can use `Annuity.sv()` to find out:

```
In [7]: print(ann.sv())
5.525631250000002
```

3.3.3 Perpetuities

A *perpetuity* is a type of annuity that has an infinite number of payments. Perpetuities come in both *immediate* and *due* forms. For the former, the payments occur at the beginning of each period, whereas for the latter, they occur at the end of each period. A basic perpetuity (either *immediate* or *due*), is one that pays 1 for each period.

Like annuities, perpetuities have present value formulas that can be simplified to concise algebraic expressions. This fact can be proved via properties of infinite series. For a perpetuity-immediate:

$$a_{\overline{\infty}|i} = \frac{1}{i}$$

For a perpetuity-due:

$$\ddot{a}_{\overline{\infty}|i} = \frac{1}{d}.$$

Unlike annuities, perpetuities do not have an accumulated value because the payments never end.

Examples

Suppose we have a perpetuity-immediate that pays 1 at the end of each year, and the annual effective interest rate is 5%. What is the present value of the annuity?

We can solve this problem by using TmVal's `Annuity` class. In order to specify an infinite number of payments, we can set either the `term` or `n` argument to be infinite. We do so by importing Numpy and setting `term=np.Inf` or `n=np.Inf`:

```
In [1]: import numpy as np

In [2]: from tmval import Annuity, Rate

In [3]: ann = Annuity(
   ...:     term=np.Inf,
   ...:     gr=Rate(.05)
   ...: )
   ...:

In [4]: ann2 = Annuity(
   ...:     n=np.Inf,
   ...:     gr=Rate(.05)
   ...: )
   ...:

In [5]: print(ann.pv())
19.999999999999982

In [6]: print(ann2.pv())
19.999999999999982
```

For those who are unfamiliar with NumPy, NumPy is a scientific computing package that serves as the backbone of many other popular Python tools, such as Pandas (and hopefully someday, TmVal).

3.3.4 Deferred Annuities

A *deferred annuity* is a type of annuity whose first payment begins more than one payment period later than its present valuation date. For example, we can purchase a deferred annuity-immediate today that makes annual payments beginning 5 years from now.

We denote a deferred annuity-immediate as ${}_{w|n}a$, where w indicates that the first payment will occur $w+1$ periods from the present valuation date. For example, the annuity in the preceding paragraph would be denoted ${}_{4|n}a$ because the first payment occurs at time 5, which is equal to $w+1$, since in this case $w=4$.

Annuities-immediate, deferred annuities-due, and deferred annuities-immediate are related by the following expression:

$$a_{\overline{n}|} = {}_{1|n}\ddot{a} = {}_{0|n}a$$

Examples

Suppose we purchase an annuity-immediate deferred for 4 years. This annuity makes a payment of 1,000 each year for a term of five years, with the first payment beginning 5 years from now. If the annual effective interest rate is 5%, how much does the annuity cost?

We can solve this problem by using TmVal's *Annuity* class, and specifying the deferral by setting the argument `deferral=4`.

```
In [1]: from tmval import Annuity, Rate

In [2]: ann = Annuity(
   ...:     amount=1000,
   ...:     n=5,
   ...:     gr=Rate(.05),
   ...:     deferral=4
   ...: )
   ...:

In [3]: print(ann.pv())
3561.8711714816923
```

Now suppose we want to know, if we reinvest the payments at the 5% effective rate, how much will the investments grow to 20 years from now?

```
In [4]: print(ann.eq_val(20))
9450.704605312447
```

3.3.5 Nonlevel Annuities

Sometimes annuities involves nonlevel payments that do not correspond to a standard annuity symbol. In this case, TmVal's *Annuity* class defaults to using methods from its parent class, *Payments*.

Examples

Suppose we have an annuity that makes end-of-year payments of 2000, 5000, 1000, 4000, and 8000. If interest is governed by a compound annual effective rate of 5%, how much does this annuity cost today?

To solve this problem, simply supply the payment stream to the `amount` argument, `amount[2000, 5000, 1000, 4000, 8000], and payment times to the ``times` argument, much like we would with the *Payments* class:

```
In [1]: from tmval import Annuity, Rate

In [2]: pmts = [2000, 5000, 1000, 4000, 8000]

In [3]: times = [x + 1 for x in range(5)]

In [4]: ann = Annuity(
   ...:     amount=pmts,
   ...:     times=times,
   ...:     gr=Rate(.05)
   ...: )
   ...:
```

(continues on next page)

(continued from previous page)

```
In [5]: print(ann.pv())
16862.766126498827
```

How much is this annuity worth at time 3?

```
In [6]: print(ann.eq_val(3))
19520.75963718821
```

3.3.6 Nonlevel Annuities - Geometric Progression

Annuities can have payments that increase geometrically. For example, an annuity might have payments that increase by 2% per year. If we have payments that increase by g% per year, we define the present value of an annuity-immediate with an initial payment P as:

$$P\left(\frac{1-\left(\frac{1+g}{1+i}\right)^n}{i-g}\right),$$

where $i - g \neq 0$, since this expression is undefined when the denominator is 0. If the payments increase at the rate of interest, we have:

$$nP(1+i)^{-1}.$$

Examples

Suppose we have an annuity-immediate with end-of-year payments that pays 1 at the end of the first period, and then whose payments increase by 2% for each year for the next 4 years. If the interest rate is 5% compounded annually, what is its present value?

We can solve this problem by using TmVal's *Annuity* class, and by providing the rate of payment increase to the argument `gprog`, which in this case is `gprog=.02`:

```
In [1]: from tmval import Annuity, Rate

In [2]: ann = Annuity(
   ...:     gr=Rate(.05),
   ...:     n=5,
   ...:     gprog=(.02)
   ...: )
   ...:

In [3]: print(ann.pv())
4.497460026576225
```

3.3.7 Nonlevel Annuities - Arithmetic Progression

TmVal's *Annuity* class can also handle annuities with increasing arithmetic progression. This means that if an annuity makes an initial payment P, the next payment is $P+Q$, and the payment after that is $P+2Q$, and so on. The present value of such an annuity is:

$$(I_{P,Q}a)\overline{n}|i = Pa_{\overline{n}|i} + \frac{Q}{i}(a_{\overline{n}|i} - nv^n).$$

The accumulated value is:

$$(I_{P,Q}s)\overline{n}|i = Ps_{\overline{n}|i} + \frac{Q}{i}(s_{\overline{n}|i} - n).$$

The *Annuity* class can also handle the companion formulas for the annuity-due case:

$$(I_{P,Q}\ddot{s})\overline{n}|i = P\ddot{s}_{\overline{n}|i} + \frac{Q}{d}(s_{\overline{n}|i} - n),$$

and

$$(I_{P,Q}\ddot{a})\overline{n}|i = P\ddot{a}_{\overline{n}|i} + \frac{Q}{d}(a_{\overline{n}|i} - n).$$

For special cases $(Is)\overline{n}|i$, $(Ia)\overline{n}|i$, $(Da)\overline{n}|i$, $(I\ddot{s})\overline{n}|i$, $(D\ddot{a})\overline{n}|i$, see the *Notation Guide*.

Examples

Suppose we have a 10-year annuity with an initial end-of-year payment of 100, and subsequent end-of-year payments increasing by 100 for each of the next 9 years. If the interest is 5% compounded annually, what is the present value?

We can solve this problem by setting the `aprog` argument of the *Annuity* class to `aprog=100`:

```
In [1]: from tmval import Annuity

In [2]: ann = Annuity(
   ...:     gr=.05,
   ...:     amount=100,
   ...:     n=10,
   ...:     aprog=100
   ...: )
   ...:

In [3]: print(ann.pv())
3937.378280472917
```

Now, suppose instead that the annuity makes beginning-of-year payments. What is the accumulated value?

```
In [4]: ann2 = Annuity(
   ...:     gr=.05,
   ...:     amount=100,
   ...:     n=10,
   ...:     aprog=100,
   ...: )
   ...:

In [5]: print(ann2.sv())
6413.57432465254
```

The special cases mentioned earlier can be achieved by simply modifying the `amount` and `aprog` argument to be equal to the corresponding special case values. For more information on what these symbols mean and how to derive them, refer to a text on interest theory (some can be found in the *References* section).

3.3.8 Perpetuities - Arithmetic Progression

TmVal's *Annuity* class can also handle perpetuities with payments of increasing arithmetic progression:

$$(I_{P,Q}a)\overline{\infty}|i = Pa_{\overline{\infty}|i} + \frac{Q}{i}\overline{\infty}|i = \frac{P}{i} + \frac{Q}{i^2}$$

$$(I_{P,Q}\ddot{a})\overline{\infty}|i = P\ddot{a}_{\overline{\infty}|i} + \frac{Q}{i}\overline{\infty}|i = \frac{P}{d} + \frac{Q}{id}$$

Examples

Suppose we have a perpetuity-immediate with an initial end-of-year payment of 100. Subsequent end-of-year payments increase 100 each year forever. If the interest rate is 5% compounded annually, what's the present value?

To solve this problem, we need the special value `np.Inf` from NumPy to specify an infinite term, passing `term=np.Inf` to TmVal's *Annuity* class.

```
In [1]: import numpy as np

In [2]: from tmval import Annuity

In [3]: ann = Annuity(
   ...:     amount=100,
   ...:     gr=.05,
   ...:     term=np.Inf,
   ...:     aprog=100
   ...: )
   ...:

In [4]: print(ann.pv())
41999.99999999993
```

What if we have a perpetuity-due instead?

```
In [5]: ann2 = Annuity(
   ...:     amount=100,
   ...:     gr=.05,
   ...:     term=np.Inf,
   ...:     aprog=100,
   ...:     imd='due'
   ...: )
   ...:

In [6]: print(ann2.pv())
44099.99999999993
```

3.3.9 Nonintegral Terms

Sometimes, certain payment structures will not yield an integral number of payments for a desired present value. For example, suppose we were to take out a loan for 10,000 at 5% interest compounded annually. If we were to pay 1,000 per year to settle the loan, solving the equation $L = Qa_{\overline{r}|i}$ for r will not result in an integer. Specifically, the formula for solving r is:

$$r = -\frac{\ln\left(1 - \frac{iL}{Q}\right)}{\ln(1+i)},$$

and for our example, $r \approx 14.207$ years.

This is problematic if we want the final payment to coincide with the payment period. Two ways to adjust for this are to have the final payment occur on the next or previous integral period. For example, in our example, we can have the final payment occur at time 15 or time 14. We then adjust the final payment for the appropriate time value of money so that the present value of payments equals the present value of the loan.

If the payment is rolled forward to the next period, this payment is called the *drop payment*. When rolled backwards and added to the previous payment, it is called the *balloon payment*.

The drop payment is equal to:

$$Q\left(\frac{(1+i)^f - 1}{i}\right)(1+i)^{1-f},$$

and the balloon payment is equal to:

$$Q + Q\left(\frac{(1+i)^f - 1}{i}\right)v^f.$$

Example

Suppose we borrow 10,000 at 5% compound annual interest, and repay it by making end-of-year payments of 1,000. If we elect to have the final payment be a drop payment, how much is the payment, and what time does it occur on?

To solve this problem, we can set the `loan` and `drb` arguments of the *Annuity* class to their appropriate values. The `drb` argument specifies whether we want a drop or balloon payment. We can then check the drop payment using the `drb_pmt` attribute.

To check the time of the last payment, we can use the `times` attribute of the *Annuity* object and pass it to the `max` function to get the result.

```
In [1]: from tmval import Annuity

In [2]: ann = Annuity(
   ...:     loan=10000,
   ...:     amount=1000,
   ...:     gr=.05,
   ...:     drb='drop'
   ...: )
   ...:

# drop payment amount
In [3]: print(ann.drb_pmt)
210.71820588633327

# drop payment time
In [4]: print(max(ann.times))
15
```

To get confidence, let's confirm that the present value is equal to 10,000:

```
In [5]: print(ann.pv())
10000.0
```

What if we elect the final payment to be a balloon payment? What is the amount, and when does it occur?

```
In [6]: ann2 = Annuity(
   ...:     loan=10000,
   ...:     amount=1000,
   ...:     gr=.05,
   ...:     drb='balloon'
   ...: )
   ...:

# drop payment amount
In [7]: print(ann2.drb_pmt)
1200.6840056060316

# drop payment time
In [8]: print(max(ann2.times))
14

# again, check the pv
In [9]: print(ann2.pv())
10000.000000000002
```

3.3.10 General Accumulation Functions

The annuities supported by TmVal need not be limited to the main growth patterns supported by the `Rate` class. TmVal can also handle annuities governed by more complex growth patterns, such as polynomial growth.

Examples

Suppose we have a basic, 5-year term annuity-immediate governed by the following growth pattern:

$$a(t) = .05t^2 + .02t + .03$$

What is the present value of the annuity?

We can solve this problem by defining a function for the quadratic growth pattern and then passing it to the `Accumulation` class, which in turn is passed to the `Annuity` class:

```
In [1]: from tmval import Accumulation, Annuity

In [2]: def f(t):
   ...:     return .05 * t ** 2 + .02 * t + 1
   ...:

In [3]: ann = Annuity(gr=Accumulation(gr=f), n=5)

In [4]: print(ann.pv())
3.36072951629534
```

What is the accumulated value?

```
In [5]: print(ann.sv())
7.897714363294049
```

3.4 Loans

TmVal provides a `Loan` class, which can be used to calculate the outstanding balance on any date over the term of the loan.

3.4.1 Outstanding Loan Balance - Retrospective Method

A common problem involving loans is calculating the outstanding loan balance at a point in time. One way to do this is called the *retrospective method*, which first calculates the accumulated value of the principal, and then subtracts the accumulated value of the payments. The formula for the outstanding loan balance after the k-th payment is thus:

$$\text{OLB}_k = La(k) - Qs_{\overline{k}|}$$

The main advantage of the retrospective method is that we do not need to know the term or the total number of payments required to settle the loan.

Examples

Suppose we have borrowed 50,000 to be paid off with annual end-of-year payments of 5,000. If the interest rate is 5% compounded annually, what is the outstanding loan balance immediately after the 5th payment?

We can solve this problem with TmVal's `Loan` class, which is its main class for performing loan calculations. First, we need to define the loan by setting the arguments for the loan amount, `amt=5000`, the payment amount, `pmt=5000`, the payment period, `period=1`, and the interest rate.

We can call the method `Loan.olb_r()` to apply the retrospective method to find the balance at time `t=5`:

```
In [1]: from tmval import Loan, Rate

In [2]: my_loan = Loan(
   ...:     amt=50000,
   ...:     pmt=5000,
   ...:     period=1,
   ...:     gr=Rate(.05),
   ...: )
   ...:

In [3]: print(my_loan.olb_r(t=5))
36185.921875
```

3.4.2 Outstanding Loan Balance - Prospective Method

Another way to calculate the outstanding loan balance at a point in time is the *prospective method*, which sums up the value of the remaining loan payments, discounted to that time period. Assuming compound interest, if the last payment is adjusted to avoid over/under payment, the outstanding loan balance calculated via the prospective method is defined as:

$$\text{OLB}_k = Qa_{\overline{n-k-1}|i} + R(1+i)^{-(n-k)}.$$

If all the payments are equal, it is defined as:

$$\text{OLB}_k = Qa_{\overline{n-k}|i}.$$

The advantage of the prospective method is that we do not need to know the original loan amount.

Examples

Suppose we need to make 10 end-of-year payments of 5,000 to pay off a loan. Assuming the rate of interest is 5% compounded annually, what is the outstanding loan balance immediately after the 5th payment?

We can solve this by using TmVal's `Loan` class. First, we define the loan by providing the characteristics in the preceding paragraph. Then, we call the method `Loan.olb_p()` to execute the prospective method:

```
In [1]: from tmval import Loan, Rate

In [2]: my_loan = Loan(
   ...:     pmt=5000,
   ...:     gr=Rate(.05),
   ...:     period=1,
   ...:     term=10
   ...: )
   ...:

In [3]: print(my_loan.olb_p(t=5))
21647.383353154095
```

Now, suppose we miss the 4th and 5th payment. What is the outstanding loan balance at time 5?

We can solve this by again calling the `Loan.olb_p()`, and supplying the 4th and 5th payments as a list to the missing argument, `missing=[4, 5]`:

```
In [4]: print(my_loan.olb_p(t=5, missed=[4, 5]))
31897.383353154095
```

3.5 Bonds

A *bond* is a financial instrument that entities can use to borrow money. An entity, called a bond issuer, can issue bonds to parties, called bondholders, who are willing to lend it money. The bond itself is a promise from the issuer to pay the bondholders a stream of one more payments on specified dates in exchange for the amount borrowed, which is called the initial bond price.

TmVal's `Bond` class is used to define bonds. You can initialize a bond by supplying certain characteristics about the bonds to the bond constructor:

1. Bond price
2. Coupons
3. Yield rate
4. Redemption amount
5. Face value
6. Term

The bond price is the amount a bondholder pays to acquire a bond. If purchased from the issuer, this also represents the amount loaned to the issuer. Otherwise, it refers to the amount one pays to another bondholder for the rights to the remaining stream of payments promised by the bond.

Coupons are regular payments made by the bond issuer to the bondholder. A bond may have zero coupons, in which case it is referred to as a *zero-coupon* bond.

The yield rate on a bond is the yield rate to the bondholder if the bond is held to maturity.

The redemption amount an amount paid on the due date of a bond, not including any coupons.

The face value is an amount used to calculate the coupon amount of a bond. Coupons are often referred to as being a certain % of the face value.

The term is the time between when the bond is purchased and its due date, after which the bond is considered to be repaid.

You do not have to specify all of these characteristics to initialize a bond in TmVal. Common bond problems involve those where not all the aforementioned quantities are known, and TmVal can be used to solve for these missing quantities.

This section explains how you can use TmVal to define bonds as well as the calculations you can perform on them.

3.5.1 Zero-Coupon Bonds

Zero-coupon bonds are bonds that do not pay any coupons. They are the simplest type of bond, in which a bond issuer sells the bond to a bondholder. The price paid for the bond is the amount of money loaned the bond issuer from the bondholder. At the end of a specified amount of time, the bond issuer pays a redemption amount to the bondholder to settle the debt.

Examples

Suppose an entity issues a bond for 1,000 to be repaid in 5 years for 1,200. What is the yield rate on the bond?

Since we know the price, redemption amount, and term, we can supply these to the *Bond* class arguments `price`, `red`, and `term` respectively. The *Bond* class constructor will automatically detect which argument is missing and solve for it. We can then find the missing value (in this case the yield rate) by printing the attribute.

```
In [1]: from tmval import Bond

In [2]: bd = Bond(
   ...:     price=1000,
   ...:     red=1200,
   ...:     term=5
   ...: )
   ...:
[-1000, 1200]

In [3]: print(bd.irr())
[0.03713728933664773]
```

To further illustrate how *Bond* can solve for missing values, here are some more examples with various missing values.

If an entity issues a bond for 1,000 to be repaid in 5 years with a yield of 5% compounded annually, what should the redemption amount be?

```
In [4]: bd = Bond(
   ...:     price=1000,
   ...:     gr=.05,
   ...:     term=5
   ...: )
   ...:

In [5]: print(bd.red)
1276.2815625000003
```

If an entity issues a bond redeemable for 1,200 in 5 years at a yield rate of 5% compounded annually, what is the price of the bond?

```
In [6]: bd = Bond(
   ...:     red=1200,
   ...:     gr=.05,
   ...:     term=5
   ...: )
   ...:

In [7]: print(bd.price)
940.2313997621505
```

If an entity issues a bond for a price of 1,000, to be redeemed for 1,200 at a yield rate of 5% compounded annually, what is the term of the bond?

```
In [8]: bd = Bond(
   ...:    price=1000,
   ...:    red=1200,
   ...:    gr=.05
   ...: )
   ...:

In [9]: print(bd.term)
3.73685
```

3.5.2 Coupon Bonds

Coupon bonds are bonds in which the bond issuer makes payments to the bondholder in addition to the redemption amount throughout the life of the bond.

Coupons are calculated specifying a face value F, a coupon rate α, and the coupon frequency m. The relationship between these values can be described by the following equation:

$$Fr = \frac{F\alpha}{m}.$$

When the face value equals the redemption amount, $F = C$, we refer to the bond as being a *par-value bond*.

Sometimes we'd like to specify the coupon rate as a *modified coupon rate* g in terms of the bond redemption amount C:

$$g = \frac{Fr}{C}.$$

We can calculate the number of coupons n by multiplying the bond term N in years by the coupon frequency in number of coupons per year m:

$$n = Nm.$$

We are often interested in the bond's yield to the investor, or bondholder, as a rate compounded yearly i, or as a rate j, specified in terms of the coupon period:

$$j = \frac{I}{m}$$

where

$$i = \left(1 + \frac{I}{m}\right)^m - 1.$$

The price P of a bond is the present value of the coupons and the redemption amount. Using this information, we arrive at the basic price formula for a bond:

$$P = (Fr)a_{\overline{n|}j} + Cv_j^n$$

Another property of a bond is called the base amount G, which is the present value of a perpetuity of the coupon payments:

$$G = \frac{Fr}{j}$$

We can express the coupon amount using the face amount, redemption value, or base amount:

$$Fr = Cg = Gj$$

Examples

Suppose we have a 5-year 1,000 5% bond with semiannual coupons, redeemable at par. What is the price of the bond if the yield is 10% compounded annually?

We can use TmVal to solve for the price just like we can with a zero coupon bond, but this time we need to supply the coupon information because this bond pays coupons. We can set the arguments `alpha=.05` and `cfreq=4` to represent the coupon rate and frequency, respectively. We also need to set the face value, `face=1000`:

```
In [1]: from tmval import Bond

In [2]: bd = Bond(
   ...:     face=1000,
   ...:     red=1000,
   ...:     alpha=.05,
   ...:     cfreq=4,
   ...:     term=5,
   ...:     gr=.10
   ...: )
   ...:

In [3]: print(bd.price)
817.4272763857732
```

Alternatively, we can specify the coupon rate and frequency to the argument `cgr`. Since this information is equivalent to a nominal interest rate, you can supply it to `cgr` instead of using `alpha` and `cfreq`:

```
In [4]: from tmval import Bond, Rate

In [5]: cgr = Rate(
   ...:    rate=.05,
   ...:    pattern="Nominal Interest",
   ...:    freq=4
   ...: )
   ...:

In [6]: bd = Bond(
   ...:     face=1000,
   ...:     red=1000,
   ...:     cgr=cgr,
   ...:     term=5,
   ...:     gr=.10
```

(continues on next page)

(continued from previous page)

```
   ...: )
   ...:

In [7]: print(bd.price)
817.4272763857732
```

Now, let’s examine various properties of this bond:

```
# coupon amount
In [8]: print(bd.fr)
12.5

# number of coupons
In [9]: print(bd.n_coupons)
20

# base amount
In [10]: print(bd.g)
0.0125

# coupon payments
In [11]: print(type(bd.coupons))
<class 'tmval.annuity.Annuity'>

In [12]: print(bd.coupons.amounts)
[12.5, 12.5, 12.5, 12.5, 12.5, 12.5, 12.5, 12.5, 12.5, 12.5, 12.5, 12.5, 12.5, 12.5,␣
↪12.5, 12.5, 12.5, 12.5, 12.5, 12.5]

In [13]: print(bd.coupons.times)
[0.25, 0.5, 0.75, 1.0, 1.25, 1.5, 1.75, 2.0, 2.25, 2.5, 2.75, 3.0, 3.25, 3.5, 3.75, 4.
↪0, 4.25, 4.5, 4.75, 5.0]

# yield per coupon period
In [14]: print(bd.j)
0.02411368908444511
```

Let’s use this information to verify $Fr = Cg = Gj$:

```
# Fr
In [15]: fr = bd.fr

In [16]: print(bd.fr)
12.5

# Cg
In [17]: cg = bd.g * bd.red

In [18]: print(cg)
12.5

# Gj
In [19]: gj = bd.base * bd.j

In [20]: print(gj)
12.5

In [21]: print(fr == cg == gj)
```

(continues on next page)

(continued from previous page)

```
True
```

3.5.3 Nonlevel Coupons

TmVal can handle nonlevel coupons as well. This feature can be found in some bonds that offer increasing coupon payments to hedge against inflation.

Examples

Suppose we have a five year 1,000 bond with a redemption amount of 1,250. Annual coupons start at 5% and then increase by 2% each year. If the bond is priced to yield 5% compounded annually, what is the price of the bond?

We can solve this problem with TmVal by providing a list of tuples to the `alpha` argument of the *Bond* class. Each tuple in the list is an ordered pair that represents a coupon rate and the beginning of the time period in which it is applicable.

In this case, we want [(.05, 0), (.051, 1), (.05202, 2), (.0530604,3), (.054121608, 4)].

We also need to supply a corresponding list to the `cfreq` argument of equal length to the list provided to `alpha`. Since that list has 5 elements, and for each of those periods we have one coupon payment, we set `cfreq=[1,1,1,1,1]`. While this particular example has all 1s, this feature allows us to model more complex bonds where the coupon frequency is not the same for each period.

```
In [1]: from tmval import Bond

In [2]: t = 5

In [3]: alpha_r = [.05 * 1.02 ** x for x in range(t)]

In [4]: alpha_t = [x for x in range(t)]

In [5]: alphas = [(x, y) for x, y in zip(alpha_r, alpha_t)]

In [6]: bd = Bond(
   ...:     face=1000,
   ...:     red=1250,
   ...:     alpha=alphas,
   ...:     cfreq=[1] * 5,
   ...:     gr=.05,
   ...:     term=t
   ...: )
   ...:

# coupon amounts
In [7]: print(bd.coupons.amounts)
[50.0, 51.00000000000001, 52.02, 53.06040000000001, 54.121608]

# coupon times
In [8]: print(bd.coupons.times)
[1.0, 2.0, 3.0, 4.0, 5.0]

In [9]: print(bd.price)
1204.280709414385
```

3.5.4 Bond Premium and Discount

The bond notation can be used to rearrange the basic price formula to arrive at the premium-discount formula:

$$P = C(g - j)a_{\overline{n}|j} + C$$

A bond is said to sell at a premium if the price P exceeds the redemption amount C. Equivalently this is also the case when the modified coupon rate exceeds the yield rate per coupon period, when $g > j$:

$$\text{premium} = P - C = C(g - j)a_{\overline{n}|j}.$$

A bond is said to sell at a discount if the price P is less than the redemption amount C. Equivalently, this is also the case when the yield rate per coupon period exceeds the modified coupon rate, when $j > g$:

$$\text{discount} = C - P = C(j - g)a_{\overline{n}|j}.$$

Examples

Suppose we have a 5-year, 1,000 5% par value bond with annual coupons, priced to yield 8% compounded annually. Find out if the bond sells at a premium or discount, and compute the magnitude of premium or discount.

TmVal's *Bond* class has an attribute called `premium` that represents the magnitude of the premium or discount. It is positive if the bond sells at a premium and negative if it sells at a discount. Since $j > g$ for this example, we would expect it to sell at a discount:

```
In [1]: from tmval import Bond

In [2]: bd = Bond(
   ...:     face=1000,
   ...:     red=1000,
   ...:     alpha=.05,
   ...:     cfreq=1,
   ...:     term=5,
   ...:     gr=.08
   ...: )
   ...:

In [3]: print(bd.price)
880.2186988876571

In [4]: print(bd.price < bd.red)
True

In [5]: print(bd.premium)
-119.78130111234293
```

Now, to see that the bond sells at a premium when $g > j$, let's switch the yield and coupon percentages:

```
In [6]: from tmval import Bond

In [7]: bd = Bond(
   ...:     face=1000,
   ...:     red=1000,
   ...:     alpha=.08,
   ...:     cfreq=1,
   ...:     term=5,
```

(continues on next page)

(continued from previous page)

```
   ...:     gr=.05
   ...: )
   ...:

In [8]: print(bd.price)
1129.8843001189243

In [9]: print(bd.price > bd.red)
True

In [10]: print(bd.premium)
129.88430011892433
```

3.5.5 Makeham's Formula

Makeham's formula can also be derived via bond notation and can be useful for obtaining the price of a bond when we do not know the number of coupons:

$$P = \frac{g}{j}(C - K) + K,$$

where K is the present value of the redemption amount.

Examples

Suppose we have a 1,000 bond that pays 5% annual coupons. The redemption amount is 1,250 and the present value of the redemption amount is 776.1516538. If it is priced to yield 10% compounded annually, what is the price of the bond? Also, find the term and number of coupons.

TmVal's *Bond* class has a method called *makeham()* which it calls internally when the term and price are missing, and when the present value of the redemption amount is provided via the `k` argument. Because it is called internally, we do not have to explicitly call this method to get the bond price.

```
In [1]: from tmval import Bond

In [2]: bd = Bond(
   ...:     face=1000,
   ...:     red=1250,
   ...:     k=776.1516538,
   ...:     alpha=.05,
   ...:     cfreq=1,
   ...:     gr=.10
   ...: )
   ...:

In [3]: print(bd.term)
5.0

In [4]: print(bd.n_coupons)
5.0

In [5]: print(bd.price)
965.690992294366
```

We could call *makeham()* anyway, just to verify:

```
In [6]: print(bd.makeham())
965.690992294366
```

3.6 Pandas Compatibility

CHAPTER

FOUR

API REFERENCE

4.1 General Functions

This section documents standalone functions that do not belong to a class.

4.1.1 tmval.simple_solver

`tmval.growth.`**`simple_solver`**(*pv: Optional[float] = None*, *fv: Optional[float] = None*, *gr: Union[float, tmval.rate.Rate, None] = None*, *t: Optional[float] = None*, *rate_res='Simple Interest'*)

Simple interest solver for when one variable is missing - returns missing value. You need to supply three out of the four arguments, and the function will solve for the missing one.

Parameters

- **`pv`** (*`float`*) – the present value
- **`fv`** (*`float`*) – the future value
- **`gr`** (*`float`*, `Rate`) – the interest rate
- **`t`** (*`float`*) – the time
- **`rate_res`** (*`str`*) – if solving for a rate, whether you want simple discount or interest

Returns the present value, future value, interest rate, or time - whichever is missing.

Return type float, *Rate*

4.1.2 tmval.osi

`tmval.growth.`**`osi`**(*beg_dt: datetime.datetime*, *end_dt: datetime.datetime*, *frac=True*) → float

Calculate the number of days using the ordinary simple interest or 30/360 rule. Set frac=True to return days as a percentage of year.

Parameters

- **`beg_dt`** (*`datetime.datetime`*) – beginning date
- **`end_dt`** (*`datetime.datetime`*) – ending date
- **`frac`** (*`bool`*, *`optional`*) – whether you want the answer in number of days or fraction of a year, defaults to True

Returns the number of days using the ordinary simple interest or 360 rule, or the percentage of year if frac=True

Return type float

4.1.3 tmval.bankers_rule

`tmval.growth.`**`bankers_rule`**(*beg_dt: datetime.datetime*, *end_dt: datetime.datetime*, *frac=True*) → float

Calculate the number of days using the Banker's rule or actual/360 rule. Set frac=True to return days as a percentage of year.

Parameters

- **`beg_dt`** (*`datetime.datetime`*) – the beginning date
- **`end_dt`** (*`datetime.datetime`*) – the ending date
- **`frac`** (*`bool, optional`*) – whether you want the answer in number of days or fraction of a year, defaults to True

Returns the number of days or percent of a year between two dates using the Banker's rule or actual/360 rule, depending on frac

Return type float

4.1.4 tmval.compound_solver

`tmval.growth.`**`compound_solver`**(*pv: Optional[float] = None*, *fv: Optional[float] = None*, *t: Optional[float] = None*, *gr: Union[float, tmval.rate.Rate, None] = None*)

Solves for a missing value in the case of compound interest supply 3/4 of: a present value, a future value an interest rate (either APY or APR), and a time. If using an APR interest rate, you need to supply a compounding frequency of m times per period, and you need to set the use_apr parameter to True.

Parameters

- **`pv`** (*`float`*) – the present value, defaults to None.
- **`fv`** (*`float`*) – the future value, defaults to None.
- **`t`** (*`float`*) – the time, defaults to None.

Returns either a present value, a future value, an interest rate, or a time, depending on which one is missing.

Return type float

4.1.5 tmval.k_solver

`tmval.growth.`**`k_solver`**(*f: Callable*, *fv: float*, *t: float*) → float

Solver to get the initial investment K, given a growth pattern and future value.

Parameters

- **`f`** (*`Callable`*) – the growth pattern.
- **`fv`** (*`float`*) – the future value.
- **`t`** (*`float`*) – the time.

Returns the initial investment, K

Return type float

4.1.6 tmval.interest_from_discount

`tmval.conversions.`**`interest_from_discount`**(*d: float*) → float

An interest/discount rate converter. Returns the interest rate, given the discount rate.

Parameters **`d`** (*`float`*) – the discount rate.

Returns the interest rate.

Return type float

4.1.7 tmval.discount_from_interest

`tmval.conversions.`**`discount_from_interest`**(*i: float*) → float

An interest/discount rate converter. Returns the discount rate, given the interest rate.

Parameters **`i`** (*`float`*) – the interest rate.

Returns the discount rate.

Return type float

4.1.8 tmval.effective_from_nominal_int

`tmval.conversions.`**`effective_from_nominal_int`**(*im: float*, *m: float*) → float

A nominal/effective interest rate converter. Given a `NominalInt` object, returns the effective interest rate. You may also supply the nominal interest rate and compounding frequency, but not when a `NominalInt` is already supplied and vice-versa.

Parameters

- **`im`** (*`float`*) – the nominal interest rate.
- **`m`** (*`float`*) – the compounding frequency.

Returns the effective interest rate.

Return type float

4.1.9 tmval.effective_from_nominal_disc

`tmval.conversions.`**`effective_from_nominal_disc`**(*dm: float*, *m: float*)

A nominal/effective discount rate converter. Given a `NominalDisc` object, returns the effective discount rate. You may also supply the nominal discount rate and compounding frequency, but not when a `NominalDisc` is already supplied and vice-versa.

Parameters

- **`dm`** (*`float`*) – the nominal discount rate.
- **`m`** (*`float`*) – the compounding frequency.

Returns the effective discount rate

Return type float

4.1.10 tmval.apy

`tmval.conversions.apy`(*im: float*, *m: float*) → float

An alias for `effective_from_nominal()`. Returns annual percentage yield, or annual effective yield (APY), given a nominal rate of interest and compounding frequency.

Parameters

- **im** (*float*) – the nominal rate of interest
- **m** (*float*) – the compounding frequency, i.e., compounded m times per year

Returns the annual percentage yield, or annual effective yield (APY)

Return type float

4.1.11 tmval.apr

`tmval.conversions.apr`(*i: float*, *m: float*) → tmval.conversions.RateTemplate

An alias for `nominal_from_eff()`. Returns annual percentage rate, or nominal interest rate (APR), given an effective rate of interest and compounding frequency of the desired nominal rate.

Parameters

- **i** (*float*) – the effective rate of interest
- **m** (*float*) – the desired compounding frequency, i.e., compounded m times per year

Returns the annual percentage rate, or nominal rate of interest (APR)

Return type float

4.1.12 tmval.eff_int_from_eff_int

`tmval.conversions.eff_int_from_eff_int`(*i: float*, *old_t: float*, *new_t: float*) → tmval.conversions.RateTemplate

Parameters

- **i** (*float*) – the effective interest rate.
- **old_t** (*float*) – the old unit of time.
- **new_t** (*float*) – the new unit of time.

Returns a converted interest rate, based off the new unit of time.

Return type RateTemplate

4.1.13 tmval.nom_int_from_eff_int

`tmval.conversions.nom_int_from_eff_int`(*i: float*, *new_m: float*, *old_t: float = None*) → tmval.conversions.RateTemplate

A nominal/effective interest/discount rate converter. Given an effective interest rate and desired compounding frequency, returns the nominal interest rate.

Parameters

- **i** (*float*) – the effective interest rate.
- **new_m** (*float,*) – the desired compounding frequency.

- **old_t** (*float, optional*) – the interval of the effective interest rate. Assumed to be 1 if not provided.

Returns the nominal interest rate.

Return type RateTemplate

4.1.14 tmval.eff_disc_from_eff_int

`tmval.conversions.`**`eff_disc_from_eff_int`**(*i: float*, *old_t: float = None*, *new_t: float = None*) → tmval.conversions.RateTemplate

A nominal/effective interest/discount rate converter. Given an effective interest rate and applicable unit of time, along with a desired unit of time for the new rate, returns an effective discount rate at the new unit of time. If old_t or new_t are not provided, they are each assumed to be 1, i.e., a 1-year period.

Parameters

- **i** (*float*) – the effective interest rate
- **old_t** (*float, optional*) – the old unit of time, defaults to None.
- **new_t** (*float, optional*) – the new unit of time, defaults to None.

Returns an effective discount rate at the new unit of time.

Return type RateTemplate

4.1.15 tmval.nom_disc_from_eff_int

`tmval.conversions.`**`nom_disc_from_eff_int`**(*i: float*, *new_m: float*, *old_t: float = None*) → tmval.conversions.RateTemplate

A nominal/effective interest/discount rate converter. Given an effective interest rate and unit of time, along with a desired compounding frequency, returns a nominal discount rate at the desired compounding frequency. If no unit of time (applicable to the effective interest rate) is provided, it is assumed to be 1, i.e., a 1-year period.

Parameters

- **i** (*float*) – the effective interest rate.
- **new_m** (*float*) – the desired compounding frequency.
- **old_t** (*float, optional*) – the unit of time applicable to the effective interest rate, defaults to None.

Returns a nominal discount rate, compounded new_m times per year.

Return type RateTemplate

4.1.16 tmval.eff_int_from_eff_disc

`tmval.conversions.`**`eff_int_from_eff_disc`**(*d: float*, *old_t: float = None*, *new_t: float = None*) → tmval.conversions.RateTemplate

A nominal/effective interest/discount rate converter. Given an effective discount rate and unit of time, along with a desired new unit of time, returns an effective interest rate at the new unit of time. If no unit of time (either old or new) is provided, it is assumed to be 1, i.e., a 1-year period.

Parameters

- **d** (*float*) – the effective discount rate.

- **old_t** (*float, optional*) – the old unit of time, defaults to None.
- **new_t** (*float, optional*) – the new unit of time, defaults to None.

Returns an effective interest rate, at the new unit of time.

Return type RateTemplate

4.1.17 tmval.eff_disc_from_eff_disc

`tmval.conversions.`**`eff_disc_from_eff_disc`**(*d: float*, *old_t: float*, *new_t: float*) → tmval.conversions.RateTemplate

A nominal/effective interest/discount rate converter. Given an effective discount rate and unit of time, along with a desired new unit of time, returns an effective discount rate at the desired new unit of time. period.

Parameters

- **d** (*float*) – the effective discount rate.
- **old_t** (*float*) – the old unit of time.
- **new_t** (*float*) – the new unit of time.

Returns an effective discount rate at the new unit of time.

Return type RateTemplate

4.1.18 tmval.nom_int_from_eff_disc

`tmval.conversions.`**`nom_int_from_eff_disc`**(*d: float*, *new_m: float*, *old_t: float = None*) → tmval.conversions.RateTemplate

A nominal/effective interest/discount rate converter. Given an effective discount rate and unit of time, along with a desired compounding frequency, returns a nominal interest rate at the desired compounding frequency. If no unit of time is provided (applicable to the effective discount rate), it is assumed to be 1, i.e., a 1-year period.

Parameters

- **d** (*float*) – the effective discount rate.
- **new_m** (*float*) – the desired compounding frequency.
- **old_t** (*float, optional*) – the old unit of time, defaults to None.

Returns a nominal interest rate, compounded new_m times per year

Return type RateTemplate

4.1.19 tmval.nom_disc_from_eff_disc

`tmval.conversions.`**`nom_disc_from_eff_disc`**(*d: float*, *new_m: float*, *old_t: float = None*) → tmval.conversions.RateTemplate

A nominal/effective interest/discount rate converter. Given an effective discount rate and unit of time, along with a desired compounding frequency, returns a nominal discount rate at the desired compounding frequency. If no unit of time is provided (applicable to the effective discount rate), it is assumed to be 1, i.e., a 1-year period.

Parameters

- **d** (*float*) – the effective discount rate.
- **new_m** (*float*) – the desired compounding frequency.

- **old_t** (*float, optional*) – the old unit of time, defaults to None.

Returns a nominal discount rate, compounded new_m times per year.

Return type RateTemplate

4.1.20 tmval.eff_int_from_nom_int

`tmval.conversions.`**`eff_int_from_nom_int`**(*im: float*, *m: float*, *new_t: float = None*) → tmval.conversions.RateTemplate

A nominal/effective interest/discount rate converter. Given a nominal interest rate, along with a desired unit of time, returns an effective interest rate at the desired unit of time. If no unit of time is provided, it is assumed to be 1, i.e., a 1-year period.

Parameters

- **im** (*float*) – the nominal interest rate.
- **m** (*float*) – the compounding frequency.
- **new_t** (*float, optional*) – the desired unit of time, defaults to None.

Returns an effective interest rate.

Return type RateTemplate

4.1.21 tmval.nom_int_from_nom_int

`tmval.conversions.`**`nom_int_from_nom_int`**(*im: float*, *m: float*, *new_m: float*) → tmval.conversions.RateTemplate

A nominal/effective interest/discount rate converter. Given a nominal interest rate, along with a desired compounding frequency, returns a nominal interest rate at the desired compounding frequency.

Parameters

- **im** (*float*) – the nominal interest rate.
- **m** (*float*) – the compounding frequency
- **new_m** (*float*) – the desired compounding frequency.

Returns a nominal interest rate, compounded new_m times per year.

Return type RateTemplate

4.1.22 tmval.eff_disc_from_nom_int

`tmval.conversions.`**`eff_disc_from_nom_int`**(*im: float*, *m: float*, *new_t: float = None*) → tmval.conversions.RateTemplate

A nominal/effective interest/discount rate converter. Given a nominal interest rate, along with a desired unit of time, returns an effective discount rate at the desired unit of time. If no unit of time is provided, it is assumed to be 1, i.e., a 1-year period.

Parameters

- **im** (*float*) – the nominal interest rate.
- **m** (*float*) – the compounding frequency
- **new_t** (*float, optional*) – the desired unit of time.

Returns an effective discount rate.

Return type RateTemplate

4.1.23 tmval.nom_disc_from_nom_int

`tmval.conversions.`**`nom_disc_from_nom_int`**(*im: float*, *m: float*, *new_m: float = None*) → tmval.conversions.RateTemplate

A nominal/effective interest/discount rate converter. Given a nominal interest rate, along with a desired compounding frequency, returns an nominal discount rate at the desired compounding frequency.

Parameters

- **im** (*float*) – the nominal interest rate.
- **m** (*float*) – the compounding frequency
- **new_m** (*float*) – the desired compounding frequency.

Returns a nominal discount rate, compounded new_m times per year

Return type RateTemplate

4.1.24 tmval.eff_int_from_nom_disc

`tmval.conversions.`**`eff_int_from_nom_disc`**(*dm: float*, *m: float*, *new_t: float = None*) → tmval.conversions.RateTemplate

A nominal/effective interest/discount rate converter. Given a nominal discount rate, along with a desired unit of time, returns an effective interest rate at the desired unit of time. If no unit of time is provided, it is assumed to be 1, i.e., a 1-year period.

Parameters

- **dm** (*float*) – the nominal discount rate.
- **m** (*float*) – the compounding frequency.
- **new_t** (*float*) – the desired unit of time, defaults to None.

Returns an effective interest rate.

Return type RateTemplate

4.1.25 tmval.nom_int_from_nom_disc

`tmval.conversions.`**`nom_int_from_nom_disc`**(*dm: float*, *m: float*, *new_m: float = None*) → tmval.conversions.RateTemplate

A nominal/effective interest/discount rate converter. Given a nominal discount rate, along with a desired compounding frequency, returns a nominal interest rate at the desired compounding frequency.

Parameters

- **dm** (*float*) – the nominal discount rate.
- **m** (*float*) – the compounding frequency.
- **new_m** (*float*) – the desired compounding frequency.

Returns a nominal interest rate, compounded new_m times per year

Return type RateTemplate

4.1.26 tmval.eff_disc_from_nom_disc

`tmval.conversions.eff_disc_from_nom_disc`(*dm: float*, *m: float*, *new_t: float*) → tmval.conversions.RateTemplate

A nominal/effective interest/discount rate converter. Given a nominal discount rate, along with a desired unit of time, returns an effective discount rate at the desired unit of time.

Parameters

- **dm** (*float*) – the nominal discount rate.
- **m** (*float*) – the compounding frequency.
- **new_t** (*float*) – the desired unit of time.

Returns an effective discount rate, at the desired unit of time.

Return type RateTemplate

4.1.27 tmval.nom_disc_from_nom_disc

`tmval.conversions.nom_disc_from_nom_disc`(*dm: float*, *m: float*, *new_m: float*) → tmval.conversions.RateTemplate

A nominal/effective interest/discount rate converter. Given a nominal discount rate, along with a desired compounding frequency, returns an nominal discount rate at the desired compounding frequency.

Parameters

- **dm** (*float*) – the nominal discount rate.
- **m** (*float*) – the compounding frequency.
- **new_m** (*float*) – the desired compounding frequency.

Returns a nominal discount rate, compounded new_m times per year.

Return type RateTemplate

4.1.28 tmval.get_loan_pmt

`tmval.annuity.get_loan_pmt`(*loan_amt: float*, *period: float*, *term: float*, *gr: Union[float, tmval.rate.Rate, tmval.growth.TieredTime]*, *imd: str = 'immediate'*, *gprog: float = 0.0*, *aprog: float = 0.0*, *cents=False*) → dict

Returns the loan payment schedule, given a loan amount, payment period, term, growth rate, and geometric or arithmetic progression of payments. If cents is set to True, uses the Daniel and Vaaler rounding algorithm to round each payment to cents, modifying the final payment such that there is no over/under payment of the loan due to rounding issues.

Parameters

- **loan_amt** (*float*) – The loan amount to be repaid.
- **period** (*float*) – The payment frequency, per fraction of a year.
- **term** (*float*) – The term of the loan, in years.
- **gr** (*Accumulation, Callable, float, Rate*) – Some kind of growth rate object specifying the interest rate
- **imd** (*str*) – 'immediate' or 'due'. Whether the payments occur at the end or beginning of each period, defaults to 'immediate'.

- **gprog** (*float*) – geometric progression, payments grow at a % of the previous payment per period, defaults to 0.
- **aprog** (*float*) – arithmetic progression, payments grow by a constant amount each period, defaults to 0.
- **cents** (*bool*) – Whether you want payments rounded to cents.

Returns a dictionary of payment amounts along with the times of the payments

Return type dict

4.1.29 tmval.get_loan_amt

tmval.annuity.**get_loan_amt**(*down_pmt: float*, *loan_pmt: float*, *period: float*, *term: float*, *gr: tmval.rate.Rate*) → float

Returns a loan amount, given a set of characteristics that defines an annuity.

Parameters

- **down_pmt** (*float*) – A down payment amount, reduces the loan amount.
- **loan_pmt** (*float*) – The periodic loan payment amount.
- **period** (*float*) – The payment period, as a fraction of a year.
- **term** (*float*) – The term of the loan, in years.
- **gr** (*Rate*) – A growth rate object

Returns A loan amount whose present value is equal to the annuity generated from the other arguments.

Return type float

4.1.30 tmval.get_savings_pmt

tmval.annuity.**get_savings_pmt**(*fv: float*, *period: float*, *term: float*, *gr: Union[float, tmval.rate.Rate]*, *cents=False*) → Union[float, tuple]

Returns the savings payments, made at regular installments specified by period, that grow to the desired future value. When cents is set to True, rounds the payments to the next cent, except for the last payment which is adjusted so that the desired future value is reached without over/under payment.

Parameters

- **fv** (*float*) – The desired future value.
- **period** (*float*) – The payment period, as a fraction of a year.
- **term** (*float*) – The amount of time required to reach the desired future value, in years.
- **gr** (*Rate*) – A growth rate object.
- **cents** (*bool*) – Whether you want the payments rounded up to the next cent, except the final one.

Returns A payment amount.

Return type float, or tuple if cents is True

4.1.31 tmval.get_number_of_pmts

`tmval.annuity.`**`get_number_of_pmts`**(*pmt: float*, *fv: float*, *period: float*, *gr:* tmval.rate.Rate) → int

Given payment information represented by an annuity, calculates the number of payments required to reach a desired future value.

Parameters

- **`pmt`** (*`float`*) – The payment amount.
- **`fv`** (*`float`*) – The desired future value.
- **`period`** (*`float`*) – The payment period, as a fraction of a year.
- **`gr`** (`Rate`) – A growth rate object.

Returns The number of payments.

Return type int

4.1.32 tmval.olb_r

`tmval.annuity.`**`olb_r`**(*loan: float*, *q: float*, *period: float*, *gr: Union[*tmval.growth.Accumulation*, float,* tmval.rate.Rate*]*, *t*) → float

Calculates the outstanding loan balance using the retrospective method.

Parameters

- **`loan`** (*`float`*) – The loan amount.
- **`q`** (*`float`*) – The payment amount.
- **`period`** (*`float`*) – The payment period.
- **`gr`** (`Accumulation`, *`float`*, *`or`* `Rate`) – A growth rate object.
- **`t`** (*`float`*) – The valuation time.

Returns The outstanding loan balance.

Return type float

4.1.33 tmval.olb_p

`tmval.annuity.`**`olb_p`**(*q: float*, *period: float*, *term: float*, *gr: Union[*tmval.growth.Accumulation*, float,* tmval.rate.Rate*]*, *t: float*, *r: float = None*, *missed: list = None*) → float

Calculates the outstanding loan balance using the retrospective method.

Parameters

- **`q`** (*`float`*) – The payment amount.
- **`period`** (*`float`*) – The payment period.
- **`term`** (*`float`*) – The loan term, in years.
- **`gr`** (`Accumulation`, *`float`*, *`or Rate.`*) – A growth rate object.
- **`t`** (*`float`*) – The valuation time, in years.
- **`r`** (*`float`*, *`optional`*) – The final payment amount, if different from the others, defaults to None.

- **missed** (*list*) – A list of missed payments, for example, 4th and 5th payments would be [4, 5].

Returns The outstanding loan balance.

Return type float

4.1.34 tmval.get_perpetuity_gr

tmval.annuity.**get_perpetuity_gr**(*amount: float*, *pv: float*, *period: float*, *imd: str = 'immediate'*) → *tmval.rate.Rate*

Given a payment amount, payment frequency, and present value, returns a compound annual effective interest rate.

Parameters

- **amount** (*float*) – The payment amount.
- **pv** (*float*) – The present value.
- **period** (*float*) – The payment period, as a fraction of a year.
- **imd** (*str*) – Whether the perpetuity is 'immediate' or 'due'.

Returns The annual effective interest rate.

Return type *Rate*

4.1.35 tmval.get_perpetuity_pmt

tmval.annuity.**get_perpetuity_pmt**(*gr:* tmval.rate.Rate, *pv: float*, *period: float*, *imd: str = 'immediate'*) → float

Given an interest rate, present value, and payment period, returns the payment amount of a perpetuity.

Parameters

- **gr** (Rate) – A growth rate object.
- **pv** (*float*) – The present value.
- **period** (*float*) – The payment period.
- **imd** (*str*) – Whether the perpetuity is 'immediate' or 'due'.

Returns The payment amount.

Return type float

4.1.36 tmval.standardize_rate

tmval.rate.**standardize_rate**(*gr: Union[float,* tmval.rate.Rate*]*) → *tmval.rate.Rate*

Given a float or Rate object, standardizes it to be simple or compound effective interest with an interval of one year. If a float is given, returns a Rate object representing a compound effective interest rate with an interval of one year.

If a Rate is given that represents a simple interest or discount rate, returns a simple interest Rate object with an interval of one year.

If a Rate is given that represents a compound rate (effective interest, nominal interest, effective discount, nominal discount, force of interest), returns a compound effective interest Rate object with an interval of one year.

Parameters **gr** (*float, Rate*) – A growth rate object.

Returns A standardized interest rate.

Return type *Rate*

4.1.37 tmval.read_iym

`tmval.growth.`**`read_iym`**(*table: dict*, *t0: float*, *t: float*) → *tmval.rate.Rate*

Reads a value from a table of interest rates using the investment year method. A table might look something like this:

iym_table = { 2000: [.06, .065, .0575, .06, .065], 2001: [.07, .0625, .06, .07, .0675], 2002: [.06, .06, .0725, .07, .0725], 2003: [.0775, .08, .08, .0775, .0715] }

Parameters

- **table** (*dict*) – A table of interest rates as a dictionary, with years as the keys.
- **t0** (*float*) – The initial investment time.
- **t** (*float*) – The desired lookup time for which the rate applies.

Returns An interest rate applicable to the desired lookup time.

Return type *Rate*

4.1.38 tmval.tt_iym

`tmval.growth.`**`tt_iym`**(*table: dict*, *t0: float*) → *tmval.growth.TieredTime*

Reads an investment year method table and returns a TieredTime object. This object can then be passed to an Amount or an Accumulation class to represent a series of interest rates applicable to an investment. A table might look something like this:

iym_table = { 2000: [.06, .065, .0575, .06, .065], 2001: [.07, .0625, .06, .07, .0675], 2002: [.06, .06, .0725, .07, .0725], 2003: [.0775, .08, .08, .0775, .0715] }

Parameters

- **table** (*dict*) – A table of interest rates.
- **t0** (*float*) – The year of the initial investment.

Returns A TieredTime growth rate object.

Return type *TieredTime*

4.1.39 tmval.standardize_acc

`tmval.growth.`**`standardize_acc`**(*gr: Union[tmval.growth.Accumulation, float, tmval.rate.Rate, tmval.growth.TieredTime]*) → *tmval.growth.Accumulation*

Returns an compound accumulation object. Usually used to enable more complex classes and functions to accept several different objects to indicate a compound interest growth rate.

Parameters **gr** (*Accumulation, float, Rate, or TieredTime*) – A growth rate object.

Returns an Accumulation object

Return type *Accumulation*

4.1.40 tmval.simple_interval_solver

`tmval.growth.`**`simple_interval_solver`**(*s: float*, *es: float*) → float

Finds the interval at which the simple interest rate equals es

Parameters

- **`s`** (*`float`*) – A simple interest rate.
- **`es`** (*`float`*) – The desired simple interest rate.

Returns The desired interval.

Return type float

4.1.41 tmval.pairwise

`tmval.value.`**`pairwise`**(*iterable: Iterable*) → Iterator

Helper function to enable pairwise iteration.

Parameters **`iterable`** (*`iterable`*) – An iterable object.

Returns An iterator.

Return type Iterator

4.1.42 tmval.time_weighted_yield

`tmval.value.`**`time_weighted_yield`**(*balance_times: list*, *balance_amounts: list*, *payments: tmval.value.Payments = None*, *payment_times: list = None*, *payment_amounts: list = None*, *annual: bool = False*) → *tmval.rate.Rate*

Given a list of balances and payments, returns the time-weighted yield. If annual is set to True, returns the rate as an annual rate. Otherwise, the rate is effective over the investment term.

You may supply a Payments object, or specify the components separately.

Parameters

- **`balance_times`** (*`list`*) – A list of balance times.
- **`balance_amounts`** (*`list`*) – A list of balance amounts, corresponding to the balance times.
- **`payments`** (`Payments`) – A Payments object.
- **`payment_times`** (*`list`*) – A list of payment times.
- **`payment_amounts`** (*`list`*) – A list of payment amounts, corresponding to the payment times.
- **`annual`** (*`bool, defaults to False`*) – Whether you want the time-weighted yield to be annualized.

Returns The time-weighted yield.

Return type *Rate*

4.1.43 tmval.parse_cgr

`bond.`**`parse_cgr`**(*cfreq: Union[float, list] = None*, *cgr: Union[tmval.rate.Rate, tmval.growth.TieredTime] = None*) → dict

Parses the coupon-related arguments provided to the Bond class. It returns a dictionary containing items that are used to calculate other features of the bond - alpha, cfreq, and cgr. This is used to improve the flexibility of the arguments one may supply to specify coupons.

This function can for example, extract the rate and frequency components out of the cgr argument and supply them to alpha and cfreq, respectively. Or, it can take the alpha and cfreq values to calculate cgr.

It also handles the case where alpha and cfreq are not constant throughout the term of the bond, in which case cgr is calculated to be a TieredTime and vice versa.

Parameters

- **alpha** (*float, list*) – When supplied as a float, a nominal coupon rate. When supplied as a list of tuples of the form [(x1, y_1), (x2, y2), . . .], represents a series of coupon rates x supplied during periods beginning at time y. For example alpha=[(.05, 0), (.04, 2)] means a 5% coupon rate the first two years and a 4% rate after that.
- **cfreq** (*float, list*) – The coupon frequency, or a list of coupon frequencies. When supplied as a list, alpha must also be supplied as a list and each element in cfreq needs to correspond to a tuple in alpha.
- **cgr** (Rate, TieredTime) – A nominal rate representing the coupon rate and frequency.

Returns A dictionary containing the alpha, cfreq, and cgr values.

Return type dict

4.1.44 tmval.rate_from_earned

`growth.`**`rate_from_earned`**(*iey: Tuple[float, float]*) → *tmval.rate.Rate*

Given the amount of interest earned in two time periods, calculates the annualized effective interest rate.

Parameters

- **iex** (*Tuple[float, float]*) – Interest earned in period x, provided as (interest earned, period x).
- **iey** (*Tuple[float, float]*) – Interest earned in period y, provided as (interest earned, period y).

Returns An annualized effective interest rate.

Return type *Rate*

4.1.45 tmval.rate_from_intdisc

`growth.`**`rate_from_intdisc`**(*dex: Tuple[float, float], x0=array([0.001, 0.01109091, 0.02118182, 0.03127273, 0.04136364, 0.05145455, 0.06154545, 0.07163636, 0.08172727, 0.09181818, 0.10190909, 0.112, 0.12209091, 0.13218182, 0.14227273, 0.15236364, 0.16245455, 0.17254545, 0.18263636, 0.19272727, 0.20281818, 0.21290909, 0.223, 0.23309091, 0.24318182, 0.25327273, 0.26336364, 0.27345455, 0.28354545, 0.29363636, 0.30372727, 0.31381818, 0.32390909, 0.334, 0.34409091, 0.35418182, 0.36427273, 0.37436364, 0.38445455, 0.39454545, 0.40463636, 0.41472727, 0.42481818, 0.43490909, 0.445, 0.45509091, 0.46518182, 0.47527273, 0.48536364, 0.49545455, 0.50554545, 0.51563636, 0.52572727, 0.53581818, 0.54590909, 0.556, 0.56609091, 0.57618182, 0.58627273, 0.59636364, 0.60645455, 0.61654545, 0.62663636, 0.63672727, 0.64681818, 0.65690909, 0.667, 0.67709091, 0.68718182, 0.69727273, 0.70736364, 0.71745455, 0.72754545, 0.73763636, 0.74772727, 0.75781818, 0.76790909, 0.778, 0.78809091, 0.79818182, 0.80827273, 0.81836364, 0.82845455, 0.83854545, 0.84863636, 0.85872727, 0.86881818, 0.87890909, 0.889, 0.89909091, 0.90918182, 0.91927273, 0.92936364, 0.93945455, 0.94954545, 0.95963636, 0.96972727, 0.97981818, 0.98990909, 1.0]), precision=5*) → list

Given interest earned over a time period on an unknown investment amount, and discount earned over a time period, on that same amount,, solves for an annualized compound interest rate.

Parameters

- **`iex`** (*`Tuple[float, float]`*) – The interest earned over a time interval.
- **`dex`** (*`Tuple[float, float]`*) – The discount earned over a time interval.
- **`x0`** – A starting guess or list of guesses for Newton's method, defaults to np.linspace(.001, 1, 100).
- **`precision`** (*`int`*) – rounding precision used to remove duplicates from Newton's method, defaults to 5.

Returns a list of interest rates, if found.

Return type list

4.1.46 tmval.amt_from_intdisc

`growth.`**`amt_from_intdisc`**(*dex: Tuple[float, float], x0: Union[float, numpy.ndarray] = array([0.001, 0.01109091, 0.02118182, 0.03127273, 0.04136364, 0.05145455, 0.06154545, 0.07163636, 0.08172727, 0.09181818, 0.10190909, 0.112, 0.12209091, 0.13218182, 0.14227273, 0.15236364, 0.16245455, 0.17254545, 0.18263636, 0.19272727, 0.20281818, 0.21290909, 0.223, 0.23309091, 0.24318182, 0.25327273, 0.26336364, 0.27345455, 0.28354545, 0.29363636, 0.30372727, 0.31381818, 0.32390909, 0.334, 0.34409091, 0.35418182, 0.36427273, 0.37436364, 0.38445455, 0.39454545, 0.40463636, 0.41472727, 0.42481818, 0.43490909, 0.445, 0.45509091, 0.46518182, 0.47527273, 0.48536364, 0.49545455, 0.50554545, 0.51563636, 0.52572727, 0.53581818, 0.54590909, 0.556, 0.56609091, 0.57618182, 0.58627273, 0.59636364, 0.60645455, 0.61654545, 0.62663636, 0.63672727, 0.64681818, 0.65690909, 0.667, 0.67709091, 0.68718182, 0.69727273, 0.70736364, 0.71745455, 0.72754545, 0.73763636, 0.74772727, 0.75781818, 0.76790909, 0.778, 0.78809091, 0.79818182, 0.80827273, 0.81836364, 0.82845455, 0.83854545, 0.84863636, 0.85872727, 0.86881818, 0.87890909, 0.889, 0.89909091, 0.90918182, 0.91927273, 0.92936364, 0.93945455, 0.94954545, 0.95963636, 0.96972727, 0.97981818, 0.98990909, 1.0]), precision: int = 5*) → *tmval.growth.Amount*

Given interest earned over a time period on an unknown investment amount, and discount earned over a time period, on that same amount, generates an Amount object.

For example, if you can earn 500 in interest in two years, and the discount in one year is 200, you can supply iex=[400, 2], dex=[200,1]

Parameters

- **`iex`** (*`Tuple[float, float]`*) – The interest earned over a time interval.
- **`dex`** (*`Tuple[float, float]`*) – The discount earned over a time interval.
- **`x0`** – A starting guess or list of guesses for Newton's method, defaults to np.linspace(.001, 1, 100).
- **`precision`** (*`int`*) – rounding precision used to remove duplicates from Newton's method, defaults to 5.

Returns An amount function.

Return type *Amount*

4.1.47 tmval.k_from_intdisc

`growth.`**`k_from_intdisc`**(*dex: Tuple[float, float], x0: Union[float, numpy.ndarray] = array([0.001, 0.01109091, 0.02118182, 0.03127273, 0.04136364, 0.05145455, 0.06154545, 0.07163636, 0.08172727, 0.09181818, 0.10190909, 0.112, 0.12209091, 0.13218182, 0.14227273, 0.15236364, 0.16245455, 0.17254545, 0.18263636, 0.19272727, 0.20281818, 0.21290909, 0.223, 0.23309091, 0.24318182, 0.25327273, 0.26336364, 0.27345455, 0.28354545, 0.29363636, 0.30372727, 0.31381818, 0.32390909, 0.334, 0.34409091, 0.35418182, 0.36427273, 0.37436364, 0.38445455, 0.39454545, 0.40463636, 0.41472727, 0.42481818, 0.43490909, 0.445, 0.45509091, 0.46518182, 0.47527273, 0.48536364, 0.49545455, 0.50554545, 0.51563636, 0.52572727, 0.53581818, 0.54590909, 0.556, 0.56609091, 0.57618182, 0.58627273, 0.59636364, 0.60645455, 0.61654545, 0.62663636, 0.63672727, 0.64681818, 0.65690909, 0.667, 0.67709091, 0.68718182, 0.69727273, 0.70736364, 0.71745455, 0.72754545, 0.73763636, 0.74772727, 0.75781818, 0.76790909, 0.778, 0.78809091, 0.79818182, 0.80827273, 0.81836364, 0.82845455, 0.83854545, 0.84863636, 0.85872727, 0.86881818, 0.87890909, 0.889, 0.89909091, 0.90918182, 0.91927273, 0.92936364, 0.93945455, 0.94954545, 0.95963636, 0.96972727, 0.97981818, 0.98990909, 1.0]), precision: int = 5*) → float

Given interest earned over a time period on an unknown investment amount, and discount earned over a time period, on that same amount, solves for the amount.

Parameters

- **iex** (*Tuple[float, float]*) – The interest earned over a time interval.
- **dex** (*Tuple[float, float]*) – The discount earned over a time interval.
- **x0** – A starting guess or list of guesses for Newton's method, defaults to np.linspace(.001, 1, 100).
- **precision** (*int*) – rounding precision used to remove duplicates from Newton's method, defaults to 5.

Returns The investment amount.

Return type float

4.2 Amount

class `tmval.growth.`**`Amount`**(*gr: Union[Callable, tmval.rate.Rate, float], k: float*)

The Amount class is an implementation of the amount function, which describes how much an invested amount of money grows over time.

The amount function's methods can return things like the valuation of an investment after a specified time, and effective interest and discount rates over an interval.

The accumulation function, which is a special case of the amount function where k=1, can be extracted from the amount function using the get_accumulation() method.

Parameters

- **gr** (*Callable, float, Rate*) – a growth object, which can either be a function that must take the parameters t for time and k for principal, or a Rate object representing an interest rate.
- **k** (*float*) – the principal, or initial investment.

Returns An amount object, which can be used like an amount function in interest theory.

Return type *Amount*

—

4.2.1 tmval.Amount.val

`Amount.val(t: float) → float`

Calculates the value of the investment at a point in time.

:param t:evaluation date. The date at which you would like to know the value of the investment. :type t: float :return: the value of the investment at time t. :rtype: float

4.2.2 tmval.Amount.interest_earned

`Amount.interest_earned(t1: float, t2: float) → float`

Calculates the amount of interest earned over a time period.

Parameters

- **t1** (*float*) – beginning of the period.
- **t2** (*float*) – end of the period.

Returns The amount of interest earned over the time period.

Return type float

4.2.3 tmval.Amount.effective_rate

`Amount.effective_rate(n: Union[float, int]) → tmval.rate.Rate`

Calculates the effective interest rate for the n-th time period.

Parameters **n** (*float, int*) – the n-th time period.

Returns the effective interest rate for the n-th timer period.

Return type float

4.2.4 tmval.Amount.effective_interval

`Amount.effective_interval(t2: float, t1: float = 0, annualized: bool = False) → tmval.rate.Rate`

Calculates the effective interest rate over a time period.

Parameters

- **t1** (*float*) – the beginning of the period.
- **t2** (*float*) – the end of the period.
- **annualized** – whether you want the results to be annualized.

Returns the effective interest rate over the time period.

Return type float

Rtype annualized bool

4.2.5 tmval.Amount.discount_interval

`Amount.`**`discount_interval`**`(`*t1: float*, *t2: float*`)` → float

Calculates the effective discount rate over a time period.

Parameters

- **`t1`** (*`float`*) – the beginning of the time period.
- **`t2`** (*`float`*) – the end of the time period.

Returns the effective discount rate over the time period.

Return type float

4.2.6 tmval.Amount.effective_discount

`Amount.`**`effective_discount`**`(`*n: int*`)` → float

Calculates the effective discount rate for the n-th time period.

Parameters **`n`** (*`int`*) – the n-th time period.

Returns the effective discount rate for the n-th time period.

Return type float

4.2.7 tmval.Amount.get_accumulation

`Amount.`**`get_accumulation`**`()` → *tmval.growth.Accumulation*

Extracts the *accumulation function*, a special case of the amount function where k=1.

Returns the accumulation function.

Return type *Accumulation*

4.3 Accumulation

`class` `tmval.growth.`**`Accumulation`**`(`*gr: Union[Callable, float,* tmval.rate.Rate*]*`)`

Special case of Amount function where k=1, Accepts an accumulation amount function, can return valuation at time t and effective interest rate on an interval

Parameters **`gr`** (*`Callable, float, Rate`*) – a growth object, which can either be a function that must take the parameters t for time and k for principal, or a Rate object representing an interest rate.

Returns an accumulation object.

Return type *Accumulation*

4.3.1 tmval.Accumulation.val

`Accumulation.val(t: float) → float`

Calculates the value of the investment at a point in time.

Parameters **t** (*float*) – evaluation date. The date at which you would like to know the value of the investment.

Returns the value of the investment at time t.

Return type float

4.3.2 tmval.Accumulation.discount_func

`Accumulation.discount_func(t: float, fv: Optional[float] = None) → float`

The discount function is the reciprocal of the accumulation function. Returns the discount factor at time t, which can be used to get the present value of an investment.

Parameters

- **t** (*float*) – the time at which you would like to get the discount factor.
- **fv** (*float, optional*) – float: the future value. Assumed to be 1 if not provided.

Returns the discount factor at time t

Return type float

4.3.3 tmval.Accumulation.future_principal

`Accumulation.future_principal(fv: float, t1: float, t2: float) → float`

Finds the principal needed at t1 to get fv at t2.

Parameters

- **fv** (*float*) – future value.
- **t1** (*float*) – time of investment.
- **t2** (*float*) – time of goal.

Returns amount of money needed at t1 to get fv at t2.

Return type float

4.4 TieredBal

`class tmval.growth.TieredBal(tiers: list, rates: list)`

TieredBal is a callable growth pattern for tiered investment accounts. A tiered investment account is one where the interest paid varies depending on the balance. For example, 1% for the first $1000, 2% for the next $4000, and 3% afterward.

To create a *TieredBal*, supply a list of tiers and a corresponding list of interest rates for those tiers. The tiers are the lower bounds for the intervals corresponding to the interest rate (the first value will usually be 0).

Parameters

- **tiers** (*list*) – a list of tiers, for example [0, 1000, 5000].

- **rates** (*list*) – a list of interest rates, for example [.01, .02, .03].

Returns a TieredBal object.

Return type *TieredBal*

4.4.1 tmval.TieredBal.get_jump_times

`TieredBal.`**`get_jump_times`**(*k: float*) → list

Calculates the times at which the interest rate is expected to change for the account, assuming an initial investment of k and no further investments.

Parameters **k** (*float*) – the principal, or initial investment.

Returns a list of times at which the interest rate is expected to change for the account.

Return type list

4.5 TieredTime

class `tmval.growth.`**`TieredTime`**(*tiers: list*, *rates: list*)

TieredTime is a callable growth pattern for investment accounts in which the interest rate can vary depending on how long the account stays open. For example, 1% for the first year, 2% for the next year, and 3% afterward.

To create a *TieredTime*, supply a list of tiers and a corresponding list of interest rates for those tiers. The tiers are the lower bounds for the intervals corresponding to the interest rate (the first value will usually be 0).

Parameters

- **tiers** (*list*) – a list of tiers, for example [0, 1, 2].
- **rates** (*list*) – a list of interest rates, for example [.01, .02, .03].

Returns a TieredTime object.

Return type *TieredTime*

4.6 SimpleLoan

class `tmval.growth.`**`SimpleLoan`**(*principal: float*, *term: float*, *discount_amt: Optional[float] = None*, *discount_rate: Optional[float] = None*)

A callable growth pattern for a simple loan, which is a lump sum loan to be paid back with a single payment with interest, and possibly no explicit rate given. A common type of informal loan between two people outside the banking system.

You should supply a discount amount, a discount rate, but not both.

Parameters

- **principal** (*float*) – the initial investment.
- **term** (*float*) – the term of the loan.
- **discount_amt** (*float*) – the discount amount, defaults to None.
- **discount_rate** (*float*) – the discount_rate, defaults to None.

Returns a *SimpleLoan* object when initialized, the value when called.

Return type SimpleLoan when initialized, float when called.

4.7 Annuity

class `tmval.annuity.`**`Annuity`**(*gr: Union[tmval.growth.Accumulation, Callable, float, tmval.rate.Rate], amount: Union[float, int, list, Callable] = 1.0, period: float = 1, term: float = None, n: float = None, gprog: float = 0.0, aprog: Union[float, int, tuple] = 0.0, times: list = None, reinv: [<class 'tmval.growth.Accumulation'>, typing.Callable, <class 'float'>, <class 'tmval.rate.Rate'>] = None, deferral: float = 0.0, imd: str = 'immediate', loan: float = None, drb: str = None*)

Annuity is TmVal's general class for representing all kinds of annuities. Among the supported types are:

1. Annuity-immediate
2. Annuity-due
3. Perpetuity-immediate
4. Perpetuity-due
5. Deferred annuity
6. Annuity with arithmetic progression
7. Annuity with geometric progression
8. Perpetuity with arithmetic progression
9. Perpetuity with geometric progression
10. Annuity with reinvested proceeds at a different growth rate

By tweaking the arguments, you can represent more types of annuities that you might find in actuarial literature.

The Annuity class comes with methods to solve for the present value, accumulated value, and equation of value at periods other than 0 or end-of-term. You do not have to provide overlapping time-related arguments. For example, if you provide the period and number of payments, the term can be inferred. Likewise, if you provide the term and period, the number of payments can be inferred.

Parameters

- **gr** (`Accumulation`, `Callable`, `float`, `Rate.`) – A growth rate object.
- **amount** (`float`, `int`, `list`) – A payment amount, defaults to 1. Can also provide a list for non-level payments.
- **period** (`float`) – The payment period, defaults to 1.
- **term** (`float`) – The annuity term.
- **n** (`float`) – The number of payments.
- **aprog** (`float`) – Arithmetic progression of payments defaults to 0.
- **gprog** (`float`) – Geometric progression of payments, defaults to 0.
- **times** (`list`) – Payment times, usually left blank but can be provided for nontraditional annuities, defaults to None
- **reinv** (`Accumulation`, `Callable`, `float`, `Rate`) – A reinvestment rate, if proceeds are reinvested at a rate other than provided to gr.
- **deferral** (`float`) – A time period in years to indicate a deferred annuity.

- **imd** (*str*) – Whether the annuity is 'immediate' or 'due', defaults to 'immediate'.
- **loan** (*float, optional*) – An amount that allows you to represent an annuity as a loan from which the payments can be inferred, defaults to None.
- **drb** (*str, optional*) – Whether the final loan payment is a 'drop' or 'balloon' payment.

—

4.7.1 tmval.Annuity.pv

`Annuity.pv()` → float

Calculates the present value of the annuity. The formula used to calculate the present value will depend on the type of annuity that gets inferred from the arguments provided to the parent class. Several classes of annuities come with shortcut formulas which will be used in favor of just the standard npv or equation of value formulas to improve computation speed:

1. Annuity-immediate: $a_{\overline{n|}i}$
2. Annuity-due: $\ddot{a}_{\overline{n|}i}$
3. Perpetuity-immediate: $a_{\overline{\infty|}i}$
4. Perpetuity-due: $\ddot{a}_{\overline{\infty|}i}$
5. Arithmetically increasing annuity-immediate: $(I_{P,Q}a)_{\overline{n|}i}$
6. Arithmetically increasing annuity-due: $(I_{P,Q}\ddot{a})_{\overline{n|}i}$
7. Arithmetically increasing perpetuity-immediate: $(I_{P,Q}a)_{\overline{\infty|}i}$
8. Arithmetically increasing perpetuity-due: $(I_{P,Q}a)_{\overline{\infty|}i}$
9. Geometrically increasing annuity-immediate
10. Geometrically increasing annuity-due
11. Geometrically increasing perpetuity-immediate
12. Geometrically increasing perpetuity-due

Returns The present value of the annuity.

Return type float

4.7.2 tmval.Annuity.sv

`Annuity.sv()`

Calculates the accumulated value of the annuity. The formula used to calculate the accumulated value will depend on the type of annuity that gets inferred from the arguments provided to the parent class. Several classes of annuities come with shortcut formulas which will be used in favor of the equation of value formula to improve computation speed:

1. Annuity-immediate: $s_{\overline{n|}i}$
2. Annuity-due: $\ddot{s}_{\overline{n|}i}$
3. Perpetuity-immediate: $s_{\overline{\infty|}i}$
4. Perpetuity-due: $\ddot{s}_{\overline{\infty|}i}$

5. Arithmetically increasing annuity-immediate: $(I_{P,Q}s)_{\overline{n}|i}$
6. Arithmetically increasing annuity-due: $(I_{P,Q}\ddot{s})_{\overline{n}|i}$
7. Arithmetically increasing perpetuity-immediate: $(I_{P,Q}s)_{\overline{\infty}|i}$
8. Arithmetically increasing perpetuity-due: $(I_{P,Q}s)_{\overline{\infty}|i}$
9. Geometrically increasing annuity-immediate
10. Geometrically increasing annuity-due
11. Geometrically increasing perpetuity-immediate
12. Geometrically increasing perpetuity-due

Returns The accumulated value of the annuity.

Return type float

4.7.3 tmval.Annuity.get_r_pmt

`Annuity.`**`get_r_pmt`**`(`*gr: Union[*tmval.growth.Accumulation*, Callable, float,* tmval.rate.Rate*]*`)` → float

When inferring the number of payment periods, returns the number of payment periods as if fractional periods were allowed. This fractional value helps calculate the drop or balloon payments, if needed.

Parameters **`gr`** (`Accumulation`, *`Callable`*, *`float`*, *`or`* `Rate`) – A growth rate object.

Returns The number of payment periods.

Return type float

4.7.4 tmval.Annuity.get_drop

`Annuity.`**`get_drop`**`(`*gr*`)` → float

If the number of payment periods does not settle to an integral number, calculates the drop payment.

Parameters **`gr`** (`Accumulation`, *`Callable`*, *`float`*, *`or`* `Rate`) – A growth rate object.

Returns The drop payment.

Return type float

4.7.5 tmval.Annuity.get_balloon

`Annuity.`**`get_balloon`**`(`*gr*`)` → float

If the number of payment periods does not settle to an integral number, calculates the balloon payment.

Parameters **`gr`** (`Accumulation`, *`Callable`*, *`float`*, *`or`* `Rate`) – A growth rate object.

Returns The balloon payment.

Return type float

4.8 Payments

class tmval.value.**Payments**(*amounts: Union[list, Iterable, Callable], times: Union[list, Iterable, Callable], gr: Union[float,* tmval.rate.Rate*,* tmval.growth.Accumulation*,* tmval.growth.TieredBal*,* tmval.growth.TieredTime*] = None*)

A collection of payments, and corresponding growth object. If no growth object (an interest rate, Rate object, Accumulation object) is provided, the payments are assumed to be undiscounted. The payments class serves as the backbone for major types of financial instruments, such as annuities and bonds. It provides methods for calculating net present value, internal rate of return (dollar weighted yield), equated time, equated value, and time-weighted yield.

Parameters

- **amounts** (*list, Iterable, Callable*) – a list of payment amounts.
- **times** (*list, Iterable, Callable*) – a list of payment times.
- **gr** (*float,* Rate*, or* Accumulation) – a growth rate object, can be supplied as a float, a Rate object, or an Accumulation object.

4.8.1 tmval.Payments.equated_time

tmval.value.Payments.**equated_time**(*self, c: float*) → float

Method of equated time. Finds T so that a single payment of $C = \sum_{k=1}^{n} C_{t_k}$ at time T has the same value at $t = 0$ as the sequence of n contributions.

While this method is formally defined to have C equal the sum of the contributions, it actually works when C is not equal to the sum of the contributions.

Parameters **c** (*float*) – The single payment C.

Returns The time T.

Return type float

4.8.2 tmval.Payments.irr

tmval.value.Payments.**irr**(*self, x0: float = 1.05*) → list

Calculates the internal rate of return, also known as the yield rate or dollar-weighted return. If the payment amounts and times result in a polynomial equation of value, the yield is solved by calculating the roots of the polynomial via the NumPy roots function. If the equation of value is not a polynomial, than Newton's method from the SciPy package is used.

Parameters **x0** (*float*) – A starting guess when using Newton's method, defaults to 1.05.

Returns A list of real roots, if found.

Return type list

4.8.3 tmval.Payments.dw_approx

`tmval.value.Payments.`**`dw_approx`**(*self*, *a: float = None*, *b: float = None*, *w_t: float = None*, *k_approx: bool = False*, *k: float = 0.5*, *annual: bool = False*) → *tmval.rate.Rate*

Calculates the approximate dollar-weighted yield rate by standardizing the investment time to 1:

$$j \approx \frac{I}{A + \sum_{t\in(0,1)} C_t(1-t)}$$

Where A is the beginning balance, I is interest earned, and the Cs are the contributions. When k_approx is set to true, k is assumed to be a fixed constant within the investment window:

$$j \approx \frac{I}{A + C(1-k)}$$

The default value for k is 1/2, simplifying the above expression to:

j approx frac{2I}{A + B - I}

Where B is the withdrawal balance. When arguments a, b, and w_t (corresponding to A, B, and the the withdrawal time) are not provided, a is assumed to be the first payment in the parent object, and b is calculated to be the last.

Parameters

- **a** (*float*) – The initial balance.
- **b** (*float*) – The withdrawal balance.
- **w_t** (*float*) – The withdrawal time.
- **k_approx** (*bool*) – Whether you want to use the k-approximation formula.
- **k** (*float*) – The value for k in the k-approximation formula, defaults to .5.
- **annual** (*bool*) – Whether you want the results annualized.

Returns The approximate dollar-weighted yield rate.

Return type *Rate*

4.9 Rate

class `tmval.rate.`**`Rate`**(*rate: float = None*, *pattern: str = None*, *freq: float = None*, *interval: float = None*, *i: float = None*, *d: float = None*, *delta: float = None*, *s: float = None*, *sd: float = None*)

The Rate class is TmVal's core class for representing interest rates. The magnitude of the rate is specified via the rate argument.

The type of interest rate is indicated by providing a value to the pattern argument. Valid patterns are:

1. Effective Interest
2. Effective Discount
3. Nominal Interest
4. Nominal Discount
5. Force of Interest
6. Simple Interest

7. Simple Discount

For Effective Interest, Effective Discount, Simple Interest, and Simple Discount, an additional argument called interval is used to denote the effective interval. These rates can be created via shortcut arguments i, d, s, and sd. When a shortcut argument is used, the interval defaults to 1 and is not strictly necessary, unless the interval happens to be different.

If you just want to initialize an annual Effective Interest rate, you do not need to use the argument i, simply pass a float object to the Rate class, as in gr=Rate(.05) to define a 5% annually compounded effective interest rate.

For the force of interest, use the shortcut argument delta.

For Nominal Interest and Nominal Discount, you need to provide the compounding frequency via the freq argument.

The compound patterns - Effective Interest, Effective Discount, Nominal Interest, Nominal Discount, and Force of Interest can be converted to each other.

Simple Interest can be converted to another Simple Interest with a different interval. Simple Discount can be converted to another Simple Discount with a different interval. Simple Interest and Simple Discount cannot be converted to each other because they do not correspond to the same accumulation function.

Compound patterns cannot be converted to simple patterns.

The Rate class mimics arithmetic types, so you may perform arithmetic operations on rate objects with other arithmetic types.

Parameters

- **`rate`** (*`float`*) – The magnitude of the rate.
- **`pattern`** (*`str`*) – The interest rate pattern.
- **`freq`** (*`float`*) – The compounding frequency, in times per year.
- **`interval`** (*`float`*) – The effective interval of the interest rate, in years.
- **`i`** (*`float`*) – Shortcut argument for Effective Interest.
- **`d`** (*`float`*) – Shortcut argument for Effective Discount.
- **`delta`** (*`float`*) – Shortcut argument for Force of Interest.
- **`s`** (*`float`*) – Shortcut argument for Simple Interest.
- **`sd`** (*`float`*) – Shortcut argument for Simple Discount.

4.9.1 tmval.Rate.convert_rate

`Rate.`**`convert_rate`**(*pattern*, *freq: float = None*, *interval: float = None*)

Converts the rate to the desired pattern.

The compound patterns - Effective Interest, Effective Discount, Nominal Interest, Nominal Discount, and Force of Interest can be converted to each other.

Simple Interest can be converted to another Simple Interest with a different interval. Simple Discount can be converted to another Simple Discount with a different interval. Simple Interest and Simple Discount cannot be converted to each other because they do not correspond to the same accumulation function.

Compound patterns cannot be converted to simple patterns.

Parameters

- **`pattern`** (*`str`*) – The pattern to which you want to convert the rate.

- **freq** (*float*) – The compounding frequency, times per year.
- **interval** (*float*) – The effective interval, in years.

Returns A rate object.

Return type *Rate*

4.9.2 tmval.Rate.amt_func

`Rate.amt_func(k, t)`

A Callable intended to be used by the Amount class when the parent Rate object is passed to it. This allows an Amount object to be declared by passing a Rate object to it.

Parameters

- **k** (*float*) – The principal.
- **t** (*float*) – The valuation time, in years.

Returns The value of k at time t.

Return type float

4.9.3 tmval.Rate.acc_func

`Rate.acc_func(t)`

A Callable intended to be used by the Accumulation class when the parent Rate object is passed to it. This allows an Accumulation object to be declared by passing a Rate object to it.

Parameters **t** (*float*) – The valuation time, in years.

Returns The value of 1 invested at time 0, at time t.

Return type float

4.9.4 tmval.Rate.standardize

`Rate.standardize()`

Used to put rates on a common basis for comparison. The following patterns will be converted to Effective Interest compounded annually:

1. Effective Interest
2. Effective Discount
3. Nominal Interest
4. Nominal Discount
5. Force of Interest

The following patterns will be converted to Simple Interest effective over a 1-year interval:

1. Simple Interest

The following patterns will be converted to Simple Discount effective over a 1-year interval:

1. Simple Discount

Returns A standardized Rate object.

Return type *Rate*

4.10 Bond

class `tmval.bond.`**`Bond`**(*face: float = None*, *term: float = None*, *red: float = None*, *gr: Union[float, tmval.rate.Rate] = None*, *cgr: tmval.rate.Rate = None*, *alpha: Union[float, list] = None*, *cfreq: Union[float, list] = None*, *price: float = None*, *pd: float = None*, *k: float = None*, *fr: float = None*)

Bond is TmVal's class for representing bonds. Bonds can be initialized if the arguments contain enough information to calculate:

1. Bond price (price)
2. Coupons (alpha and cfreq or cgr)
3. Yield rate (gr)
4. Redemption amount (red)
5. Face value (face)
6. Term (term)

When one of these values is absent, the initialization will be able to solve for the missing quantity. When two values are missing, you can still initialize the bond under certain cases:

1. Missing price and redemption amount - can be solved if premium/discount is supplied.
2. Missing price and term - can be solved if Makeham's k is supplied.
3. Missing price and coupon rate - can be solved if coupon amount is supplied.

Coupon information is usually supplied by specifying an alpha amount for the nominal coupon rate and cfreq for the coupon frequency. Alternatively, you may supply a nominal interest rate with a compounding frequency to the argument cgr. It's possible to specify a coupon amount via fr and the coupon frequency of cfreq if you do not know alpha but know the coupon amount.

Parameters

- **`face`** (*`float`*) – The face value.
- **`red`** (*`float`*) – The redemption amount.
- **`gr`** (*`float, Rate`*) – The yield rate used to price the bond.
- **`cgr`** (*`Rate`*) – A nominal rate representing the coupon rate and frequency.
- **`alpha`** (*`float, list`*) – When supplied as a float, a nominal coupon rate. When supplied as a list of tuples of the form [(x1, y_1), (x2, y2), ...], represents a series of coupon rates x supplied during periods beginning at time y. For example alpha=[(.05, 0), (.04, 2)] means a 5% coupon rate the first two years and a 4% rate after that.
- **`cfreq`** (*`float, list`*) – The coupon frequency, or a list of coupon frequencies. When supplied as a list, alpha must also be supplied as a list and each element in cfreq needs to correspond to a tuple in alpha.
- **`price`** (*`float`*) – The bond price.
- **`pd`** (*`float`*) – The bond premium or discount.
- **`k`** (*`float`*) – The present value of the redemption amount, used if instantiating from Makeham's formula.

- **fr** (*float*) – The coupon amount per period. Can be supplied with cfreq instead of alpha.

4.10.1 tmval.Bond.get_coupon_times

`Bond.`**`get_coupon_times`**`()` → list

Calculates the times at which the coupon payments occur.

Returns A list of coupon payment times.

Return type list

4.10.2 tmval.Bond.get_coupon_amt

`Bond.`**`get_coupon_amt`**`()` → Union[float, list]

Calculates the coupon amount, or a list of coupon amounts and time periods in which they are applicable if they are not level. In the case of non-level coupons, the list takes the form of [(x1, y1), (x2, y2), . . .] Where the x values represent the coupon amounts and the y values represent the lower bound of the interval in which the coupon amount is applicable. For example, [(50, 0), (60, 2)] means that coupon payments of 50 happen in the first two periods and coupon payments of 60 happen afterwards.

Returns A coupon amount, or list of coupon amounts.

Return type float or list

4.10.3 tmval.Bond.get_redemption

`Bond.`**`get_redemption`**`()` → float

Calculates the redemption amount.

Returns The redemption amount.

Return type float

4.10.4 tmval.Bond.get_n_coupons

`Bond.`**`get_n_coupons`**`()` → int

Calculates the number of coupons.

Returns The number of coupons.

Return type int

4.10.5 tmval.Bond.get_coupons

`Bond.`**`get_coupons`**`()` → *tmval.annuity.Annuity*

Calculates the coupons and returns them as an Annuity object.

Returns The bond coupons.

Return type *Annuity*

4.10.6 tmval.Bond.get_coupon_intervals

`Bond.get_coupon_intervals()` → list

Calculates the time intervals between coupon payments.

Returns The time intervals between coupon payments.

Return type list

4.10.7 tmval.Bond.makeham

`Bond.makeham()` → float

Calculates the bond price using Makeham's formula.

Returns The bond price.

Return type float

4.10.8 tmval.Bond.base_amount

`Bond.base_amount()` → float

Calculates the base amount of the bond. The base amount is the investment needed to produce a perpetuity of the coupon payments.

Returns The base amount.

Return type float

4.10.9 tmval.Bond.balance

`Bond.balance`(*t: float*, *n: int = None*) → float

Calculates the bond balance at time t, where t must coincide with a payment time. Alternatively, if you'd like to know the balance just prior to the n-th coupon payment, you may supply the value to the argument n.

Parameters

- **t** (*float*) – The balance time.
- **n** (*int*) – The n-th coupon payment.

Returns The bond balance.

Return type float

4.10.10 tmval.Bond.am_prem

`Bond.am_prem`(*t: float*) → float

Calculates the amortized premium in the last coupon payment prior to time t.

Parameters **t** (*float*) – The valuation time.

Returns The amortized premium.

Return type float

4.10.11 tmval.Bond.acc_disc

`Bond.acc_disc(t: float) → float`

Calculates the accumulation of discount in the last coupon prior to time t.

Parameters **t** (*float*) – The valuation time.

Returns The accumulation of discount.

Return type float

4.10.12 tmval.Bond.am_interest

`Bond.am_interest(t: float) → float`

Calculates the amortized interest in the last coupon prior to time t.

Parameters **t** (*float*) – The valuation time.

Returns The amortized interest.

Return type float

4.10.13 tmval.Bond.amortization

`Bond.amortization() → dict`

Calculates the amortization table for the bond. This method returns a dictionary that can be supplied to a pandas DataFrame.

Returns The amortization table.

Return type dict

4.10.14 tmval.Bond.dirty

`Bond.dirty(t: float, tprac: str = 'theoretical', gr: Union[float, tmval.rate.Rate] = None) → float`

Calculates the dirty value of a bond. It can be toggled to switch between theoretical and practical dirty values.

Parameters

- **t** (*float*) – The valuation time.
- **tprac** (*str*) – Whether you want the practical or theoretical dirty value. Defaults to 'theoretical'.
- **gr** (*float, Rate*) – The valuation yield, if pricing a bond at a different yield. Defaults to the current bond yield.

Returns The dirty value.

Return type float

4.10.15 tmval.Bond.clean

Bond.**clean**(*t: float*, *gr: Union[float,* tmval.rate.Rate*] = None*, *tprac: str = 'theoretical'*) → float

Calculates the clean value. The argument tprac can be toggled to switch between the theoretical, semipractical, or practical clean value.

Parameters

- **t** (*float*) – The valuation time.
- **gr** (*float*, Rate) – The valuation yield, if pricing a bond at a different yield. Defaults to the current bond yield.
- **tprac** (*str*) – Whether you want the 'theoretical', 'semipractical', or 'practical' clean value.

Returns The clean value.

Return type float

4.10.16 tmval.Bond.accrued_interest

Bond.**accrued_interest**(*t: float*, *gr: Union[float,* tmval.rate.Rate*] = None*, *tprac: str = 'theoretical'*) → float

Calculates the accrued interest in a coupon payment. Can be toggled between the clean or practical values.

Parameters

- **t** (*float*) – The valuation time.
- **gr** (*float*, Rate) – The valuation yield, if pricing a bond at a different yield. Defaults to the current bond yield.
- **tprac** (*str*) – Whether you want the 'theoretical' or 'practical' value.

Returns The accrued interest.

Return type float

4.10.17 tmval.Bond.yield_s

Bond.**yield_s**(*t: float*, *sale: float*) → list

Calculates the yield from the perspective of the person to whom you are selling the bond.

Parameters

- **t** (*float*) – The valuation time.
- **sale** (*float*) – The sale price.

Returns The yield to the person to whom you are selling the bond.

Return type list

4.10.18 tmval.Bond.yield_j

`Bond.`**`yield_j`**(*t: float*, *sale: float*) → list

Calculates the yield to the bondholder should they sell the bond at time t.

Parameters

- **`t`** (*`float`*) – The valuation time.
- **`sale`** (*`float`*) – The sale price.

Returns The yield to the bondholder.

Return type list

4.10.19 tmval.Bond.sale_prem

`Bond.`**`sale_prem`**(*t: float*, *gr: Union[float, tmval.rate.Rate]*) → float

Calculates the sale premium.

Parameters

- **`t`** (*`float`*) – The valuation time.
- **`gr`** (*`float, Rate`*) – The valuation yield.

Returns The sale premium.

Return type float

4.10.20 tmval.Bond.last_coupon_amt

`Bond.`**`last_coupon_amt`**(*t: float*) → float

Calculates the amount of the last coupon paid out prior to time t.

Parameters **`t`** (*`float`*) – The valuation time.

Returns The last coupon amount

Return type float

4.10.21 tmval.Bond.next_coupon_amt

`Bond.`**`next_coupon_amt`**(*t: float*) → float

Returns the amount of the next coupon payment after time t.

Parameters **`t`** (*`float`*) – The valuation time.

Returns The next coupon amount.

Return type float

4.10.22 tmval.Bond.last_coupon_t

`Bond.`**`last_coupon_t`**`(`*t: float*`)` → float

Returns the time of the last coupon payment prior to time t.

Parameters **`t`** (*`float`*) – The valuation time.

Returns The time of the last coupon.

Return type float

4.10.23 tmval.Bond.next_coupon_t

`Bond.`**`next_coupon_t`**`(`*t: float*`)` → float

Returns the time of the next coupon payment following time t.

Parameters **`t`** (*`float`*) – The valuation time.

Returns The time of the next coupon.

Return type float

4.10.24 tmval.Bond.coupon_bound_t

`Bond.`**`coupon_bound_t`**`(`*t: float*`)` → tuple

Returns the time boundaries of the last and next coupons around time t.

Parameters **`t`** (*`float`*) – The valuation time.

Returns The coupon time boundaries.

Return type tuple

4.10.25 tmval.Bond.coupon_f

`Bond.`**`coupon_f`**`(`*t: float*`)` → float

Calculates the fraction of a coupon period between time t and the time of the last coupon payment prior to time t.

Parameters **`t`** (*`float`*) – The valuation time.

Returns The fraction of a coupon period.

Return type float

4.10.26 tmval.Bond.adj_principal

`Bond.`**`adj_principal`**`(`*t: float*, *gr: Union[float,* tmval.rate.Rate*] = None*, *tprac: str = 'theoretical'*`)` → float

Calculates the adjustment to principal in the accrued interest for a coupon.

Parameters

- **`t`** (*`float`*) – The valuation time.
- **`gr`** (*`float, Rate`*) – The valuation rate, if different from the yield.
- **`tprac`** (*`str`*) – Whether you want to use 'theoretical' or 'practical' values.

Returns The adjustment to principal in the accrued interest.

Return type float

4.10.27 tmval.Bond.interest_on_accrued

`Bond.`**`interest_on_accrued`**(*t: float*, *gr: Union[float, tmval.rate.Rate] = None*, *tprac: str = 'theoretical'*) → float

Calculates the interest on accrued interest for a coupon.

Parameters

- **`t`** (*`float`*) – The valuation time.
- **`gr`** (*`float`*, `Rate`) – The valuation rate, if different from the yield.
- **`tprac`** (*`str`*) – Whether you want to use 'theoretical' or 'practical' values.

Returns The interest on accrued interest.

Return type float

4.10.28 tmval.Bond.yield_c

`Bond.`**`yield_c`**(*times: list = None*, *premiums: list = None*) → list

Calculates the yields given a list of call times.

Parameters

- **`times`** (*`list`*) – A list of call times.
- **`premiums`** (*`list`*) – A list of call premiums.

Returns Yield rates for the corresponding call times.

Return type list

4.10.29 tmval.Bond.prior_coupons

`Bond.`**`prior_coupons`**(*t: float*) → *tmval.value.Payments*

Gets the coupons prior to the time t.

Parameters **`t`** (*`float`*) – The valuation time.

Returns The coupons prior to the time t.

Return type *Payments*

4.10.30 tmval.Bond.term_floor

`Bond.`**`term_floor`**() → bool

Calculates whether the term provided coincides with bond issuance or a coupon payment.

Returns Whether the term provided coincides with bond issuance or a coupon payment.

Return type bool

4.11 Loan

class tmval.loan.**Loan**(*gr: Union[float,* tmval.rate.Rate*,* tmval.growth.TieredTime*] = None*, *pmt: Union[float, int,* tmval.value.Payments*] = None*, *period: float = None*, *term: float = None*, *amt: float = None*, *cents: bool = False*, *sfr: Union[float,* tmval.rate.Rate*,* tmval.growth.TieredTime*] = None*, *sfd: float = None*, *sf_split: float = 1.0*, *sfh_gr: Union[float,* tmval.rate.Rate*,* tmval.growth.TieredTime*] = None*, *pp: float = None*)

Loan is TmVal's class for representing loans. TmVal currently supports amortized loans, sinking fund loans, amortized/sinking hybrid loans, and payment of fixed principal loans. You can specify the loan type by supplying the appropriate argument.

The default case is an amortized loan, when gr, period, term, and payment amount are supplied without a corresponding sinking fund rate or fixed principal payment. You do not have to supply all of the arguments. If enough of them are supplied, the missing arguments are automatically calculated.

For sinking fund loans, specify the amount of the sinking fund deposit and the sinking fund rate.

For fixed principal loans, the payment is a fixed amount of principal which can be supplied with the argument pp.

Parameters

- **gr** (*float*, Rate, TieredTime) – The interest rate of the loan.
- **period** (*float*) – The time interval between payments, if the loan has regular payments.
- **term** (*float*) – The term of the loan.
- **amt** (*float*) – The loan amount.
- **cents** (*bool*) – Whether you want the payments to be rounded to cents. Defaults to False.
- **sfr** (*float*, Rate, TieredTime) – The sinking fund rate, if different from the loan interest rate.
- **sfd** (*float*) – The sinking fund deposit amount.
- **sf_split** (*float*) – If using a hybrid amortized/sinking loan, the % of the loan that is a sinking fund loan.
- **sfh_gr** (*float*, Rate, TieredTime) – In the case of a hybrid loan, the loan interest rate of the sinking fund portion, if it differs from the amortization rate.
- **pp** (*float*) – The principal payment, in the case of a fixed principal loan.

—

4.11.1 tmval.Loan.get_payments

Loan.**get_payments**() → *tmval.value.Payments*

Takes the arguments supplied and creates the payment schedule for the loan. The return type is a *Payments* object, so it contains the payment times and amounts.

Returns The payment schedule.

Return type *Payments*

4.11.2 tmval.Loan.olb_r

`Loan.olb_r`(*t: float*, *payments: tmval.value.Payments = None*) → float

Calculates the outstanding loan balance at time t, using the retrospective method. If the actual payments differ from the original payment schedule, they may be supplied to the payments argument.

Parameters

- **t** (*float*) – The valuation time.
- **payments** (Payments) – A list of payments, if they differed from the original payment schedule.

Returns The outstanding loan balance.

Return type float

4.11.3 tmval.Loan.olb_p

`Loan.olb_p`(*t: float*, *r: float = None*, *missed: List[int] = None*) → float

Calculates the outstanding loan balance via the prospective method. If there were missed payments, they may be indicated by supplying them to the missed argument.

Parameters

- **t** (*float*) – The valuation time.
- **r** (*float*) – The final payment amount, if different from the others, defaults to None.
- **missed** (*List[int]*) – A list of missed payments, for example, 4th and 5th payments would be [4, 5].

Returns The outstanding loan balance.

Return type float

4.11.4 tmval.Loan.principal_paid

`Loan.principal_paid`(*t2: float*, *t1: float = 0*) → float

Calculates the principal paid between two points in time..

Parameters

- **t2** (*float*) – The second point in time.
- **t1** (*float*) – The first point in time.

Returns The principal paid.

Return type float

4.11.5 tmval.Loan.total_payments

`Loan.`**`total_payments`**(*t2: float*, *t1: float = 0*) → float

Calculates the total loan payments that occurred between two points in time.

Parameters

- **`t2`** (*`float`*) – The second point in time.
- **`t1`** (*`float`*) – The first point in time.

Returns The total loan payments.

Return type float

4.11.6 tmval.Loan.interest_paid

`Loan.`**`interest_paid`**(*t2: float*, *t1: float = 0*, *frac: bool = False*) → float

Calculates the total interest paid between two points in time.

Parameters

- **`t2`** (*`float`*) – The second point in time.
- **`t1`** (*`float`*) – The first point in time.
- **`frac`** (*`bool`*) – Whether you want the answer as a fraction of the total interest of the loan.

Returns The total interest paid.

Return type float

4.11.7 tmval.Loan.principal_val

`Loan.`**`principal_val`**(*t: float*) → float

Calculates the time-value adjusted principal at a desired point in time.

Parameters **`t`** (*`float`*) – The valuation time.

Returns The time-value adjusted principal.

Return type float

4.11.8 tmval.Loan.amortization

`Loan.`**`amortization`**() → dict

Calculates the amortization table based off the original payment schedule. Returned as a dict which can be supplied to a pandas DataFrame for further analysis and viewing.

Returns The amortization table.

Return type dict

4.11.9 tmval.Loan.sf_final

`Loan.`**`sf_final`**`(`*payments:* tmval.value.Payments = *None*`)` → float

Calculates the final payment required to settle a sinking fund loan. You may supply payments, if they differed from the original payment schedule.

Parameters **`payments`** (`Payments`, *`optional`*) – A list of payments, if different from the original payment schedule.

Returns The final sinking fund payment.

Return type float

4.11.10 tmval.Loan.sink_payments

`Loan.`**`sink_payments`**`(`*payments:* tmval.value.Payments`)` → dict

Calculates a sinking fund schedule for a list of payments. Intended to be called externally only if the payments differ from the original sinking fund schedule. Otherwise, use the *`sinking()`* method. The return type is a dict which can be passed to a pandas DataFrame for further analysis and viewing.

Parameters **`payments`** (`Payments`) – A list of payments.

Returns The sinking fund schedule for the payments.

Return type dict

4.11.11 tmval.Loan.sinking

`Loan.`**`sinking`**`()` → dict

Calculates the sinking fund schedule based off the original payment schedule. The return type is a dict which can be passed to a pandas DataFrame for further analysis and viewing.

Returns The sinking fund schedule.

Return type dict

4.11.12 tmval.Loan.rc_yield

`Loan.`**`rc_yield`**`()` → list

Calculates the yield rate based off replacement of capital.

Returns The yield rate based off replacement of capital.

Return type list

4.11.13 tmval.Loan.fixed_principal

`Loan.`**`fixed_principal`**`()` → *tmval.value.Payments*

Calculates the loan payment schedule if a fixed amount of principal is paid each period.

Returns The payment schedule.

Return type *Payments*

4.11.14 tmval.Loan.hybrid_principal

`Loan.hybrid_principal()` → float

Calculates the loan amount based on a hybrid amortized/sinking fund loan.

Returns The loan amount.

Return type float

4.11.15 tmval.Loan.sgr_equiv

`Loan.sgr_equiv()` → *tmval.rate.Rate*

Calculates the sinking fund rate such that would produce a loan payment schedule equivalent to that of an amortized loan.

Returns The sinking fund rate.

Return type *Rate*

CHAPTER

FIVE

NOTATION GUIDE

TmVal is intended to provide a more natural connection to actuarial notation than other time value of money packages from general areas of finance. By providing the main actuarial financial instruments as classes, TmVal makes it easier to represent and manipulate them in ways that more easily match actuarial theory.

This guide shows how common actuarial symbols can be replicated in TmVal. These are written in pseudocode and not meant to be executed directly, and serve as a guide as to what classes and functions correspond to which symbols.

Amount function: $A_K(t)$

```
Amount(gr, k).val(t)
```

Accumulation function: $a(t)$

```
Accumulation(gr).val(t)
```

Effective interest rate for the interval: $i_{[t_1,t_2]} = \frac{a(t_2)-a(t_1)}{a(t_1)}$

```
Accumulation(gr).effective_interval(t1, t2)
```

or, assuming proportionality:

```
Amount(gr, k).effective_interval(t1, t2)
```

Effective interest for the n-th time period: $i_n = \frac{a(n)-a(n-1)}{a(n-1)}$

```
Amount(gr, k).effective_rate(n)
```

Simple interest amount function: $A_K(t) = K(1+st)$

```
Amount(gr=Rate(s), k).val(t)
```

Compound interest accumulation function: $a(t) = (1+i)^t$

```
Accumulation(gr=Rate(i)).val(t)
```

or

```
Accumulation(gr=i).val(t)
```

Effective discount rate for the interval: $d_{[t1,t2]} = \frac{a(t_2)-a(t_1)}{a(t_2)}$

```
Accumulation(gr).discount_interval(t1, t2)
```

or, assuming proportionality:

```
Amount(gr, k).discount_interval(t1, t2)
```

Effective discount rate for the n-th time period: $d_n = \frac{a(n)-a(n-1)}{a(n)}$

```
Accumulation(gr).effective_discount(n)
```

Discount function: $v(t) = \frac{1}{a(t)}$

```
Accumulation(gr).discount_func(t)
```

Future principal: $Sv(t_2)a(t_1) = S\frac{a(t_1)}{a(t_2)} = S\frac{v(t_2)}{v(t_1)}$

```
Accumulation(gr).future_principal(fv=S, t1, t2)
```

Simple discount amount function: $A_K(t) = \frac{K}{(1-dt)}$

```
Amount(gr=Rate(sd), k).val(t)
```

Simple discount accumulation function: $a(t) = \frac{1}{(1-dt)}$

```
Accumulation(gr=Rate(sd), k).val(t)
```

Nominal interest rate of $i^{(m)}$ convertible or compounded or payable m times per year

```
Rate(
    rate,
    pattern="Nominal Interest",
    freq=m)
```

Nominal discount rate $d^{(m)}$ convertible or compounded or payable m times per year

```
Rate(
    rate,
    pattern="Nominal Discount",
    freq=m)
```

Force of interest: $\delta = \lim_{m\to\infty} i^{(m)} = \ln(1+i)$

```
Rate(delta)
```

Accumulation function under the force of interest: $a(t) = e^{\delta t}$

```
Accumulation(gr=Rate(delta))
```

Time τ equation of value: $\sum_k C_{t_k}\frac{a(\tau)}{a(t_k)} = B\frac{a(\tau)}{a(T)}$

```
Payments(amounts, times, gr).eq_val(t)
```

Equated time: $T = \frac{\ln\left(\frac{\sum_{k=1}^{n} C_{t_k} v^{t_k}}{C}\right)}{\ln v}$

```
Payments(amounts, times, gr).equated_time(c=C)
```

Approximate dollar-weighted yield, k=1/2: $j \approx \frac{2I}{A+B-I}$

```
Payments(amounts, times, gr).dollar_weighted_yield(k_approx=True)
```

Annual time-weighted yield rate: $i_{tw} = (1 + j_{tw})^{\frac{1}{T}} - 1 = \left[\prod_{k=1}^{r+1}(1 + j_k)\right]^{\frac{1}{T}} - 1$

```
Payments(amounts, times, gr).time_weighted_yield()
```

Present value of basic annuity-immediate: $a_{\overline{n}|i}$

```
Annuity(gr=i, n).pv()
```

Accumulated value of basic annuity-immediate: $s_{\overline{n}|i}$

```
Annuity(gr=i, n).sv()
```

Loan payment, level: $Q = \frac{L}{a_{\overline{n}|i}}$

```
get_loan_pmt(
    loan_amt=L,
    period,
    term,
    gr=Rate(i))
```

Savings payment to obtain accumulated balance: $Q = \frac{B}{s_{\overline{n}|i}}$

```
get_savings_pmt(
    fv=B,
    period,
    term,
    gr=Rate(i)
)
```

Present value of basic annuity-due: $\ddot{a}_{\overline{n}|i}$

```
Annuity(
    gr=i,
    n=n,
    imd='due'
).pv()
```

Accumulated value of basic annuity-due: $\ddot{s}_{\overline{n}|i}$

```
Annuity(
    gr=i,
    n=n,
    imd='due'
).sv()
```

Present value of basic perpetuity-immediate: $a_{\overline{\infty}|i}$

```
Annuity(
    gr=i,
    term=np.Inf,
).pv()
```

Present value of basic perpetuity-due: $\ddot{a}_{\overline{\infty}|i}$

```
Annuity(
    gr=i,
    term=np.Inf,
```

(continues on next page)

(continued from previous page)

```
    imd='due'
).sv()
```

Present value of deferred annuity-immediate: $_{w|n}a$

```
Annuity(
    gr,
    n=n,
    deferral=w
).pv()
```

Present value of deferred annuity-due: $_{w|n}\ddot{a}$

```
Annuity(
    gr,
    n=n,
    deferral=w,
    imd='due'
).pv()
```

Outstanding loan balance, retrospective method: $\text{OLB}_k = La(k) - Qs_{\overline{k}|}$

```
Loan(
    amt=L,
    pmt=Q,
    gr,
    period,
    term
).olb_r()
```

Outstanding loan balance, prospective method (adjusted final payment): $\text{OLB}_k = Qa_{\overline{n-k-1}|i} + R(1+i)^{-(n-k)}$

```
Loan(
    amt=L,
    pmt=Q,
    gr,
    period,
    term
).olb_p(t, r)
```

Outstanding loan balance, prospective method (equal payments): $\text{OLB}_k = Qa_{\overline{n-k}|i}$

```
Loan(
    amt=L,
    pmt=Q,
    gr,
    period,
    term
).olb_p(t)
```

Present value of an annuity-immediate with geometrically increasing payments: $P\left(\frac{1-\left(\frac{1+g}{1+i}\right)^n}{i-g}\right)$

```
Annuity(
    amount=P,
    n=n,
    gr=Rate(i),
```

(continues on next page)

(continued from previous page)

```
    grog=g
).pv()
```

Present value of annuity-immediate with arithmetically increasing payments: $(I_{P,Q}a)_{\overline{n}|i}$

```
Annuity(
    amount=P,
    n=n,
    gr=Rate(i)
    aprog=Q
).pv()
```

Accumulated value of annuity-immediate with arithmetically increasing payments: $(I_{P,Q}s)_{\overline{n}|i}$

```
Annuity(
    amount=P,
    n=n,
    gr=Rate(i)
    aprog=Q
).sv()
```

Present value of annuity-immediate with arithmetically decreasing payments: $(Da)_{\overline{n}|i}$

```
 Annuity(
    amount=n,
    n=n,
    gr=Rate(i)
    aprog=-1
).pv()
```

Accumulated value of annuity-immediate with arithmetically decreasing payments: $(Ds)_{\overline{n}|i}$

```
 Annuity(
    amount=n,
    n=n,
    gr=Rate(i)
    aprog=-1
).sv()
```

Present value of annuity-due with arithmetically increasing payments: $(I_{P,Q}\ddot{a})_{\overline{n}|i}$

```
Annuity(
    amount=P,
    n=n,
    gr=Rate(i)
    aprog=Q,
    imd='due'
).pv()
```

Accumulated value of annuity-due with arithmetically increasing payments: $(I_{P,Q}\ddot{s})_{\overline{n}|i}$

```
Annuity(
    amount=P,
    n=n,
    gr=Rate(i)
    aprog=Q,
```

(continues on next page)

(continued from previous page)

```
    imd='due'
).sv()
```

Present value of annuity-due with arithmetically decreasing payments: $(D\ddot{a})_{\overline{n}|i}$

```
 Annuity(
    amount=n,
    n=n,
    gr=Rate(i)
    aprog=-1,
    imd='due'
).pv()
```

Accumulated value of annuity-due with arithmetically decreasing payments: $(D\ddot{s})_{\overline{n}|i}$

```
 Annuity(
    amount=n,
    n=n,
    gr=Rate(i),
    aprog=-1,
    imd='due'
).sv()
```

Present value of perpetuity-immediate with arithmetically increasing payments: $(I_{P,Q}a)_{\overline{\infty}|i}$

```
 Annuity(
    amount=P,
    term=np.Inf,
    gr=Rate(i),
    aprog=Q
).pv()
```

Present value of perpetuity-due with arithmetically increasing payments: $(I_{P,Q}\ddot{a})_{\overline{\infty}|i}$

```
 Annuity(
    amount=P,
    term=np.Inf,
    gr=Rate(i)
    aprog=Q,
    imd='due'
).pv()
```

Present value of annuity-immediate with payments more frequent than each interest period: $a^{(m)}_{\overline{n}|i}$

```
Annuity(
    amount=1/m,
    term=n,
    gr=Rate(i),
    period=1/m
).pv()
```

Accumulated value of annuity-immediate with payments more frequent than each interest period: $s^{(m)}_{\overline{n}|i}$

```
Annuity(
    amount=1/m,
    term=n,
```

(continues on next page)

(continued from previous page)

```
    gr=Rate(i),
    period=1/m
).sv()
```

Present value of annuity-due with payments more frequent than each interest period: $\ddot{a}_{\overline{n}|i}^{(m)}$

```
Annuity(
    amount=1/m,
    term=n,
    gr=Rate(i),
    period=1/m,
    imd='due'
).pv()
```

Accumulated value of annuity-due with payments more frequent than each interest period: $\ddot{s}_{\overline{n}|i}^{(m)}$

```
Annuity(
    amount=1/m,
    term=n,
    gr=Rate(i),
    period=1/m,
    imd='due'
).sv()
```

Present value of perpetuity-immediate with payments more frequent than each interest period: $a_{\overline{\infty}|i}^{(m)}$

```
Annuity(
    amount=1/m,
    term=np.Inf,
    gr=Rate(i),
    period=1/m
).pv()
```

Present value of perpetuity-due with payments more frequent than each interest period: $\ddot{s}_{\overline{\infty}|i}^{(m)}$

```
Annuity(
    amount=1/m,
    term=np.Inf,
    gr=Rate(i),
    period=1/m,
    imd='due'
).pv()
```

CHAPTER

SIX

GLOSSARY

accumulation function A function that describes how much as single unit of currency grows over time. It is a special case of the amount function, where the amount invested is restricted to be one unit of currency.

amount function A function that describes how much an invested amount of money grows over time.

annual percentage rate (APR) The nominal interest rate.

annual percentage yield (APY) The annual effective interest rate.

annuity A series of payments made at specified intervals for a fixed or contingent period.

annuity-due An annuity in which the payments occur at the beginning of each payment period.

annuity-immediate An annuity in which the payments occur at the end of each payment period.

balloon payment An adjustment to the value of the final payment of a loan so that it can occur on the previous integral payment period.

basic annuity-due An annuity-due that pays 1 at the beginning of each period.

basic annuity-immediate An annuity-immediate that pays 1 at the end of each period.

basic perpetuity-due A perpetuity-due that pays 1 at the beginning of each period.

basic perpetuity-immediate A perpetuity-immediate that pays 1 at the end of each period.

bond A financial instrument that entities, called bond issuers, can use to borrow money from other entities, called bondholders.

compound interest A geometric pattern of money growth in which interest earned is reinvested at the rate of interest.

deferred annuity A type of annuity whose first payment begins more than one payment period later than its present valuation date.

discount The amount of money that must be paid up front on a loan.

discount function The reciprocal of the accumulation function:

$$v(t) = \frac{1}{a(t)}$$

dollar-weighted yield rate See *yield rate*

drop payment An adjustment to the amount of the final payment of a loan so that it can occur on the next integral payment period.

effective rate of interest A measurement of money growth equal to the percentage change in the value of an investment between two time periods:

$$i_{[t_1,t_2]} = \frac{a(t_2) - a(t_1)}{a(t_1)}$$

force of interest The limit of the nominal interest or discount rate as the compounding frequency approaches infinity.

growth The change in the value of money over time.

interest earned A measurement of money growth equal to the change in the value of an investment between two time periods:

$$A_K(t_2) - A_K(t_1).$$

internal rate of return See *yield rate*

modified coupon rate The bond coupon rate expressed in terms of the redemption amount.

net present value (NPV) The sum of the present value of a stream of returns:

$$\sum_{k=0}^{n} R_k v(t_k)$$

nominal discount rate The discount rate $d^{(m)}$ compounded m times per year.

nominal interest rate The interest rate $i^{(m)}$ compounded m times per year.

retrospective method A way of calculating the outstanding loan balance at time t by subtracting the accumulated value of the payments to date from the accumulated value of the principal.

par-value bond A bond whose face value equals its redemption amount.

perpetuity An annuity with an infinite number of payments

perpetuity-due A perpetuity that makes payments at the beginning of each period.

perpetuity-immediate A perpetuity that makes payments at the end of each period.

principal

(1) An initial investment of money.

(2) The original amount of a loan that must be paid back.

prospective method A method of calculating the outstanding loan balance at a point in time t that sums up the remaining payments and discounts them to time t.

present value The value today of money to be received in the future.

simple interest A linear pattern of money growth in which interest earned is a fixed amount per time period.

yield rate The rate that solves the time τ equation of value.

CHAPTER

SEVEN

REFERENCES

1. David Beauchemin and Vincent Goulet, Actuarial symbols of life contingencies and financial mathematics, École d'actuariat, Université Laval, (2019).
2. Travis E, Oliphant. A guide to NumPy, USA: Trelgol Publishing, (2006).
3. Leslie Vaaler and James Daniel, Mathematical Interest Theory, Washington, D.C.: Mathematical Association of America, (2009).
4. Pauli Virtanen, Ralf Gommers, Travis E. Oliphant, Matt Haberland, Tyler Reddy, David Cournapeau, Evgeni Burovski, Pearu Peterson, Warren Weckesser, Jonathan Bright, Stéfan J. van der Walt, Matthew Brett, Joshua Wilson, K. Jarrod Millman, Nikolay Mayorov, Andrew R. J. Nelson, Eric Jones, Robert Kern, Eric Larson, CJ Carey, İlhan Polat, Yu Feng, Eric W. Moore, Jake VanderPlas, Denis Laxalde, Josef Perktold, Robert Cimrman, Ian Henriksen, E.A. Quintero, Charles R Harris, Anne M. Archibald, Antônio H. Ribeiro, Fabian Pedregosa, Paul van Mulbregt, and SciPy 1.0 Contributors. (2020) SciPy 1.0: Fundamental Algorithms for Scientific Computing in Python. Nature Methods, in press.

CHAPTER

EIGHT

PDF VERSION

TmVal Documentation

INDEX