USER GUIDE FOR

Enginomic$

A Python package for engineering economic decision-making.

Version 0.8.4
June 4, 2025

Table of Contents

1 License and Terms

`Enginomics` may be used under the Apache License 2.0. A copy of the license is provided in Appendix A of this document.

Within the terms of the Apache License 2.0, you are free to take the source code, change it, and re-distribute it. However, you may **not** call any derivative product ”Enginomics”, nor use the word ”Enginomics” in the derivative product’s name. This applies regardless of the use of upper-case or lower-case letters.

2 About Enginomics

The purpose of Enginomics is to provide utilities for the manipulation and analysis of cash flow series. It is designed with engineering economic analysis in mind, but is not limited to use with or for engineering projects. Enginomics will also amortize loans.

Why did I write Enginomics? I am a (much) older student enrolled in Mississippi State University's online BS in Industrial Engineering program. I recently completed a course in engineering economics, a key course for industrial engineering students. (I entered the BSIE program already having earned a degree in accounting. I still learned a ton.) I have arthritic hands, so it was easier to type up my homework in LaTeX than to write it by hand. The one problem: cash flow diagrams. I ended up graphing those in Excel, saving the graphic, and embedding into my LaTeX document. I quickly realized that I could also use Excel to check my work. While I was a "good boy" and did all my work by hand using the interest tables, I used Excel to confirm my answers. I would have lost several points had I not checked my work in this way.

However, even using Excel was cumbersome. I realized that I could use Python to do many of these tasks automatically. A search of the Python package repository did not lead me to any packages that did what I wanted Python to do, so I wrote the software myself, and decided to share it.

Bug reports, suggestions, and requests for features may all be sent to enginomics ∂ engihelp $\odot$ net.

Regards,
Wayne Matthew Syvinski
May 17, 2025

Glory to God for all things!

3 Installation and Required Libraries

The following libraries are required to use `Enginomics`:

- `matplotlib`
- `numpy`
- `numpy-financial`
- `pandas`
- `pandasql`
- `seaborn`

The analyst may also have to import `copy` and `math`, but these are part of the Python Standard Library.

Dependencies will be installed if using `pip`:

- `pip install Enginomics`

If using a `conda` environment (such as *Anaconda*), please consult the appropriate documentation.

4 Class Cashflow

4.1 About

NOTE!

Class `Cashflow` is the heart of the `Enginomics` package.

It allows cash flow series to be entered, analyzed, and graphed easily. It also facilitates quick and easy pairwise incremental analysis when used with classes `IncrementalIRR` and `IncrementalBCR`.

WARNING!

Note that class `Cashflow` has **no notion** of years, quarters, months, or any other specific time span. There are only compounding periods. For example, if the analyst wants to use monthly compounding, the nominal APR must be divided by 12 to get the appropriate interest rate for the compounding period.

4.2 How to Use

4.2.1 Import Statement

```
from Enginomics import Cashflow
```

4.2.2 Constructors and Quasi-Constructors

While there is a default constructor as required by Python, it is **strongly** recommended that analysts do not use the default constructor. Rather, there are static methods that serve as quasi-constructors that are **far** easier to use.

Quasi-Constructor `from_list`

Arguments: `(cflist: list, rate: float = None)`

This quasi-constructor takes a list representing a cash flow sequence and creates a `Cashflow` object. Compounding periods may not be skipped. If you have a specific compounding period with no cash flow, you must enter a `0`. In addition, the list is always processed as starting from period 0.

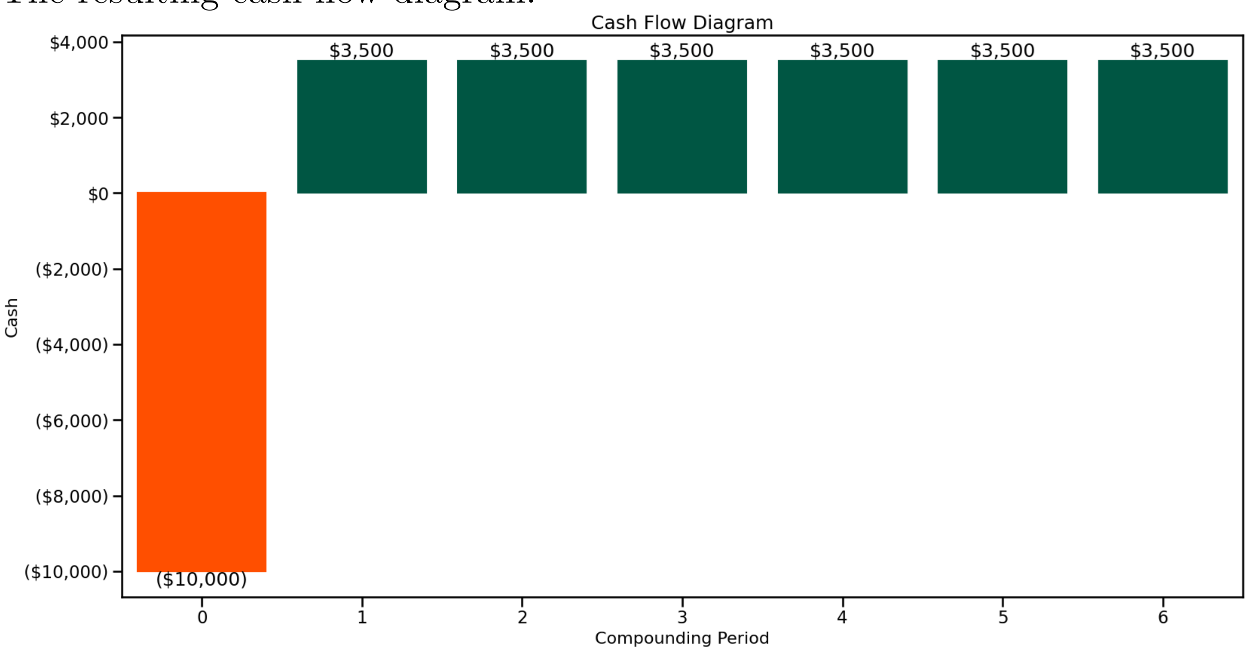
Optionally, the analyst can include the discount rate for the cash flow series, but if not specified here, it will need to be specified using method `set_rate()`.

The above paragraph is true for all quasi-constructors, so this information will not be repeated.

Example: For a project with initial cost of $10,000 yielding yearly revenue of $3,500 for six years:

```
cfs = Cashflow.from_list([-10000, 3500, 3500, 3500, 3500, 3500, 3500])
```

The resulting cash flow diagram:



Quasi-Constructor `from_dict`

Arguments: `(cfdict: dict, rate: float = None)`

This quasi-constructor takes a dictionary representing a cash flow sequence and creates a `Cashflow` object. The compounding periods are the keys, and the cash flows are the values.

Example, which results in the same cash flow diagram as above: `cfs = Cashflow.from_dict({0: -10000, 1: 3500, 2: 3500, 3: 3500,` `4: 3500, 5: 3500, 6: 3500})`

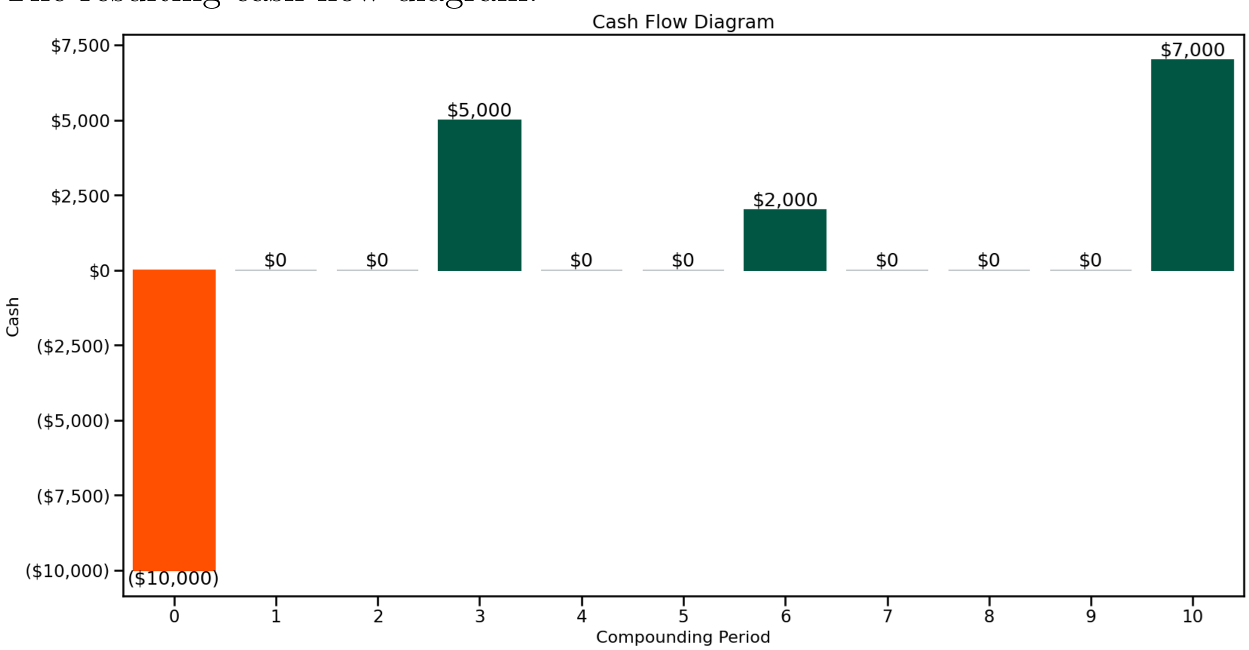
However, this quasi-constructor allows you to have gaps in your cash flow series. In this case, `Enginomics` will insert the missing compounding periods. This feature is called *autovivification*. Missing compounding periods are autovivified with a cash value of 0.

NOTE!

Make sure you understand the concept of *autovivification*. It is used throughout `Enginomics`, and will be referenced often in this document.

Example: `cfs = Cashflow.from_dict({0: -10000, 3: 5000, 6:2000, 10:7000})`

The resulting cash flow diagram:



Quasi-Constructor `expansion_from_dict`

Arguments: `(cfdict: dict, rate: float = None)`

This quasi-constructor takes a dictionary input, but instead of autovivifying missing entries with a zero cash value, it autovivifies missing entries by looking at the latest defined value in the cash flow and repeating it until it gets to the next defined value.

Example: A project has an initial investment of $15,000. In years 1 through 3, the return is a uniform annual series of $3,000. In years 4 through 7, the return is a uniform annual series of $8,000. Finally, in years 8 through 12, the return is an annual series of $6,500.

If using the `from_list()` quasi-constructor, you would have to pass every cash flow explicitly:

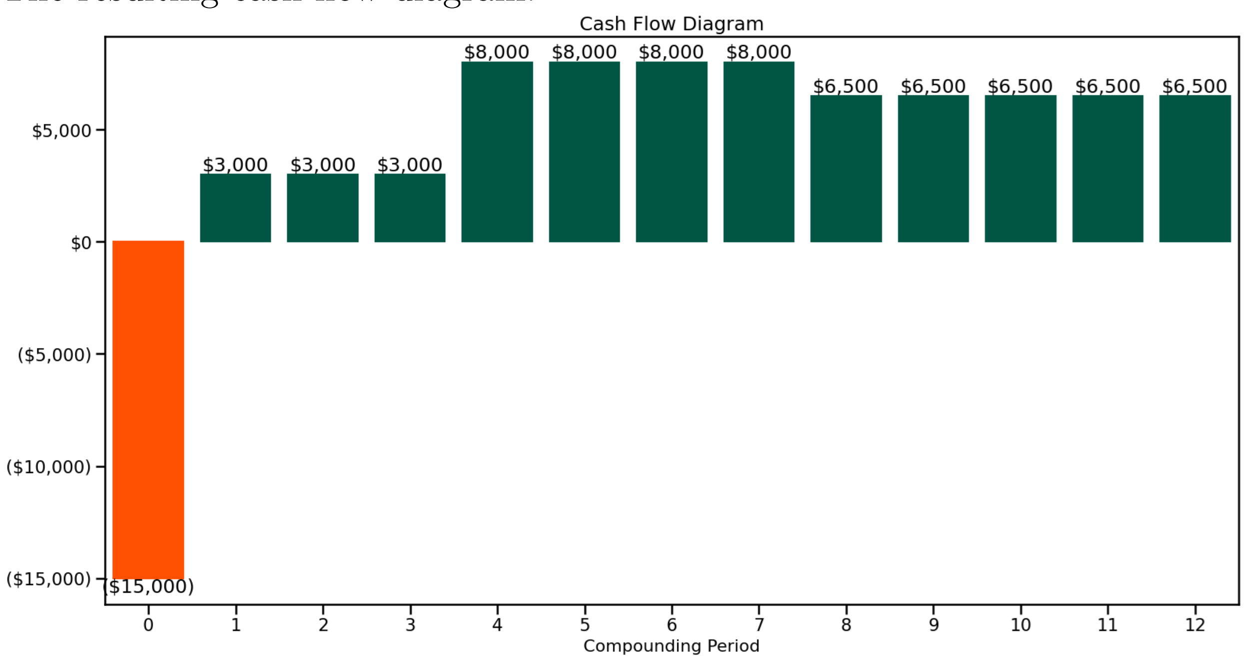
```
[-15000, 3000, 3000, 3000, 8000, 8000, 8000, 8000,
6500, 6500, 6500, 6500, 6500]
```

Using `expansion_from_dict()`, you can express this more compactly:

```
cfd = Cashflow.expansion_from_dict({0: -15000, 1: 3000, 4: 8000,
8: 6500, 12: 6500})
```

Note that you have to terminate the series explicitly, which is why the value `6500` is given twice. The last entry in the dictionary tells `Enginomics` when to stop autovivifying entries. The last entry need not be the same as the immediately preceding defined entry; it could, among other things, reflect the salvage value at the end of the project.

The resulting cash flow diagram:



Quasi-Constructor `from_annuity`

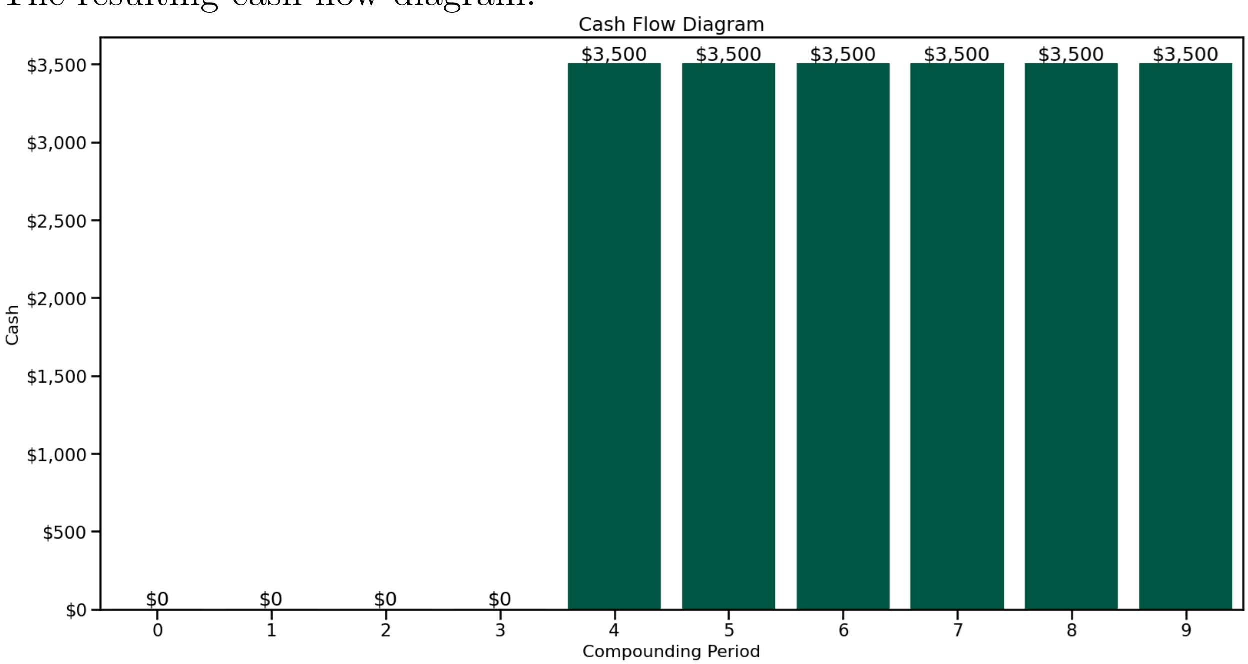
Arguments: `(start_period: int, end_period: int, amount: float, rate: float = None)`

This quasi-constructor generates a uniform annual series based on a defined starting compounding period, an ending compounding period, and an annual amount. Any compounding periods that come before the defined starting compounding period are autovivified with a cash value of 0.

Example: Model a uniform series from year 4 to year 9 with a value of $3,500 per year.

```
cfd = Cashflow.from_annuity(start_period = 4, end_period = 9, amount = 3500)
```

The resulting cash flow diagram:



Quasi-Constructor `from_arith`

Arguments: `(start_period: int, end_period: int, amount: float, rate: float = None)`

This quasi-constructor generates an arithmetic gradient based on a defined starting compounding period, an ending compounding period, and an amount of annual increase. Any compounding periods that come before the defined starting compounding period are autovivified with a cash value of 0.

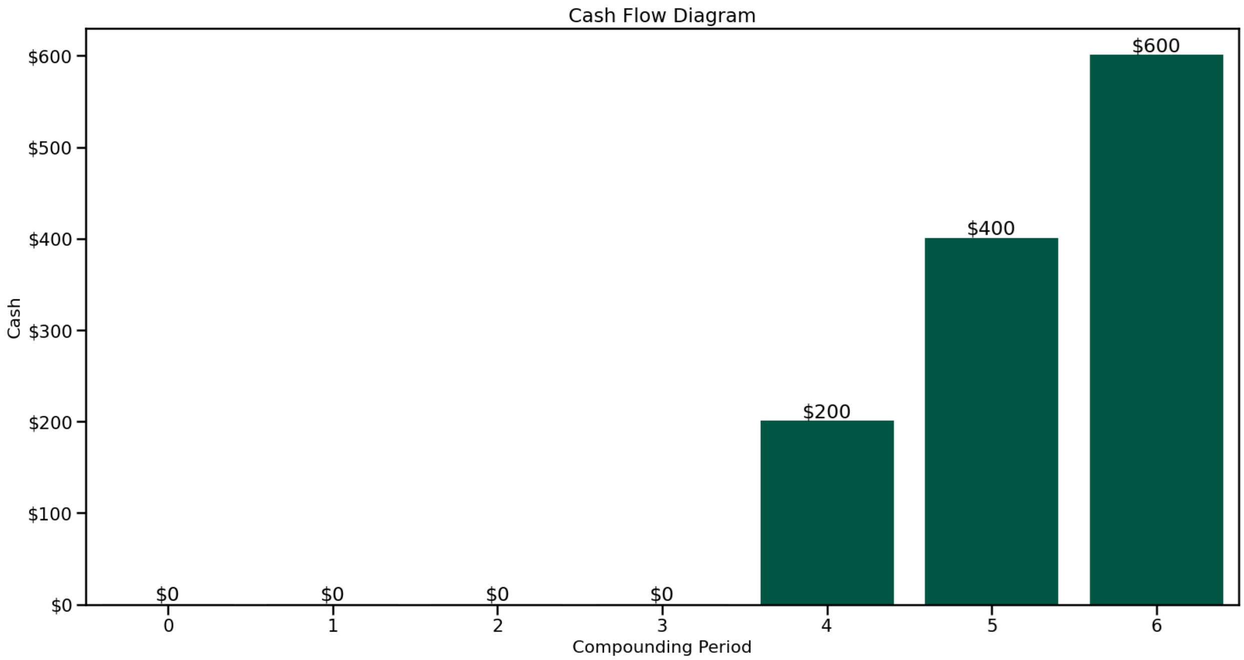
Example: Model an arithmetic gradient from year 3 to year 6 with a value of $200 per year.

NOTE!

Remember: the first compounding period of an arithmetic gradient has a cash value of zero!

```
cfd = Cashflow.from_arith(start_period = 3, end_period = 6, amount
= 200)
```

The resulting cash flow diagram:



Quasi-Constructor `from_geom`

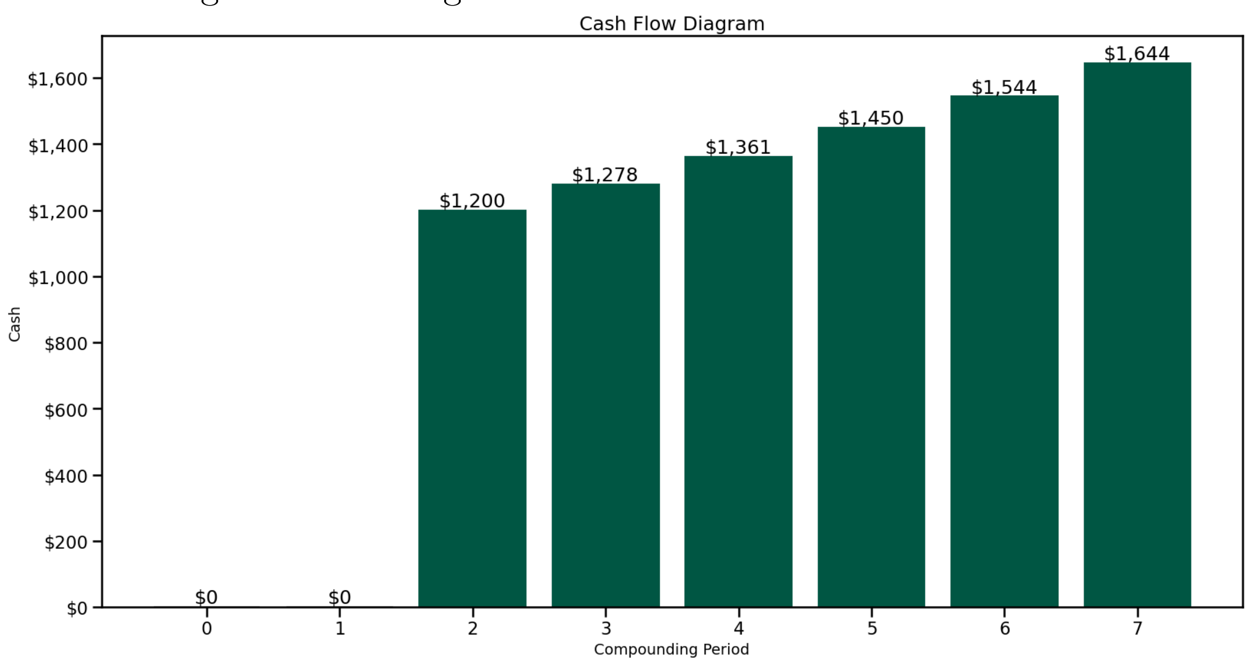
Arguments: `(start_period: int, end_period: int, amount: float, pct_change: float, rate: float = None)`

This quasi-constructor generates a geometric gradient based on a defined starting compounding period, an ending compounding period, a starting amount, and the period-over-period percentage of annual increase. Any compounding periods that come before the defined starting compounding period are autovivified with a cash value of 0. The percent change must be expressed as a decimal fraction (e.g. 8% is given as `0.08`), but you can express this as a percentage value if you divide by 100 in the quasi-constructor call (e.g. 8% given as `8.0/100.0` or `float(8)/float(100)`)

Example: Model a cash flow that has an initial value of $1,200, with a year-over-year increase of 6.5%, starting at year 2 and ending at year 7.

```
cfd = Cashflow.from_geom(start_period = 2, end_period = 7, amount
= 1200, pct_change = 6.5/100.0)
```

The resulting cash flow diagram:



4.2.3 SPECIAL NOTE: Cash Flow Time Horizons

This concept will come up later in this documentation, so it is important to understand it now. Therefore, it is being presented here rather than in the methods section of the class documentation

NOTE!

Two cash flow series have the same *horizon* **if and only if**:

1. They each start at the same compounding period
2. They each end at the same compounding period
3. They have the same number of compounding periods (i.e. there are no missing compounding periods in the sequence)

It is not necessary for the cash values in each compounding period to be equal for each cash flow series.

There is a method defined to test whether two cash flow series have the same horizon: `same_horizon(cfd: 'Cashflow')`

Example:

Benefits (income) cash flow:
`cf_ben = Cashflow.from_list([0, 5000, 7000, 8000, 6000, 2700])`

Costs cash flow:
`cf_cost = Cashflow.from_list([-10000, -1000, -1000, -3500])`

`horizon_test = cf_ben.same_horizon(cf_cost)`

This can be read as, "For cash flow series `cf_ben`, check if it has the same horizon as cash flow series `cf_cost`."

`print(horizon_test)`

`> False`

4.2.4 Overloaded Operators

The following operators are overloaded for use by class `Cashflow`:

- addition: `cashflow1 + cashflow2`
- subtraction: `cashflow2 - cashflow2`
- multiplication: `cashflow * number` *or* `number * cashflow`
- division: `cashflow / number`

- unary negation: `-cashflow`

NOTE!

Addition and subtraction do not require cash flows to have the same horizon. This is to allow building up net cash flows from individual component cash flows.

WARNING!

`Cashflow` objects created using overloaded operators will not have a discount rate set. The analyst will have to assign it to the result using the `set_rate()` property method.

Examples:

Benefits (income) cash flow:
`cf_ben = Cashflow.from_list([0, 5000, 7000, 8000, 6000, 2700])`

Costs cash flow:
`cf_cost = Cashflow.from_list([-10000, -1000, -1000, -3500, -500, -750])`

Net cash flow: `cf_net = cf_ben + cf_cost`

Result: `print((cf_cost + cf_ben).to_list())`

`> [-10000, 4000, 6000, 4500, 6000.0, 2700.0]`

The other overloaded operators work in a similar, intuitive manner.

4.2.5 Property Methods

These are not true properties in the sense that using Python decorators would provide. This is a deliberate design decision. However, these methods do act as property getters and setters.

Property Method `get_rate`
Arguments: none

Retrieve the discount rate for the `Cashflow` object. This is a decimal fraction; e.g. a discount rate of 7.5% will be returned as `0.075`.

Property Method `set_rate`
Arguments: `(rate: float)`

Set the discount rate for the `Cashflow` object. This is a decimal fraction; e.g. a discount rate of 6.25% must be entered as `0.0625`.

Property Method `get_period`
Arguments: `(period: int)`

Retrieve the cash value of the `Cashflow` object at the specified compounding period.

Property Method `set_period`
Arguments: `(period: int, amount: float = 0.0)`

Set the cash value of the `Cashflow` object at the specified compounding period using the value provided with argument `amount`. The default is 0. If the compounding period does not exist in the `Cashflow` object, it will be autovivified. If the compounding period indicated will create gaps in the cash flow series, the intervening compounding periods will be autovivified with a cash value of 0.

4.2.6 Text Processing Methods for Cash Flow Definition Strings

Introduction

It is not necessary to use the quasi-constructor methods to create `Cashflow` objects. `Enginomics` can parse *cash flow definition strings* to generate them; in fact, it can read from text files containing many cash flow definition strings.

Conversely, `Enginomics` can write cash flow definition strings to a text file to save them for later use.

Structure of Cash Flow Definition Strings

Here is an example of the contents of a text file holding several cash flow definition strings:

```
Project A|list|0.08|-12000,3000,4000,5000,6000
Project B|dict|0.07|0:-10000,1:3400,5:2800,10:7800
Project C|expand|0.065|0:-10000,1:3400,5:2800,10:7800
Project D|arith|0.10|5,12,1000
Project E|geom||3,10,400,0.075
Project F$C$|list||-10000,-1000,-1000,-5500,-1000,-500
Project F$B$|list||0,4000,4000,5000,5000,8500
Project G|annuity|0.115|4,9,3500
```

Each line is pipe-delimited into four fields in this order:

- name of cash flow
- parsing method
- discount rate
- cash flow definition details

Every cash flow definition string **must** have a name; it cannot be blank. This is because `Cashflow` does not return bare `Cashflow` objects when parsing cash flow definition strings, but a dictionary where the cashflow name is the key, and the `Cashflow` object is the value.

Please make special note of the entries for Project F. Project F is listed twice, but each entry has a different suffix. Suffix `$C$` is used to denote the cash flow of costs for a project, and suffix `$B$` is used to denote the cash flow of benefits for the same project. This will be discussed in detail in the documentation for class `IncrementalBCR`, which is used for incremental benefit-cost ratio analysis.

Each parsing method corresponds to one of the quasi-constructor methods defined in class `Cashflow`. If dictionary `cashflow` is the target to receive the result of parsing:

- parsing the entry for Project A, which uses parsing method `list`, is equivalent to:
  `cashflow[‘Project A’] =`
  `Cashflow.from_list(-12000,3000,4000,5000,6000), 0.08)`

- parsing the entry for Project B, which uses parsing method `dict`, is equivalent to:
  `cashflow['Project B'] = Cashflow.from_dict({0:-10000,1:3400,5:2800,10:7800}, 0.08)`

- parsing the entry for Project C, which uses parsing method `expand`, is equivalent to:
  `cashflow['Project C'] = Cashflow.expansion_from_dict({0:-10000,1:3400,5:2800,10:7800}, 0.08)`

- parsing the entry for Project D, which uses parsing method `arith`, is equivalent to:
  `cashflow['Project D'] = Cashflow.from_arith(start_period = 5, end_period = 12, amount=1000, rate=0.08)`

- parsing the entry for Project E, which uses parsing method `geom`, is equivalent to:
  `cashflow['Project E'] = Cashflow.from_geom(start_period = 5, end_period = 12, amount=1000, pct_change = 0.075, rate=0.08)`

- parsing the entry for Project G, which uses parsing method `annuity`, is equivalent to:
  `cashflow['Project G'] = Cashflow.from_annuity(start_period = 4, end_period = 9, amount=3500, rate=0.115)`

Including the discount rate in the cash flow definition string is optional, but if not included there, it must be specified later using the `set_rate()` method. The fourth column contains the specification for building the `Cashflow` object. Based on the previous discussion of parsing methods, the structures should be easily understandable.

Method `make_entry_text`
Arguments: `(entry_name: str)`

Create a cash flow definition string using `entry_name` as the name of the cash flow. Note that while `Cashflow` can parse cash flow definition strings using any of the five defined parsing methods, `Cashflow` **always** generates cash flow definition strings for use with the `dict` parsing method, which is best for human readability.

Example:
`cfds = mycashflow.make_entry_text('Huge Project')`

Static Method `parse_entry_text`
Arguments: `(entry: str)`

Creates a `Cashflow` object from a cash flow definition string.

Example:
```
mycashflow =
Cashflow.parse_entry_text('Project D|arith|0.10|5,12,1000')
```

Method `write_entry`
Arguments: `(file: str, entry_name: str)`

Create a cash flow definition string using `entry_name` as the name of the cash flow and saves it to file `file`. It is recommended to give the full path of the save file in `file`. The cash flow definition string is always written to the file in append mode; if the file does not exist, Python will create it. Make sure you are not saving a cash flow definition string with a duplicate name to an already-existing file.

Static Method `write_all_entries`
Arguments: `(file: str, cf_named_dict: dict)`

Given a dictionary `cf_named_dict` with the names of cash flows as the keys, and the corresponding `Cashflow` objects as the values, render each entry as a cash flow definition string and write them all to file `file`. It is recommended to give the full path of the save file in `file`. The cash flow definition strings are written to the file in append mode; if the file does not exist, Python will create it. Make sure you are not saving a cash flow definition strings with a duplicate names to an already-existing file.

Example:

```
cashflow1 = Cashflow.from_list([-10000, 3500, 3500, 3500, 3500])

cashflow2 = Cashflow.expansion_from_dict({0:-25000,1:5000, 10:7500},
0.085)

cashflow3 = Cashflow.from_dict(0:-10000,1:3400,5:2800,10:7800, 0.08)

my_cashflow_dict['Project 1'] = cashflow1
my_cashflow_dict['Project 2'] = cashflow2
my_cashflow_dict['Project 3'] = cashflow3

savefile = r'C:\path\to\my\cashflows\bigcashflowfile.txt'

Cashflow.write_all_entries(file = savefile,
cf_named_dict = my_cashflow_dict)
```

Static Method `fetch_entry_names`
Arguments: `(file: str))`

From a file given in `file`, return the list of names of the cash flow definition strings contained in it.

Example:

File `C:\cashflows\cashflowfile.txt` contains the following cash flow definition strings:

```
Project A|list|0.08|-12000,3000,4000,5000,6000
Project B|dict|0.07|0:-10000,1:3400,5:2800,10:7800
Project C|expand|0.065|0:-10000,1:3400,5:2800,10:7800
Project D|arith|0.10|5,12,1000
Project E|geom||3,10,400,0.075
```

To get the list of cash flow names:

```
names_list = Cashflow.fetch_entry_names(
```

```
file = ‘C:\cashflows\cashflowfile.txt’)

print(names_list)
> [‘Project A’,‘Project B’,‘Project C’,‘Project D’,‘Project E’]
```

Static Method fetch_entries

Arguments: (file: str, entry_names: list = None, merge_into: dict = {})

From file file containing cash flow definition strings, read them into a dictionary with the cash flow names as keys, and Cashflow objects as values. If the analyst wishes to merge this dictionary with an existing one, pass the name of the existing dictionary to argument merge_into. If no merging is required, argument merge_into may be omitted.

The default for this method is that all records will be read in (entry_names = None). However, an entry other than None is supplied for entry_names, this method will read in only the records indicated in argument entry_names, which is a list of strings containing the entry names desired for extraction. If only a single entry is desired, this argument can be passed as a string rather than a list.

Examples:

File C:\cashflows\cashflowfile.txt contains the following cash flow definition strings:

```
Project A|list|0.08|-12000,3000,4000,5000,6000
Project B|dict|0.07|0:-10000,1:3400,5:2800,10:7800
Project C|expand|0.065|0:-10000,1:3400,5:2800,10:7800
Project D|arith|0.10|5,12,1000
Project E|geom||3,10,400,0.075
Project G|annuity|0.115|4,9,3500
```

To get only the cash flow for Project D:

```
filename = ‘C:\cashflows\cashflowfile.txt’
```

```
cashflow_dict = Cashflow.fetch_entries(file = filename,
entry_names = ‘Project D’)
```

This yields a `dict` with one entry with `‘Project D’` as the key, and the corresponding `Cashflow` object as the value.

Now, fetch the entries for Project A and Project E, merge them with the dictionary for Project D into a new dictionary, and assign it to a new dictionary.

```
cashflow_dict_2 = Cashflow.fetch_entries(
file = filename,
entry_names = [‘Project A’, ‘Project E’],
merge_into = cashflow_dict)
```

To read all the cash flows from the file into a new dictionary:

```
cashflow_dict_all = Cashflow.fetch_entries(file = filename)
```

4.2.7 Analysis Methods

WARNING!

None of the analysis methods take the discount rate as an argument. The discount rate must be set in the `Cashflow` object either at the time of creation (using one of the quasi-constructors), or by using property method `set_rate()`.

Method `irr`
Arguments: None

Calculates the internal rate of return for the cashflow.

Usage: `my_irr = mycashflow.irr()`

Method `mirr`
Arguments: `(reinvest_rate: float)`

Given the reinvestment rate `reinvest_rate`, calculate the modified internal rate of return.

Usage: `my_mirr = mycashflow.mirr(0.11)`

Method `npv`
Arguments: None

Calculates the net present value of the cashflow.

Usage: `my_npv = mycashflow.npv()`

Method `nfv`
Arguments: None

Calculates the net future value of the cashflow.

Usage: `my_nfv = mycashflow.nfv()`

Method `nus`
Arguments: None

Calculates the net uniform series value (EUAB, EUAC, EUAW) of the cashflow.

Usage: `my_nus = mycashflow.nus()`

Method `nper`
Arguments: None

Returns the number of compounding periods in the cashflow.

Usage: `my_nper = mycashflow.nper()`

Method `payback`
Arguments: None

Usage: `mycashflow.payback()`
Calculates the number of compounding periods required to recover an investment cost. This is calculated using nominal dollars. Note that, while costs after period 0 are accounted for, once the cash balance is 0 or greater, the calculation stops, and will not account for costs beyond that compounding period.

If payback is never achieved, the value `inf` is returned (infinity from package `math`: `math.inf`). This is to facilitate comparisons such as
`if cf1.payback() < cf2.payback():`

Method `payback_disc`
Arguments: None

Usage: `mycashflow.payback_disc()`

Calculates the number of compounding periods required to recover an investment cost. This is calculated using constant dollars based on the discount rate given in the `Cashflow` object. Note that, while costs after period 0 are accounted for, once the cash balance is 0 or greater, the calculation stops, and will not account for costs beyond that compounding period.

If payback is never achieved, the value `inf` is returned (infinity from package `math`: `math.inf`). This is to facilitate comparisons such as
`if cf1.payback_disc() < cf2.payback_disc():`

4.3 Other Methods

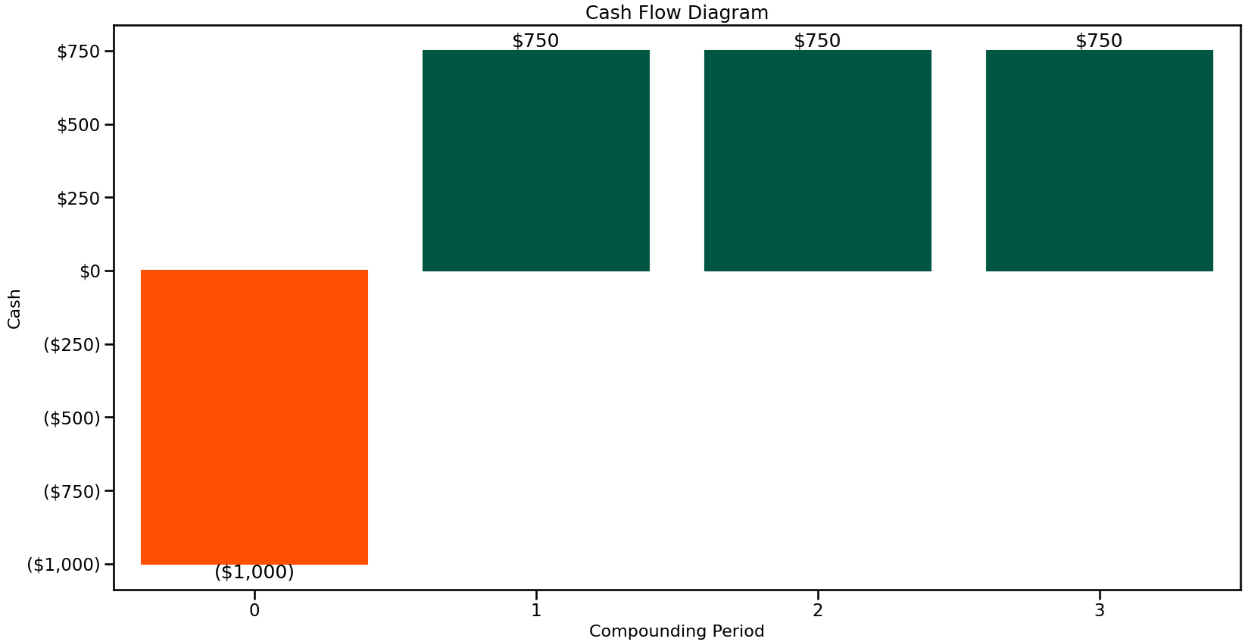
Method `replicate`
Arguments: `(times: int = 2)`

This method is used for incremental analysis to ensure that cashflows with different horizons are made to have identical horizons. This method is used in class `IncrementalIRR` for the pairwise comparison of incremental IRRs. The cashflow is replicated as many times as given in the `times` argument. A new `Cashflow` object is returned; the original one is unchanged.

Example:

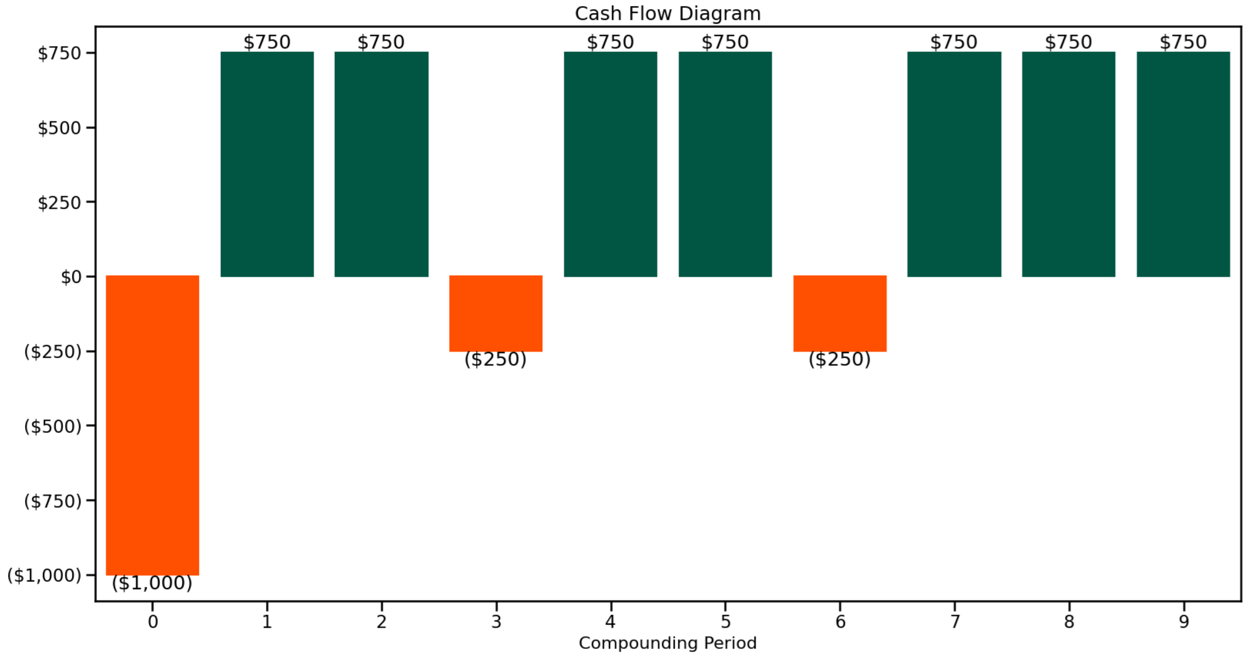
Original cashflow: `cf_original = Cashflow.from_list([-1000,750,750,750])`

Cashflow diagram:



Replicate the original cashflow three times:
`cf3 = cf_original.replicate(times = 3)`

Cashflow diagram:



Method `separate_bc`
Arguments: None

This method breaks a `Cashflow` object into two `Cashflow` objects. One of them contains only positive entries or zero (benefits), and the other contains for negative entries or zero (costs). This is used to facilitate analysis of benefit/cost ratios. However, this should be used only for the simplest benefit/cost models, such as an initial investment followed only by revenues. Benefit/cost analysis must not be done on netted cashflows. If cashflows are more complicated, where compounding periods have both benefits and costs, two `Cashflow` objects should be defined from the start. There will be more discussion of this in the section for class `IncrementalBCR`.

Usage: `[cf_benefit, cf_cost] = mycashflow.separate_bc()`

Method `to_dict`
Arguments: None

Returns a `dict` object containing the compounding periods as the keys, and the cash amounts as the values.

Usage: `mycashflow.to_dict()`

Method `to_list`
Arguments: None

Returns a `list` object containing the cash values for each compounding period, starting with period 0 and continuing until the end of the cash flow series.

Usage: `mycashflow.to_list()`

Method `plot_cfd`
Arguments: (
```
subject:  str = None,
scale:  int = 1,
savefile:  str = None,
showplot:  bool = False,
poscolor:  str = '005643',
negcolor:  str = 'FF4F00',
barfontsize int:  = 12
)
```

Returns a cash flow diagram, of which several examples are in this document.

Argument `subject` adds text chosen by the analyst to the plot title.

Argument `scale` is used when large cash amounts are used in the `Cashflow` object to make the cash flow diagram easier to read.

Argument `savefile` indicates to where `Enginomics` should save the plot. The file extension provided (`.png`, `.jpg`, `.svg`) determines the file format for export. The best results have come by exporting to `.png`. If `savefile` is set to `None`, no file will be saved; this is the default behavior.

Argument `showplot` indicates whether to show the cash flow diagram while the program is running. Setting `showplot = True` will cause the cash flow diagram to appear during execution. This will suspend execution of the program until the plot is closed. Setting `showplot = False` will suppress display of the cash flow diagram; this is the default behavior.

Argument `poscolor` is the color selection for showing positive cash flow in a compounding period. A hex color code should be used for this argument. The default is NDSU green (#005643).

Argument `negcolor` is the color selection for showing negative cash flow in a compounding period. A hex color code should be used for this argument. The default is engineering orange (#FF4F00).

Argument `barfontsize` sets the font size of the labels above (or below) the bars of the cash flow diagram.

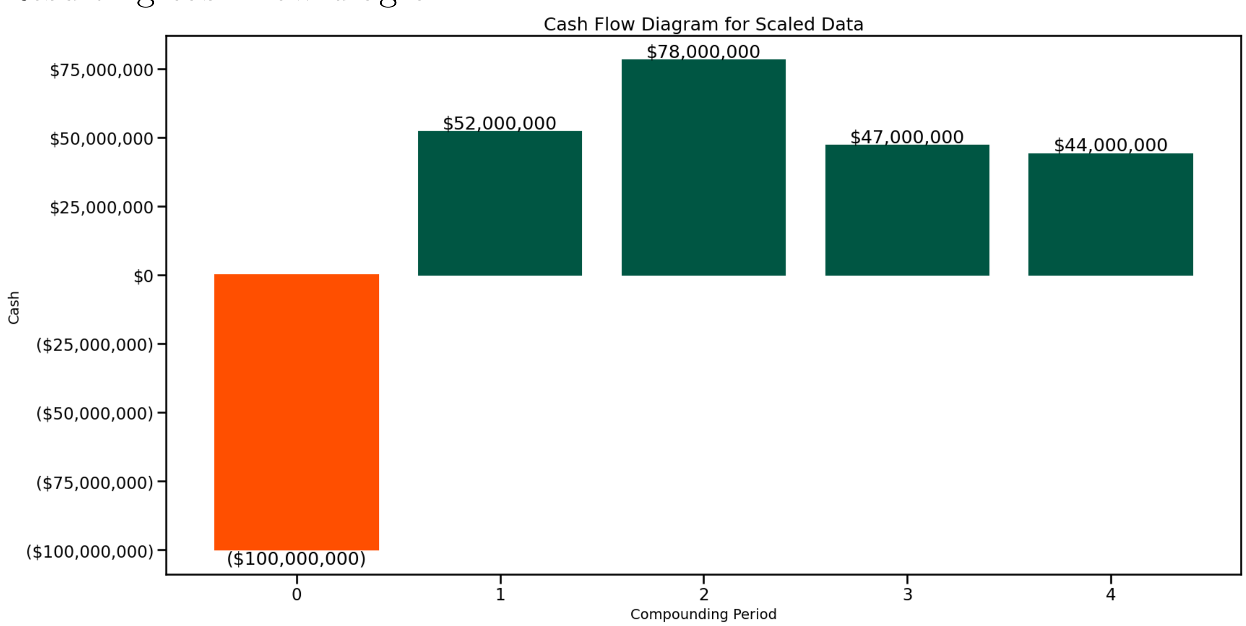
As an aside, while commas cannot be placed in numeric literals in Python, underscores can be use in their place to increase readability, as shown below.

Example:

```
mycashflow = Cashflow.from_list([-100_000_000,
52_000_000, 78_000_000, 47_000_000, 44_000_000])

mycashflow.plot_cfd(subject = 'Unscaled Data',
savefile=r'C:\save\to\unscaled.png', scale=1_000_000)
```

Resulting cash flow diagram:

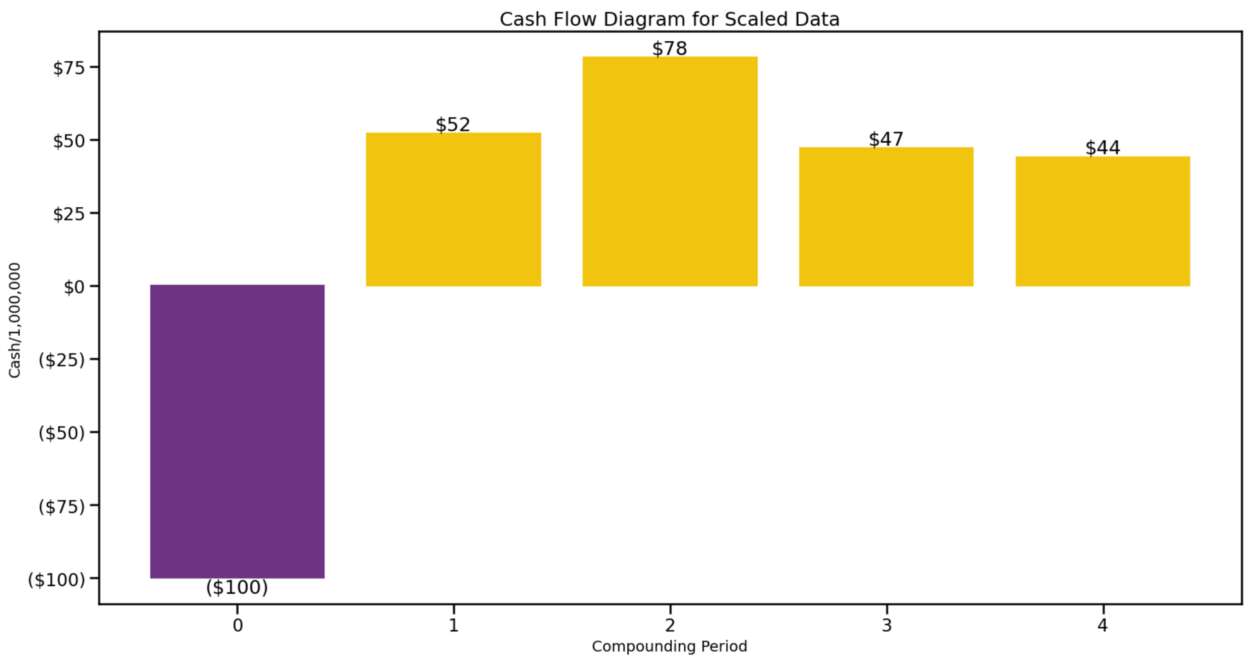


Now use scaling, and also change the color scheme:

```
savefile=r'C:\save\to\scaled.png'

mycashflow.plot_cfd(subject = 'Scaled Data',
scale=1_000_000, poscolor = '#f1c40f', negcolor = '#6c3483')
```

Resulting cash flow diagram:



5 Class IncrementalBCR

5.1 About

Class `IncrementalBCR` is for performing incremental benefit-cost ratio (BCR) analysis for an arbitrary number of project alternatives. A detailed report is available upon performing the analysis.

5.2 How to Use

5.2.1 Import Statement

```
from Enginomics import IncrementalBCR
```

5.2.2 Constructor

```
bcr_model = IncrementalBCR(outputfile: str = None, marr: float = None)
```

The minimum acceptable rate of return (MARR) (a.k.a. *hurdle rate*) must be passed as a decimal fraction; e.g. if the MARR is 7.5%, then the argument should be passed as `marr = 0.075`.

Argument `outputfile` should contain the full path of the output file, and have an extension of `.xlsx`

If the analyst chooses not to pass values for a parameter, then the parameter value will need to be set using the property methods `set_marr()` and/or `set_outputfile()`.

5.2.3 Static Methods

Static Method `prep_irr`
Arguments: None

Usage: `irr_model = bcr_model.prep_irr()`

`IncrementalBCR` does not perform incremental internal rate of return (IRR) analysis. However, method `prep_irr()` creates an `IncrementalIRR` object so that the analyst does not need to set up an incremental IRR analysis from scratch when analyzing the same project alternatives.

If performing both incremental BCR and incremental IRR analyses, the analyst should perform incremental BCR first. This is because the benefit and cost cash flows will be summed, and the net cash flow series send to the `IncrementalIRR` object. This causes a decrease in resolution and loss of information. Incremental BCR analysis should not be performed on netted cash flows.

5.2.4 Property Methods

Method `set_marr`
Arguments: `(marr: float)`

Example: For a MARR of 9%:
`bcr_model.set_marr(0.09)`

Method `get_marr`
Arguments: None

Example: `marr = bcr_model.get_marr()`

Method `set_outputfile`
Arguments: `(outputfile: str)`

Example: `bcr_model.set_outputfile('C:\cashflows\bcrmodel.xlsx')`

Method `get_outputfile`
Arguments: None

Example: `outputfile = bcr_model.get_outputfile()`

5.2.5 Methods

Method `add_alternative`
Arguments:
`(altname: str,`

`cf_benefit: 'Cashflow',`
`cf_cost: 'Cashflow')`

Example:
`bcr_model.add_alternative('Project A', project_a_cashflow_benefit, project_a_cashflow_cost)`

Every option for analysis must have a unique name, which should be assigned to parameter `altname`.

For an incremental cost-benefit ratio analysis, benefits and costs **must** have separate cash flow series defined. The `Cashflow` object with the project benefits should be passed to parameter `cf_benefit`, and the `Cashflow` object with the project costs should be passed to parameter `cf_cost`.

WARNING!

For any single alternative, the benefit cash flow and the cost cash flow **must** have the same horizon. If they do not, `Enginomics` will throw an exception, and the program will terminate.

NOTE!

It is not required that all alternatives have the same horizon. Only the benefit and cost cash flows within a single alternative must have the same horizon.

Method `del_alternative`

Arguments: `(altname: str, fail_silent: bool = False)`

Remove an alternative from the specified object by passing the name of the alternative to argument `altname`.

WARNING!

Spelling, whitespace, and capitalization must match **exactly**.

If argument `fail_silent = False`, `Enginomics` will throw an exception, and the program will terminate; this is the default behavior. If argument `fail_silent = True`, the program will continue executing even if the alternative name does not exist.

Example: `bcr_model.del_alternative('Alternative A')`

Method `ingest_file`
Arguments: `(file: str)`

Reads in the cash flow definition string entries from a text file. Since benefit-cost ratio analysis requires separate cash flows for benefit and cost, any cash flow definition string that does not have a name (i.e. first column value) ending with **B** or **C** will be silently discarded.

Example: `bcr_model.ingest_file('C:\cashflows\bcrentries.txt')`

Method `list_alternatives`
Arguments: None

Return the list of alternative names from the `IncrementalBCR` object.

Example: `bcr_altnames = bcr_model.list_alternatives()`

Method `generate`
Arguments: None

This method performs the pairwise incremental benefit-cost ratio analysis, and calculates a number of other metrics. If an output file was specified in the constructor or with property method `set_outputfile()`, a Microsoft Excel spreadsheet will be generated. The output spreadsheet contains two worksheets: **alternatives listing** and **pairwise comparisons**. The contents of these worksheets will be discussed in methods `fetch_alternatives` and `fetch_pairwise`.

WARNING!

Neither of the following methods will return meaningful results until method `generate()` is invoked.

Method `fetch_alternatives`
Arguments: None

Usage: `bcr_model_alts_df = bcr_model.fetch_alternatives()`

This method returns a `pandas DataFrame` containing the same information contained in worksheet **alternatives listing**. It is not required that a Microsoft Excel spreadsheet be generated to use this method. The field names in the `pandas DataFrame` are the same as the column headers in the worksheet.

An example of the Excel output: (please note that the graphic is divided into two sections for readability; in the worksheet, all information is displayed in a single row for each alternative)

alternative	MARR	PW(benefit)	PW(cost)	NPW	EUAB	EUAC
PLAN A	0.18	195.3474198	-145	50.34741982	43.46766105	-32.26462299
PLAN B	0.18	293.839578	-250	43.83957804	65.38360832	-55.62866033
PLAN C	0.18	383.5624065	-310	73.56240649	85.34825131	-68.97953881
PLAN D	0.18	683.7540948	-425	258.7540948	152.1452972	-94.56872256
PLAN E	0.18	898.817259	-750	148.817259	200	-166.885981

EUAW	CBR	IRR	payback period	discounted payback period
11.20303806	1.347223585	0.271584072	3.372093023	5.661180209
9.754947985	1.175358312	0.225587905	4.133333333	7.224847819
16.3687125	1.237298085	0.246123454	3.5	6.336428486
57.5765746	1.608833164	0.337589793	2.814569536	4.284843529
33.11401901	1.198423012	0.234131365	3.75	6.804005457

Columns (with descriptions if necessary):

1. alternative: alternative name as assigned by the analyst
2. MARR: minimum acceptable rate of return, assigned by the analyst
3. PW(benefit): present worth of benefits
4. PW(cost): present worth of costs
5. NPW: net present worth
6. EUAB: equivalent uniform annual series for benefits
7. EUAC: equivalent uniform annual series for costs
8. EUAW: equivalent uniform annual series for net worth

9. CBR: cost-benefit ratio
10. IRR: internal rate of return
11. payback period
12. discounted payback period

Method `fetch_pairwise`
Arguments: None

Usage: `bcr_model_df = bcr_model.fetch_pairwise()`

This method returns a `pandas DataFrame` containing the same information contained in worksheet **pairwise comparison**. It is not required that a Microsoft Excel spreadsheet be generated to use this method. The field names in the `pandas DataFrame` are the same as the column headers in the worksheet.

An example of the Excel output:

comparison	defender	challenger	winner	incremental BCR	MARR
1	NOTHING	PLAN A	PLAN A	1.347223585	0.18
2	PLAN A	PLAN B	PLAN A	0.938020555	0.18
3	PLAN A	PLAN C	PLAN C	1.140696889	0.18
4	PLAN C	PLAN D	PLAN D	2.610362507	0.18
5	PLAN D	PLAN E	PLAN D	0.661732813	0.18

Note that the first defender is always NOTHING, i.e. the do-nothing option.

6 Class IncrementalIRR

6.1 About

Class `IncrementalIRR` is for performing incremental internal rate of return (IRR) analysis for an arbitrary number of project alternatives. A detailed report is available upon performing the analysis.

6.2 How to Use

6.2.1 Import Statement

```
from Enginomics import IncrementalIRR
```

6.2.2 Constructor

```
bcr_model = IncrementalIRR(outputfile: str = None,
marr: float = None)
```

The minimum acceptable rate of return (MARR) (a.k.a. *hurdle rate*) must be passed as a decimal fraction; e.g. if the MARR is 7.5%, then the argument should be passed as `marr = 0.075`.

Argument `outputfile` should contain the full path of the output file, and have an extension of `.xlsx`

If the analyst chooses not to pass values for a parameter, then the parameter value will need to be set using the property methods `set_marr()` and/or `set_outputfile()`.

6.2.3 Static Methods

Static Method `prep_bcr`
Arguments: None
Usage: `bcr_model = irr_model.prep_bcr()`

`IncrementalIRR` does not perform incremental benefit-cost ratio (BCR) analysis. However, method `prep_irr()` creates an `IncrementalIRR` object so that the analyst does not need to set up an incremental IRR analysis from scratch when analyzing the same project alternatives.

If performing both incremental BCR and incremental IRR analyses, the analyst should perform incremental BCR first. This is because the benefit and cost cash flows will be summed, and the net cash flow series send to the `IncrementalIRR` object. This causes a decrease in resolution and loss of information. Incremental BCR analysis should not be performed on netted cash flows. However, if the cash flows are simple, with a single initial cost and only benefits thereafter, `prep_bcr()` can be used to achieve an accurate incremental BCR analysis.

6.2.4 Property Methods

Method `set_marr`
Arguments: `(marr: float)`

Example: For a MARR of 9%:
`irr_model.set_marr(0.09)`

Method `get_marr`
Arguments: None

Example: `marr = irr_model.get_marr()`

Method `set_outputfile`
Arguments: `(outputfile: str)`

Example: `irr_model.set_outputfile('C:\cashflows\irrmodel.xlsx')`

Method `get_outputfile`
Arguments: None

Example: `outputfile = irr_model.get_outputfile()`

6.2.5 Methods

Method `add_alternative`
Arguments:

`(altname: str,`
`cfd: 'Cashflow')`

Example: `irr_model.add_alternative('Project A', project_a_cashflow)`

Every option for analysis must have a unique name, which should be assigned to parameter `altname`. The corresponding `Cashflow` object should be assigned to parameter `cfd`

> **NOTE!**
> It is not required that all alternatives have the same horizon.

Method `del_alternative`
Arguments: `(altname: str, fail_silent: bool = False)`

Example: `irr_model.del_alternative('Alternative A')`

Remove an alternative from the specified object by passing the name of the alternative to argument `altname`.

> **WARNING!**
> Spelling, whitespace, and capitalization must match **exactly**.

If argument `fail_silent = False`, `Enginomics` will throw an exception, and the program will terminate; this is the default behavior. If argument `fail_silent = True`, the program will continue executing even if the alternative name does not exist.

Method `ingest_file`
Arguments: `(file: str)`

Reads in the cash flow definition string entries from a text file.

If the file contains separate cash flows for benefit and cost, i.e. any cash flow definition string that has a name (i.e. first column value) ending with **B** or **C**, the two parts will be added together. If both parts are not present in the file, the unpaired cash flow will be silently discarded. Regular cash flows are processed normally.

Example: `irr_model.ingest_file('C:\cashflows\irrentries.txt')`

Method `list_alternatives`
Arguments: None

Example: `irr_altnames = irr_model.list_alternatives()`

Return the list of alternative names from the `IncrementalIRR` object.

Method `generate`
Arguments: None

Example: `irr_model.generate()`

This method performs the pairwise incremental benefit-cost ratio analysis, and calculates a number of other metrics. If an output file was specified in the constructor or with property method `set_outputfile()`, a Microsoft Excel spreadsheet will be generated. The output spreadsheet contains two worksheets: **alternatives listing** and **pairwise comparisons**. The contents of these worksheets will be discussed in methods `fetch_alternatives` and `fetch_pairwise`.

WARNING!

Neither of the following methods will return meaningful results until method `generate()` is invoked.

Method `fetch_alternatives`
Arguments: None

Example: `irr_model_alts_df = irr_model.fetch_alternatives()`

This method returns a `pandas DataFrame` containing the same information contained in worksheet **alternatives listing**. It is not required that a Microsoft Excel spreadsheet be generated to use this method. The field names in the `pandas DataFrame` are the same as the column headers in the worksheet.

An example of the Excel output: (please note that the graphic is divided into two sections for readability; in the worksheet, all information is displayed in a single row for each alternative)

alternative	periods	IRR	NPV	EUAW	CBR	payback period	discounted payback period
PLAN A	10	0.271584072	50.34741982	11.20303806	1.347223585	3.372093023	5.661180209
PLAN B	10	0.225587905	43.83957804	9.754947985	1.175358312	4.133333333	7.224847819
PLAN C	10	0.246123454	73.56240649	16.3687125	1.237298085	3.5	6.336428486
PLAN D	10	0.337589793	258.7540948	57.5765746	1.608833164	2.814569536	4.284843529
PLAN E	10	0.234131365	148.817259	33.11401901	1.198423012	3.75	6.804005457

cf_0	cf_1	cf_2	cf_3	cf_4	cf_5	cf_6	cf_7	cf_8	cf_9	cf_10
-145	43	43	43	43	43	43	43	43	43	54
-250	60	60	60	60	75	75	75	75	75	60
-310	90	90	90	80	80	80	80	80	80	92
-425	151	151	151	151	151	151	160	160	160	140
-750	200	200	200	200	200	200	200	200	200	200

Columns (with descriptions if necessary):

1. alternative: alternative name as assigned by the analyst
2. periods: number of compounding periods in the cash flow
3. IRR: internal rate of return
4. NPW: net present worth
5. CBR: cost-benefit ratio
6. payback period
7. discounted payback period
8. cf_n: cash flows for individual compounding periods

WARNING!

While the cost-benefit ratio is given in the alternatives listing, this will be inaccurate if the cash flow is the sum of a distributed cost cash flow and a distributed benefit cash flow. Benefit-cost analysis should not be run on netted cash flows!

Method `fetch_pairwise`

Arguments: None

Usage: `irr_model_df = irr_model.fetch_pairwise()`

This method returns a `pandas DataFrame` containing the same information contained in worksheet **pairwise comparison**. It is not required that a Microsoft Excel spreadsheet be generated to use this method. The field names in the `pandas DataFrame` are the same as the column headers in the worksheet.

An example of the Excel output:

comparison	defender	challenger	winner	Delta IRR	MARR
1	NOTHING	PLAN A	PLAN A	0.271584072	0.18
2	PLAN A	PLAN B	PLAN A	0.164124486	0.18
3	PLAN A	PLAN C	PLAN C	0.221429753	0.18
4	PLAN C	PLAN D	PLAN D	0.54882861	0.18
5	PLAN D	PLAN E	PLAN D	0.076029851	0.18

Note that the first defender is always NOTHING, i.e. the do-nothing option.

7 Class Amortization

7.1 About

Class `Amortization` generates amortization tables for the repayment of loans.

> **WARNING!**
>
> This class is very different from the others discussed in this document. Class `Cashflow` is not used in class `Amortization`. Do not assume that parameters and behavior will be similar to class `Cashflow`.

7.2 How to Use

7.2.1 Import Statement

```
from Enginomics import Amortization
```

7.2.2 Constructor

```
Amortization(
principal: float,
intrate: float,
periods: int,
description: str = None,
period_type: str = 'month',
extra_pmt: float = 0
)
```

Description of the arguments:

- `principal`: the principal (borrowed amount) of the loan
- `intrate`: annual nominal interest rate of the loan
  NOTE: (example) an interest rate of 8% is passed as `intrate = 8`, **NOT** `intrate = 0.08`,
- `periods`: number of periods of the loan; for example, a 30-year mortgage would be expressed as `periods = 360` or `periods = 30 * 12`

- `description`: description of the amortization model (e.g. `description` = 'New Home Purchase')
- `period_type`: can be any of `'month'` (default), `'quarter'`, `'year'`, `'semiannual'`, or `'period'`. Class `Amortization` will calculate the correct periodic interest rate for you. The exception is if 'period' is chosen. This can be used to select unusual compounding periods, but in that case, the analyst is required to peform the calculation to determine the appropriate periodic interest rate.
- `extra_pmt`: This allows the analyst to model an amortization for which an extra amount is paid every period. By (U.S.) law, extra amounts paid must be applied to the principal, and this will reduce the amount of interest paid, and shorten the lifetime of the loan.

7.2.3 Methods

Method `generate`
Arguments: None

Example: for an `Amortization` object `mortgage`: `mortgage.generate()`

WARNING!

None of the following methods will return meaningful results until method `generate()` is invoked.

Method `get_minimum_payment`
Arguments: None

Example: `minimum_payment = mortgage.get_minimum_payment()`

Get the required minimum payment for each payment period.

Method `get_number_payments`
Arguments: None

Example: `number_of_payments = mortgage.get_number_payments()`

Get the total number of payments made to retire the loan. If argument `extra_pmt` is greater than zero, the value retrieved by this method will be less than the value given in argument `periods`.

Method `get_total_interest`
Arguments: None

Example: `total_interest_paid = mortgage.get_total_interest()`

Get the total amount of interest paid.

Method `get_total_payments`
Arguments: None

Example: `total_paid = mortgage.get_total_payments()`

Get the total amount paid.

Method `fetch_amortization`
Arguments: None

Example: `mortgage_df = mortgage.fetch_amortization()`
Returns the amortization table as a `pandas DataFrame`. See method `export_to_excel` for a discussion of the structure of the `DataFrame`, which has the same column names as worksheet **Amortization**

Method `export_to_excel`
Arguments: **filename: str**

Example: `mortgage.export_to_excel('C:\loans\loan_amortization.xlsx')`

This method exports two worksheets to the Excel workbook indicated by the analyst: **Summary** and **Amortization**.

The **Summary** tab has information about the loan.

metric	value
Description	New Home Purchase
Principal	$350,000.00
Interest Rate	7.5%
Loan Term	360
Compounding Period	month
Extra Payment per Term	$50.00
Required Minimum Payment	$2,447.25
Number of Payments Made	335
Total Payments Made	$836,556.44
Total Interest Paid	$486,556.44

The **Amortization** tab contains the amortization schedule.

period	principal	payment	interest paid	principal paid	remaining_principal
1	350000	2497.25	2187.5	309.75	349690.25
2	349690.25	2497.25	2185.56	311.69	349378.56
3	349378.56	2497.25	2183.62	313.63	349064.93
4	349064.93	2497.25	2181.66	315.59	348749.33
5	348749.33	2497.25	2179.68	317.57	348431.77
6	348431.77	2497.25	2177.7	319.55	348112.21
7	348112.21	2497.25	2175.7	321.55	347790.66

If an extra periodic payment is indicated, you will see something like this in the amortization table.

333	7376.92	2497.25	46.11	2451.15	4925.77
334	4925.77	2497.25	30.79	2466.46	2459.31
335	2459.31	2474.68	15.37	2459.31	0
336	0	0	0	0	0
337	0	0	0	0	0
338	0	0	0	0	0

In this example, a $350,000 30-year mortgage paid monthly with an extra $50 paid each month results in the mortgage being paid off in month 335 instead of month 360, i.e. 25 months early. Class `Amortization` will list all payment periods, even if the loan is paid off early.

8 Appendix A: Full Text of License

Apache License
Version 2.0, January 2004
http://www.apache.org/licenses/

TERMS AND CONDITIONS FOR USE, REPRODUCTION, AND DISTRIBUTION

1. **Definitions**

   "License" shall mean the terms and conditions for use, reproduction, and distribution as defined by Sections 1 through 9 of this document.

   "Licensor" shall mean the copyright owner or entity authorized by the copyright owner that is granting the License.

   "Legal Entity" shall mean the union of the acting entity and all other entities that control, are controlled by, or are under common control with that entity. For the purposes of this definition, "control" means (i) the power, direct or indirect, to cause the direction or management of such entity, whether by contract or otherwise, or (ii) ownership of fifty percent (50%) or more of the outstanding shares, or (iii) beneficial ownership of such entity.

   "You" (or "Your") shall mean an individual or Legal Entity exercising permissions granted by this License.

   "Source" form shall mean the preferred form for making modifications, including but not limited to software source code, documentation source, and configuration files.

   "Object" form shall mean any form resulting from mechanical transformation or translation of a Source form, including but not limited to compiled object code, generated documentation, and conversions to other media types.

"Work" shall mean the work of authorship, whether in Source or Object form, made available under the License, as indicated by a copyright notice that is included in or attached to the work (an example is provided in the Appendix below).

"Derivative Works" shall mean any work, whether in Source or Object form, that is based on (or derived from) the Work and for which the editorial revisions, annotations, elaborations, or other modifications represent, as a whole, an original work of authorship. For the purposes of this License, Derivative Works shall not include works that remain separable from, or merely link (or bind by name) to the interfaces of, the Work and Derivative Works thereof.

"Contribution" shall mean any work of authorship, including the original version of the Work and any modifications or additions to that Work or Derivative Works thereof, that is intentionally submitted to Licensor for inclusion in the Work by the copyright owner or by an individual or Legal Entity authorized to submit on behalf of the copyright owner. For the purposes of this definition, "submitted" means any form of electronic, verbal, or written communication sent to the Licensor or its representatives, including but not limited to communication on electronic mailing lists, source code control systems, and issue tracking systems that are managed by, or on behalf of, the Licensor for the purpose of discussing and improving the Work, but excluding communication that is conspicuously marked or otherwise designated in writing by the copyright owner as "Not a Contribution."

"Contributor" shall mean Licensor and any individual or Legal Entity on behalf of whom a Contribution has been received by Licensor and subsequently incorporated within the Work.

2. **Grant of Copyright License**. Subject to the terms and conditions of this License, each Contributor hereby grants to You a perpetual, worldwide, non-exclusive, no-charge, royalty-free, irrevocable copyright license to reproduce, prepare Derivative Works of, publicly display, publicly perform, sublicense, and distribute the Work and such Derivative Works in

Source or Object form.

3. **Grant of Patent License**. Subject to the terms and conditions of this License, each Contributor hereby grants to You a perpetual, worldwide, non-exclusive, no-charge, royalty-free, irrevocable (except as stated in this section) patent license to make, have made, use, offer to sell, sell, import, and otherwise transfer the Work, where such license applies only to those patent claims licensable by such Contributor that are necessarily infringed by their Contribution(s) alone or by combination of their Contribution(s) with the Work to which such Contribution(s) was submitted. If You institute patent litigation against any entity (including a cross-claim or counterclaim in a lawsuit) alleging that the Work or a Contribution incorporated within the Work constitutes direct or contributory patent infringement, then any patent licenses granted to You under this License for that Work shall terminate as of the date such litigation is filed.

4. **Redistribution**. You may reproduce and distribute copies of the Work or Derivative Works thereof in any medium, with or without modifications, and in Source or Object form, provided that You meet the following conditions:

   a You must give any other recipients of the Work or Derivative Works a copy of this License; and

   b You must cause any modified files to carry prominent notices stating that You changed the files; and

   c You must retain, in the Source form of any Derivative Works that You distribute, all copyright, patent, trademark, and attribution notices from the Source form of the Work, excluding those notices that do not pertain to any part of the Derivative Works; and

   d If the Work includes a "NOTICE" text file as part of its distribution, then any Derivative Works that You distribute must include a readable copy of the attribution notices contained within such NOTICE file, excluding those notices that do not pertain to any part of the Derivative Works, in at least one of the following places: within a NOTICE text file distributed as part of the Derivative Works; within the Source form or documentation, if provided along with the Derivative Works; or, within a display generated by the Derivative Works, if and wherever such third-party notices normally appear. The contents of the NOTICE file are for informational purposes only and do not modify the License.

You may add Your own attribution notices within Derivative Works that You distribute, alongside or as an addendum to the NOTICE text from the Work, provided that such additional attribution notices cannot be construed as modifying the License.

5. **Submission of Contributions**. Unless You explicitly state otherwise, any Contribution intentionally submitted for inclusion in the Work by You to the Licensor shall be under the terms and conditions of this License, without any additional terms or conditions. Notwithstanding the above, nothing herein shall supersede or modify the terms of any separate license agreement you may have executed with Licensor regarding such Contributions.

6. **Trademarks**. This License does not grant permission to use the trade names, trademarks, service marks, or product names of the Licensor, except as required for reasonable and customary use in describing the origin of the Work and reproducing the content of the NOTICE file.

7. **Disclaimer of Warranty**. Unless required by applicable law or agreed to in writing, Licensor provides the Work (and each Contributor provides its Contributions) on an "AS IS" BASIS, WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied, including, without limitation, any warranties or conditions of TITLE, NON-INFRINGEMENT, MERCHANTABILITY, or FITNESS FOR A PARTICULAR PURPOSE. You are solely responsible for determining the appropriateness of using or redistributing the Work and assume any risks associated with Your exercise of permissions under this License.

8. **Limitation of Liability**. In no event and under no legal theory, whether in tort (including negligence), contract, or otherwise, unless required by applicable law (such as deliberate and grossly negligent acts) or agreed to in writing, shall any Contributor be liable to You for damages, including any direct, indirect, special, incidental, or consequential damages of any character arising as a result of this License or out of the use or inability to use the Work (including but not limited to damages for loss of goodwill, work stoppage, computer failure or malfunction, or any and all other

commercial damages or losses), even if such Contributor has been advised of the possibility of such damages.

9. **Accepting Warranty or Additional Liability**. While redistributing the Work or Derivative Works thereof, You may choose to offer, and charge a fee for, acceptance of support, warranty, indemnity, or other liability obligations and/or rights consistent with this License. However, in accepting such obligations, You may act only on Your own behalf and on Your sole responsibility, not on behalf of any other Contributor, and only if You agree to indemnify, defend, and hold each Contributor harmless for any liability incurred by, or claims asserted against, such Contributor by reason of your accepting any such warranty or additional liability.

END OF TERMS AND CONDITIONS

9 Appendix B: History and Release Notes

- Version 0.8.4, June 1, 2025
    1. Added method `annuity` to allowable cash flow definition strings.
- Version 0.8.3, May 26, 2025
    1. Initial Public Release

10 Appendix C: Development Roadmap

10.1 Short Term

1. Adding comments to source code – essentially, putting this document in the source code. I know, I know...but I really wanted to get this package out in the wild.

10.2 Medium Term

1. Convert most data frame functionality from `pandas` to `duckdb`; `pandas` `DataFrame`s will still be used when such is required by an external library.

10.3 Long Term

There are no long-term plans at this time.