FN6815_Lect_02_Python

February 27, 2024

FN6815 Numerical Methods for Financial Instrument Pricing

Lecture 2: Python programming

- Dr. Yang Ye
- Email: yy@runchee.com

- 2023/2024 Mini Term 5

1. Setting up

1.1 Installation

To use Python, we need to install Python distribution and (optionally) an IDE.

Python distribution: Miniconda Note: Miniconda is a slim version than Anaconda. I prefer Mini-version because it offers some learning to use the packaging system.

Installation steps:

- Download Miniconda for your OS (Operation System): https://docs.conda.io/en/latest/miniconda.html
- (Windows) run the EXE installer
- (Mac) Open terminal and run command: `bash Miniconda3-latest-MacOSX-x86_64.sh`

(Optional) IDE like Visual Studio Code / PyCharm

- We can use Jupyter notebook as our IDE
- Or, we use VS Code or PyCharm. VS Code is preferred with good support for Jupyter notebook and many add-ons: https://code.visualstudio.com/

You could ask for my help for installation problem during class intervals.

1.2 Setting up Miniconda

- (Windows) Start Menu -> Anaconda Prompt
- (Mac) Open Terminal

1.3 Python Package Installation

There are two package management system: `pip` and `conda`.

- `pip` is from wider Python package repository (pypi.org), anyone can submit a package there.
- `conda` is a curated version by Python developer community and Anaconda, Inc.

Under the conda environment, `conda` is preferred to do installation.

1. Use `conda search package_to_install` first.
2. If not found, `pip install package`.

1.3 Setup environment

Conda provides multiple environments - one environment can contains a set of packages at their versions. We can have as many environment as possible.

Python's packages are built by its community and provide extension to the base Python. We mostly rely on the packages for our work.

Different environment provides separation of packages and their versions. This is useful for two reasons:

1. For one project, we may use a set of packages at their versions. We are fine with them for their usage and result. We would like to keep the packages unchanged as long as the project is running.

For another project, we may use another set of packages at their versions. We don't want to mess up with the packages and their versions.

2. Package may change over versions and packages depends on each other. Dependency may have conflict: package_A requires package_B > 1.0 while package_C requires package_B < 1.0. For a small set of packages, we can manually resolve the conflict. But for a large set of packages, it is hard to do so. This would happen if we use one environment for all projects.

Below is an example we create an environment called `dyson`, choose your own name.

I have added python=3.11.8 to specify the version of Python. Although the latest version is 3.12, we may not want to use the latest version because some packages may not support the latest version. Use the 2nd latest major version is a good choice.

```
conda create --name dyson python=3.11.8
activate dyson
# once we have activated the environment, we can install packages
pip ...
# after using the environment, we can deactivate it
deactivate
```

1.4 Install the packages

After you have created the environment, you can install the packages by the following command: Type below commands one-by-one in the command prompt/Terminal

```
conda update --all
conda clean --all
cd .../fn6815
# I have provided a requirements.txt file for you to install the packages nee
pip install -r requirements.txt
```

1.5 Setup the Development Environment

You could choose one of the two ways:

1. Jupyter Lab, which is an upgraded version of earlier Jupyter Notebook server.
2. Visual Studio Code or PyCharm also supports to open Jupyter notebook.

1.6 Start Jupyter Lab

- Under Anaconda prompt, activate of your environment if any.

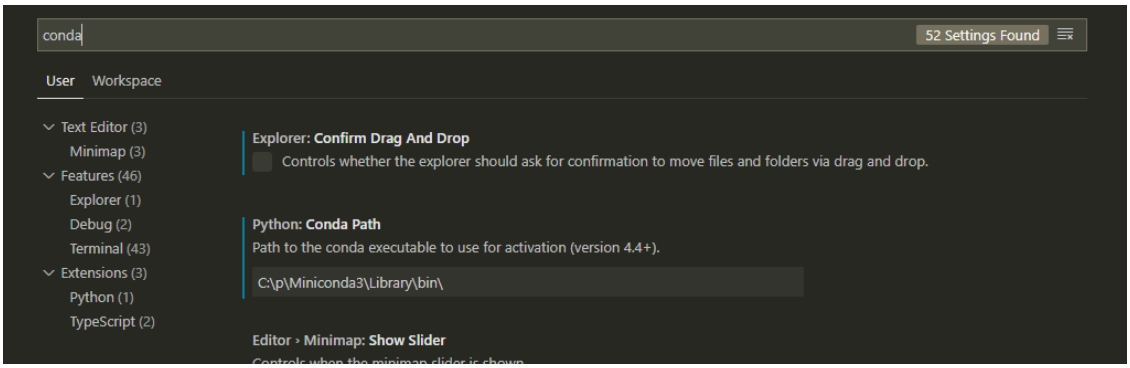
You shall see browser to open the jupyter page.

- Create a new console or notebook.
- Shortcut keys:
  - `Enter` to edit a cell
  - `Esc` to quit edit
  - `Up/Down` to move around
  - `Ctrl+Enter` to run a cell
  - `a` is to create a cell above
  - `b` is to create a cell below
  - `m` is to turn a cell to markdown text
  - `c` is to turn a cell to code
  - `dd` or `x/z` is to delete/un-delete a cell

1.7 Start Visual Studio

1. Install Extensions: **Python, Pylance, Jupyter**

2. Go to settings (left-bottom 'Gear'-icon), type conda. Fill in conda's location (get it from command: where conda)



3. Setting up Python

Open the Command Palette (Ctrl+Shift+P) and select the **Python: Select Interpreter command**.

4. Open a Jupyter notebook

5. (Optional) Setting Up Linters in VS Code

Linter scans code to prompt for improvements. They are good helpers. You could install Add-on like "pylint" or "ruff".

Setup in JSON

```
"notebook.formatOnSave.enabled": true,
"notebook.codeActionsOnSave": {
    "source.fixAll": "explicit",
    "source.organizeImports": "explicit"
}
```

2. Python

Let's do a quick introduction / review!

2.1 Basic types

[1]:
```
a = 10
type(a)  # int
```

[1]:
```
int
```

[2]:
```
a = 10.0
type(a)  # float
```

[2]:
```
float
```

[3]:
```
print(type("astr"))
print(type("astr") is str)  # not so preferred
print(isinstance("astr", str))  # preferred
```

```
<class 'str'>
True
True
```

[4]:
```
True, False, True and False, True or False, not True
```

[4]:
```
(True, False, False, True, False)
```

`float`, `int`, `str` and `bool` are basic types in Python.

Because there is no declaration for the variable in Python, you can check whether a variable is of a certain type with `isinstance()` function.

```
isinstance(a, float)
isinstance(d, dict)
isinstance(date, datetime.datetime)
```

We have a special value `None` which is used to represent the absence of a value.

```
a = None
type(a) # NoneType

a is None # True
```

2.1.1 Type Hint Type hint (also known as Type annotation) has been introduced from v3.5.

Annotation means it is not enforced during run time but there are tools use them to check your code and notify you for the discrepancy. This is known as "static type checking".

```
a: str = "int"
b: int = 3

c: float = "a" # wrong annotation but still runnable.

def add(a: int, b: int) -> int:
    return a + b
```

2.2 Operators

- numeric operators: `+, - , * , /, //, %, **`
- logical operators: `and, or`

Common functions:

- `len()`
- `print()`
- `repr()`, `eval()`

[5]:
```
a = "1"
eval("1 + " + a)
```

[5]:
```
2
```

[6]:
```
"abc".count("a")
```

[6]:
```
1
```

2.3 String operations

- Both single and double quotes are used for string. `"abc" == 'abc'`
- String is an object in Python. There are many methods for it.

```
"a".find("a") # 0
"a".find("z") # -1 not found
print("\n".join(["a","b"]) # a and b in two lines
"abc".count('a')
```

- slicing reference: `[::]`,
  - `[:3]` means first three characters
  - `[3:]` means after first three characters
  - `[::2]` means every two characters
  - `[::-1]` means reversing the order
  - `[1:-1]` means remove first and last character

[7]:
```
"aaa".count("a")
"abc"[::-1]
"aabbcc"[1:-1]
```

[7]:
```
'abbc'
```

2.4 String formatting: Anything to String conversion (succeeded by f-string)

[8]:
```
print("a{}".format(1))

print("a{a}".format(a=1))

print("a{a} and a{b}".format(a=1, b=2))
```

```
a1
a1
a1 and a2
```

2.5 f-string Construction, since Python 3.6

[9]:
```
# f-string: since Python 3.6
from datetime import date

f"{1+1:.2f}_{3}_{date(2020, 1, 1):%Y-%b-%d}"
```

[9]:
```
'2.00_3_2020-Jan-01'
```

[10]:
```
a_str = "MFE"

print(f"{a_str} FE8828")  # equivalent to : print(a_str + ' FE8828')

print(f"{a_str=} FE8828")  # add = will print the name of the variable

# print with formatting and calculation
print(f"{1+1:03d}")

print(f"{3.1415926:.3f}")
```

```
MFE FE8828
a_str='MFE' FE8828
002
3.142
```

2.6 Regular expression (did you remember this from R?)

[11]:
```
import re

if re.match("^A", "A"):
    print("Found A")
if not re.match("^A", "B"):
    print("Not Found A")

print(re.sub("[0-9]+", "{number}", "We have 3 apples."))
```

```
Found A
Not Found A
We have {number} apples.
```

2.7 Composite Data types

- tuple: `(,)`
- list: `[]`
- dict: `{'a': 'a'}`
- set: `{'a', 'b'}`

2.8 Tuples

A tuple is a data structure, yet it's still quite simple and widely used to combine a few types of data together. It is defined by providing objects in parentheses:

For tuple, you can even drop the parentheses and provide multiple objects, just separated by commas:

[12]:
```
t = (1, 2.5, "data")
print(t)
```

```
t = 1, 2.5, "data"
print(t)
```

```
(1, 2.5, 'data')
(1, 2.5, 'data')
```

[13]:
```
isinstance(t, tuple)
```

[13]:
```
True
```

[14]:
```
# tuple indexing: zero-based
t[2]
```

[14]:
```
'data'
```

Items in tuple is accessible by index.

[15]:
```
print(t.count("data"))
print(t.index(1))
```

```
1
0
```

Tuple is read-only. Once created, you cannot append or update existing values

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Input In [21], in <module>
----> 1 t[3] = 'data'

TypeError: 'tuple' object does not support item assignment
```

Usage of Tuple

[16]:
```
# Use * to melt the tuple to merge into the new tuple
t2 = (*t, "data")
t2
```

[16]:
```
(1, 2.5, 'data', 'data')
```

[17]:
```
# Use tuple to assign values
a, b = 1, 2
print(a, b)
a, b = b, a  # sw
print(a, b)
```

```
1 2
2 1
```

[18]:
```
# return from a function
def func1():
```

```
    return 1, 2, 3


func1()
```

[18]: (1, 2, 3)

2.9 List

List is like C++'s std::vector class. It's expandable and mutable.

- Initialize it with `[]` and use `+` to concatenate them.
- It has methods like `.append()` single-element and `.expand()` for adding another list, `.index()` to find an element.

[19]:
```
a_list = []
a_list.append(1)
a_list.append(2)
print(a_list)
print(len(a_list))
```

```
[1, 2]
2
```

[20]:
```
# Create a list with a fixed length
a_list = [0] * 10
print(a_list)

# melt the list to merge into the new list
print([1, *a_list])
```

```
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
[1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
```

2.10 Dict

`dict` objects are dictionaries, that allow data retrieval by keys (which can, for example, be str objects). They are so-called key-value stores or hashmaps.

The items in a dict will preserve the order of insertion, since Python 3.7.

We can utilize `dict` to store and return complex results from a function. Ensure that the keys are unique and free from typo errors.

"'{python} def a_func(): # second dividend will overwrite the first one return {'dividend': 0.04, 'rate': 0.04, 'time': 1.3, 'dividend': "a"}

result = a_func() result['rase'] # KeyError "'

Simple example

[21]:
```
dict_books = {}
dict_books["Time Traveler"] = 3
dict_books["Time Traveler"] += 1

dict_books["Space and Time"] = 7

print(dict_books.keys())
print(dict_books.values())
print(dict_books.items())

for k in dict_books:
    print(f"{k, dict_books[k]}")

for k, v in dict_books.items():
    print(f"{k}:{v}")

# convert a dictionary to a list and then get the first element
list(dict_books.items())[0]
```

```
dict_keys(['Time Traveler', 'Space and Time'])
dict_values([4, 7])
dict_items([('Time Traveler', 4), ('Space and Time', 7)])
('Time Traveler', 4)
('Space and Time', 7)
Time Traveler:4
Space and Time:7
```

[21]:
```
('Time Traveler', 4)
```

2.11 Looping over a Dict

There are three ways to loop over a dictionary.

[22]:
```
users = {"a": 3, "b": 5}

# To update values in a dictionary
for k in users:
    users[k] += 4

# To read the value from a dictionary
for v in users.values():
    print(v)

# To read both the key and value from a dictionary (can update via dd[k])
for k, v in users.items():
    print(k, v)
```

```
7
9
```

```
a 7
b 9
```

[23]:
```python
# Below is a bad example.
# But do not del key from dict in a for loop for a dictionary as below.
# The error occurs because users.keys() changes when dict changes.
# We used try...except to avoid the crash of the program.

try:
    for k in users.keys():
        del users[k]
except Exception as e:
    print(f"Got error: {e}")
    pass

print(users)
print(users)
```

```
Got error: dictionary changed size during iteration
{'b': 9}
{'b': 9}
```

[24]:
```python
# This is a good example. list(users.keys()) stores it to a list.

kk = list(users.keys())
for k in kk:
    if k.startswith("a"):
        del users[k]
print(users)
```

```
{'b': 9}
```

2.12 Set

Set object has a similar form to dictionary. Because keys in both set and dict are unique.

[25]:
```python
set1 = set([1, 2, 3, 4, 3, 9])
set2 = {1, 2, 3, 4, 5}
print("set1", set1)
print("set2", set2)
print("set1.union(set2)", set1.union(set2))
print("set1 | set2", set1 | set2)
print("set1 & set2", set1 & set2)
print("set1 - set2", set1 - set2)
# Other methods of set
# Calling these methods via .
# s.intersection()
# s.difference()
# s.symmetric_difference(t) # Items in either s or t but not both.
```

```
set1 {1, 2, 3, 4, 9}
set2 {1, 2, 3, 4, 5}
set1.union(set2) {1, 2, 3, 4, 5, 9}
set1 | set2 {1, 2, 3, 4, 5, 9}
set1 & set2 {1, 2, 3, 4}
set1 - set2 {9}
```

[26]:
```
unique_visitors = set(["Uncle A", "Uncle B", "Aunty C"]) | set(
    ["Grandson D", "Grandson E", "Aunty C"]
)
print(unique_visitors)

common_visitors = set(["Uncle A", "Uncle B", "Aunty C"]) & set(
    ["Grandson D", "Grandson E", "Aunty C"]
)
print(common_visitors)

set(["Uncle A", "Uncle B", "Aunty C"]) - set(["Grandson D", "Grandson E",␣
 ↪"Aunty C"])
```

```
{'Uncle B', 'Aunty C', 'Grandson E', 'Grandson D', 'Uncle A'}
{'Aunty C'}
```

[26]:
```
{'Uncle A', 'Uncle B'}
```

3. Datetime

`datetime` is a module in Python. Inside it, there is a `datetime` object (of the same name)

I usually use `datetime.datetime`, to represent both date and datetime. Usually this is effective to save some typing but also avoids conversion between them.

On module Usage

- With `import datetime`, the module name is prefixed for the object, `datetime.datetime`
- Write `from datetime import datetime` before you want to use datetime object. (preferred)
- Other example of `import`:
- math module, `import math`, to use `math.exp`, `math.isclose`, etc.
- Other libraries, `import numpy as np`. `as` adds an alias so you can use `np.exp()`

[27]:
```
from datetime import datetime

isinstance(datetime.now(), datetime)

datetime.today(), datetime.now()
```

[27]:
```
(datetime.datetime(2024, 2, 27, 21, 11, 20, 840445),
 datetime.datetime(2024, 2, 27, 21, 11, 20, 840445))
```

3.1 Examples for Datetime

- `from datetime import datetime`: Usually use datetime for date (with time = 0:0:0) and datetime (with time)
- `datetime.now()` return datetime
- `datetime.today()` same as now()
- `datetime.now().date()`: returns a date from now()

Create a date/datetime:

- `datetime.strptime()`: convert from str to time, p for parse
- `dt.strftime()`: convert from time to str
- You will get a "timedelta" object when subtraction two dates, use `.days()` to get the number of days.

[28]:
```
from datetime import datetime

print(datetime.now())
print(datetime(2018, 9, 5))
dt = datetime.strptime("20200101", "%Y%m%d")
print(dt)

print(dt.strftime("%Y-%B-%d"))
```

```
2024-02-27 21:11:20.845501
2018-09-05 00:00:00
2020-01-01 00:00:00
2020-January-01
```

[29]:
```
dt1 = datetime(2022, 1, 1)
dt2 = datetime(2019, 1, 1)
n_days = (dt1 - dt2).days + 1
n_days
```

[29]:
```
1097
```

3.2 Convert from date to datetime by combine()

[30]:
```
from datetime import date

dt = datetime.combine(date.today(), datetime.min.time())
print(dt)
```

```
2024-02-27 00:00:00
```

4. Control

We use statements to control flow to dictate the order in which the code is executed, by condition, loop, etc.

To indicate the block of code, we use indentation in Python. This is different from other languages, which use `{}` or `begin/end` to indicate the block.

4.1 Indention

- Although we can use any number of spaces or tabs. The recommended use is 4 spaces for each level.
- Indentation shall be consistent between different lines in the same block.
- In Jupyter notebook and VS Code, indentation is automatically done after `if` or `for`.
- For manual indentation, you can use `Tab` to indent and `Shift+Tab` to un-indent.

4.2 if

- `if`, `elif`, `else` are used for conditional execution.

[31]:
```
if 1 > 1 and 0 < 1:
    print("a")
    print("b")
elif 2 > 1:
    pass
else:
    print("can't reach here")
```

4.3 for

To create a loop, we use `for` statement.

- It is used to iterate over a sequence (list, tuple, string, etc.) or other iterable objects.
- It is used to execute a block of code repeatedly, use `range()`

[32]:
```
# Example of using for with a sequence

seq = "ATGCATGCGGGCC"
cc = 0
for s in seq:
    if s == "A":
        cc += 1

# There are many Python built-in functions can save us from using a for loop
# Learn them and use them to make your code more readable and efficient.
print(cc, seq.count("A"))

for a in ["MFE", "FN6815"]:
    print((a, len(a), a.find("E")))
```

```
2 2
('MFE', 3, 2)
('FN6815', 6, -1)
```

Although we can use `for` with a list `[1,2,3]` for iteration, we usually use `range()` to generate a sequence of numbers. `range()` is an object that would return value in the range one-by-one.

It can take 1, 2 or 3 integers as arguments.

- 1 argument: from 0 up to that number, $[0, a)$
- 2 arguments: from the first number to the second number, $[a, b)$
- 3 arguments: from the first number to the second number, with the step of the third number, $[a, b)$

`{python} range(3) # 0, 1, 2 range(1, 3) # 1, 2 range(1, 27, 3) # 1, 4, 7, 10, 13, 16, 19, 22, 25`

Question on range

- What's the output of `range(-3)`, `range(3, 0)`, `range(1, 27, -3)`?
- Hint: use `list(range())` to realize the range to be a list of values.

[33]:
```python
for x in range(3):
    print(x)

for _ in range(3):
    print(3)

## Since I don't need x here, use _ to represent throwaway variable.
for _ in range(3, 6):
    print(2)
```

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

The advantage of using `range()` is space and time saving. It does not generate the list of numbers in memory. It generates the number one-by-one.

4.4 List comprehension

List comprehension in Python offers a more concise way to process elements from a list, compared to the traditional for loop. Here are some advantages:

- Brevity: List comprehension provides a shorter and more readable code. Instead of writing multiple lines of code with a for loop, you can achieve the same result in a single line.
- Memory Efficiency: It uses less memory as it generates the output list without needing to store each loop's output.

[34]:
```
# use for loop
cc = []
for i in range(1, 10):
    cc.append(i**2)
print(cc)

# Use list comprehension, we avoid using append in a loop
print([i**2 for i in range(1, 10)])

# add guard `if`
print([i for i in range(10) if i % 2 == 0])
```

```
[1, 4, 9, 16, 25, 36, 49, 64, 81]
[1, 4, 9, 16, 25, 36, 49, 64, 81]
[0, 2, 4, 6, 8]
```

[35]:
```
# Multi-variable list comprehension, also called nested list comprehension
# a matrix's locations
print([(x, y) for x in range(3) for y in range(3)])

# half-matrix locations
print([(x, y) for x in range(3) for y in range(3) if x >= y])
```

```
[(0, 0), (0, 1), (0, 2), (1, 0), (1, 1), (1, 2), (2, 0), (2, 1), (2, 2)]
[(0, 0), (1, 0), (1, 1), (2, 0), (2, 1), (2, 2)]
```

[36]:
```
# For nested list comprehension, it is equivalent to the following for loop
print(
    [(x, y) for x in range(3) for y in range(2)],  # List 1
    [(x, y) for x in range(2) for y in range(3)],  # List 2
)

# Equivalent to List 1
for x in range(3):
    for y in range(2):
        print((x, y))
```

```
[(0, 0), (0, 1), (1, 0), (1, 1), (2, 0), (2, 1)]
 [(0, 0), (0, 1), (0, 2), (1, 0), (1, 1), (1, 2)]
(0, 0)
(0, 1)
(1, 0)
(1, 1)
(2, 0)
(2, 1)
```

The list comprehension syntax also extends to create *dictionary* and *set*.

[37]:
```
# A set is created when the generator expression is enclosed in curly braces.
{i for i in range(3)} == set([0, 1, 2])
```

[37]:
```
True
```

[38]:
```
# A dict is created when the generator expression contains "pairs" of the form␣
↪key:value.
{str(i): i**2 for i in range(3)}
```

[38]:
```
{'0': 0, '1': 1, '2': 4}
```

4.5 Functions

- Definition: Use the def keyword to define a function. For example, `def my_function():`.
- Inputs: A function can take inputs from its argument list and the environment it runs in. For instance, `def my_function(arg1, arg2):`.
- Default Arguments: You can provide default values for arguments. These should be placed after arguments without default values. For example, `def my_function(arg1, arg2='default'):`, but not `def my_function(arg1, arg2='default', arg3):`
- Return Statement: The `return` keyword is used to exit a function and return a result. If you want to return multiple values, you can use a tuple.

[39]:
```
def a_func(c, d=3):
    a = 1 + b + d
    return a, c


b = 2
# it uses b from the environment, so the result is 6
print(a_func(3))

a, c = a_func(3)  # receive tuple-typed result
ac = a_func(3)  # or store the tuple into one variable
(ac[0], ac[1]) == (a, c)
```

```
(6, 3)
```

[39]:
```
True
```

4.6 Matching arguments for functions

Python provides a flexible mechanism for dealing with variable-length arguments for a function. You can use the following placeholders:

- `*args` is used to scoop up a variable number of non-keyword arguments. These arguments are captured into a tuple.
- `**kwargs` is used to capture an arbitrary number of keyword arguments. These are captured into a dictionary.

Note: args and kwargs are conventional names, but you can use any valid identifier like `*myargs` or `**mykwargs`.

[40]:
```
def my_proc(*kargs, **kwargs):
    print(f"kargs: type:{type(kargs)} len:{len(kargs)}")
    print(f"kwargs: type:{type(kwargs)} len:{len(kwargs)}")
    print(kargs)
    print(kwargs)


my_proc(1, 2, 3, a=3, b=4, c="a_string")
# kargs (tuple) stores 1, 2, 3
# kwargs (dict) stores a=3, b=4, c="a_string"
```

```
kargs: type:<class 'tuple'> len:3
kwargs: type:<class 'dict'> len:3
(1, 2, 3)
{'a': 3, 'b': 4, 'c': 'a_string'}
```

When a function only has `*kargs`, or `**kwargs`.

[41]:
```
# do not allow to pass a non-keyword argument
def f1(**kwargs):
    print(f"f1: {type(kwargs), kwargs}")


try:
    f1(a=1, b=3)  # Pass
    f1(1, 3)  # Error
except TypeError as err:
    print(f"Error: {err}")


# do not allow keyword arguments
def f2(*args):
    print(f"f2: {type(args), args}")


try:
    f2(1, 3)  # Pass
    f2(a=1, b=3)  # Error
except TypeError as err:
    print(f"Error: {err}")
```

```
f1: (<class 'dict'>, {'a': 1, 'b': 3})
Error: f1() takes 0 positional arguments but 2 were given
f2: (<class 'tuple'>, (1, 3))
Error: f2() got an unexpected keyword argument 'a'
```

4.7 Another way round – Use * to expand a list and ** to expand a dict

Previously, we used `*` to create new tuple, list, dict from an existing one. We can use it to expand a tuple/list/dict to be the arguments of a function.

[42]:
```
def myFun(arg1, arg2, arg3):
    print("arg1:", arg1)
    print("arg2:", arg2)
    print("arg3:", arg3)


# Now we can use *args or **kwargs to
# pass arguments to this function :
args = ("Geeks", "for", "Geeks")
myFun(*args)

kwargs = {"arg1": "Geeks", "arg2": "for", "arg3": "Geeks"}
myFun(**kwargs)

# Use ** to expand a dictionary and then combine multiple of them.
fruit_1 = {"apple": 3}
fruit_2 = {"orange": 6}
dict(**fruit_1, **fruit_2)
```

```
arg1: Geeks
arg2: for
arg3: Geeks
arg1: Geeks
arg2: for
arg3: Geeks
```

[42]:
```
{'apple': 3, 'orange': 6}
```

4.8 Use `yield` to create customized generator function (Optional)

`yield` is used to create a generator function. It is a special type of function that remembers its state and can resume from where it left off. This is how the `range()` function works.

Below example re-creates the built-in `range()` function in Python.

[43]:
```
def range_own(end):
    start = 0
    while start < end:
        yield start
        start += 1


print(range_own(10))
print(list(range_own(10)))
```

```
<generator object range_own at 0x00000207043AC110>
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

4.9 zip

Zip operation can merge two equal-long list to a list of tuples.

```
 a    b
[ ]  [ ]
 |    |
[ ]  [ ]
 |    |
[ ]  [ ]
```

After zip(a, b),

```
[ [ ] - [ ]

  [ ] - [ ]

  [ ] - [ ] ]
```

[44]:
```
zip([1, 2, 3], ["a", "b", "c"])
```

[44]:
```
<zip at 0x2070436b340>
```

[45]:
```
print([x for x in zip([1, 2], ["a", "b", "c"])])

# or, use list()
list(zip([1, 2], ["a", "b", "c"]))
```

```
[(1, 'a'), (2, 'b')]
```

[45]:
```
[(1, 'a'), (2, 'b')]
```

zip also has a property: `zip(*zip(a, b)) == (a, b)`

1. Melting/breaking zip(a,b) down with *

```
[ ] - [ ]

[ ] - [ ]

[ ] - [ ]
```

2. And put this list of three elements back to zip, it will restore the original input of a and b

```
 a     b
[ ]  [ ]
 |    |
[ ]  [ ]
 |    |
[ ]  [ ]
```

[46]:
```
# Restore a zip
a, b = zip(*[x for x in zip([1, 2, 3], ["a", "b", "c"])])
print(a, b)
```

```
(1, 2, 3) ('a', 'b', 'c')
```

5. Class

5.1 Define a class

OOP brings good practice to organize the code. Although you will write most functions in this course, you could package them better with OOP.

Python classes utilize special methods with names starting with double underscores ___, known as dunder methods (dunder = double underline). Understanding when these methods are called allows us to customize class behavior.

- `__init__`: The constructor method, called when an instance of the class is created.
- `__del__`: The destructor method, invoked when an instance is about to be destroyed. This is useful for cleaning up resources. Usually you do not need to define them as Python has garbage collection mechanism.

All class methods should have `self` as their first argument. This refers to the instance of the class and is used to access other class members (variables and methods).

However, methods decorated with `@staticmethod` don't require `self` and can be called directly from the class name.

Example for `__init__` and `self`:

```
class FinancialInstrument:
    def __init__(self, type='swap'):
        self.type = type
    def get_price(self):
        return self.price
    def set_price(self, price):
        self.price = price
```

Example for `__del__`:

```
class AAA:
    def __init__(self):
        self.randoms = np.random.uniform(size=100000)
    def __del__(self):
        # manual deletion of the variable, if not, it will also be deleted by Python's garbage
        del self.randoms
        print("AAA")

a = AAA()
a = 1 # __del__ is triggered to release the memory of instance a
>> "AAA"
```

[47]:
```python
import numpy as np


class AAA:
    def __init__(self):
        self.randoms = np.random.uniform(size=10)

    def __del__(self):
        del self.randoms
        print("~AAA")
```

[48]:
```python
a = AAA()
del a
```

```
~AAA
```

5.2 More dunder methods

Dunder methods allow us to emulate the behavior of built-in Python objects. Here are some commonly used ones:

- `__str__` or `__repr__` (if `__str__` is not defined) are invoked by `print()`.
- `__repr__` is used by `repr()`.
- `__eq__` is used for equality comparison (`==`).
- `__add__` and `__sub__` are used for addition (`+`) and subtraction (`-`), respectively.

Implement these methods within your class to customize its interaction with built-in Python operations, similar to operator overloading in C++.

Note:

- `__str__()` should return a user-friendly string representation of the object, while `__repr__()` should return a string that `eval()` can interpret.
- `__repr__()` is intended for developers, providing a detailed representation of the object.

[49]:
```python
class City:
    def __init__(self, x, y):
        self.x = x
        self.y = y

    def distance(self, city):
        xDis = abs(self.x - city.x)  # Pass
        yDis = abs(self.y - city.y)
        distance = np.sqrt((xDis**2) + (yDis**2))
        return distance

    def __eq__(self, city):
        return self.x == city.x and self.y == city.y

    def __repr__(self):
```

```
        # This output a string like the command line of constructor. i.e. a =␣
↪City(1,2)
        return f"City ({str(self.x)},{str(self.y)})"

    def __add__(self, city):
        return self.__class__(self.x + city.x, self.y + city.y)


class CityWithStr(City):
    def __str__(self):
        return f"This city is at {self.x}, {self.y}."


a_city = City(1, 2)
b_city = City(3, 4)
print(f"b->a: {a_city.distance(b_city)}")
print(f"a->b: {b_city.distance(a_city)}")
print(f"b->b: {b_city.distance(b_city)}")
print("--This is from class City")
print(a_city)

print("--This is from class CityWithStr")
c_city = CityWithStr(1, 2)
print(c_city + c_city)

# __repr__() shall return a string that eval() knows how to interpret.
eval(repr(a_city))
```

```
b->a: 2.8284271247461903
a->b: 2.8284271247461903
b->b: 0.0
--This is from class City
City (1,2)
--This is from class CityWithStr
This city is at 2, 4.
```

[49]:
```
City (1,2)
```

[50]:
```
class Vector:
    def __init__(self, x, y):
        self.x = x
        self.y = y

    def __eq__(self, other):
        return self.x == other.x and self.y == other.y

    def __add__(self, other):
        return Vector(self.x + other.x, self.y + other.y)
```

```
    def __minus__(self, other):
        return Vector(self.x - other.x, self.y - other.y)

    def __str__(self):
        return "({0}, {1})".format(self.x, self.y)

    def __mul__(self, other):
        return self.x * other.x + self.y * other.y


v1 = Vector(3, 4)
v2 = Vector(4, -3)
print(v1 * v2)

print(Vector(1, 2) == Vector(1, 2))
```

```
0
True
```

5.4 Python internal methods, like *__dict-*.

- Convert a `dict` to `object`: turning the keys to the object's attributes.

`dict['a'] = 1 => dict.a = 1`

[51]:
```
class Dict2Obj:
    def __init__(self, entries):
        self.__dict__.update({str(k): v for k, v in entries.items()})


converted = Dict2Obj({"an_int": 156, "b_string": "Good morning"})
print(converted.an_int)
print(converted.b_string)
```

```
156
Good morning
```

6. Handling Error Message in Python

6.1 Handling Error with `try...except...finally`

Numerical code could raise exception due to invalid input or ineffective algorithm that result in division by zero, overflow or wrong input for the math functions.

- 1 / 0
- np.log(-3)

Handling error could help us to capture more information when the error happens to help us to avoid the problem.

We can find the specific type of error by trying the wrong input first.

Example:

Try below expression in Python

```
{python} 1 / 0
```

You will see below message

```
Traceback (most recent call last):
    File "<stdin>", line 1, in <module>
ZeroDivisionError: division by zero
```

Note that in Jupyter notebook, it has slightly more information.

```
---------------------------------------------------------------------------
ZeroDivisionError                         Traceback (most recent call last)
Cell In[120], line 1
----> 1 1/0

ZeroDivisionError: division by zero
```

And

“‘{python} import numpy as np np.seterr(all=‘raise’) np.log(-10)

FloatingPointError Traceback (most recent call last) 1 import numpy as np 2 np.seterr(all=‘raise’) —-> 3 np.log(-10)

FloatingPointError: invalid value encountered in log “‘

We can create a try…except to capture different errors in a section of code.

```
{python} try:     v1 = 1 / alpha     v2 = np.log(v1 * beta) except
ZeroDivisionError as err:     print((f'ZeroDivisionError', alpha, beta, err))
raise except FloatingPointError as err:     print((f'FloatingPointError', alpha,
beta, err))     raise
```

The Python exception is similar to the `try...catch` in C++ or Java.

- `raise` is for `throw` in C++ or Java.
- `Exception` is the base class for all exceptions.
- After `except`, we shall follow up with a specific type of error to catch. Add `as err` to save the error as variable `err`, so we can print it out.

6.2 Read and interpret error messages

If an error happens in Python, interpreter would print out a trace to tell what went wrong.

```
Traceback (most recent call last):
  File "c:\dev\process_material.py", line 127, in <module>
    res_collect.append(process.process_run(cob_date, now_dt, k, v))
  File "c:\dev\process\process_run.py", line 26, in process_run
```

```
    err = eval(f"{k['process_run']}(cob_date, now_dt, rniv_process, k, out_k)")
  File "<string>", line 1, in <module>
  File "c:\dev\process\run_basis.py", line 23, in run_basis
    tenor_mapping, ac_info, ac_total = load_mkt(cob_date, **subdict(k, ['process_class', 'cache
  File "c:\dev\process\process_load.py", line 25, in load_mkt
    tenor_mapping = pd.read_pickle(os.path.join(cache_dir, f"tenor_mapping.pkl{ext}"), compress
  File "C:\ProgramData\workspace\m\mini3\lib\site-packages\pandas\io\pickle.py", line 170, in r
    f, fh = get_handle(fp_or_buf, "rb", compression=compression, is_text=False)
  File "C:\ProgramData\workspace\m\mini3\lib\site-packages\pandas\io\common.py", line 384, in g
    f = gzip.open(path_or_buf, mode)
  File "C:\ProgramData\workspace\m\mini3\lib\gzip.py", line 53, in open
    binary_file = GzipFile(filename, gz_mode, compresslevel)
  File "C:\ProgramData\workspace\m\mini3\lib\gzip.py", line 163, in __init__
    filets = self.myfileobj = builtins.open(filename, mode or 'rb')
FileNotFoundError: [Errno 2] No such file or directory: 'cache\\tenor_mapping.pkl.gz'
```

Understanding Python Tracebacks When a Python program encounters an error, it generates a traceback. Here's how to interpret it:

1. **Hierarchy of Function Calls**: A program is a collection of functions, with outer functions calling inner ones.
2. **Error Propagation**: If an error occurs, Python passes it up the calling hierarchy until it finds an error handler (a try…except block). If no handler is found, the error reaches the top level and a traceback is printed. Each line beginning with 'File' represents a level in the calling hierarchy, with deeper levels lower in the traceback.
3. **Identifying User Code**: Divide the traceback into sections representing your code and system library code. The last line of your code is usually a good indicator. For example:

```
File "c:\dev\process\process_load.py", line 25, in load_mkt
    tenor_mapping = pd.read_pickle(os.path.join(cache_dir, f"tenor_mapping.pkl{ext}"), compress
```

4. **Reading the Error Message**: The final line of the traceback is the error message. For example:

```
FileNotFoundError: [Errno 2] No such file or directory: 'cache\\tenor_mapping.pkl.gz'
```

5. **Interpreting the Error**: Combine steps 3 and 4 to understand the error. In this case, 'When my code tries to read the tenor_mapping.pkl file, the system can't find it.'

Remember, don't panic when you see a traceback. Read it line by line to understand what went wrong.

6.3 Use Error/Exception for your good

Exception helps to stop the program if we have unexpected input.

I use these two often: NotImplementedError, ValueError

1. NotImplementedError

   ```
   if type == 'call':
   ```

```
    ...
elif type == 'put':
    ...
else:
    raise NotImplementedError(
        f"Options {type} other than call/put is not implemented yet.")
```

2. ValueError: General error. Use this when you just want to quit the program with a message.

```
if rf < 0:
    raise ValueError('interest rate shall not be less than zero.')
```

6.4 Measure running time

6.4.1 In a code file

[52]:
```
# time the program
import logging
import time

if __name__ == "__main__":
    logging.basicConfig(
        level=logging.INFO,
        format="%(asctime)s - %(name)s - %(levelname)s - %(message)s",
    )
    start = time.process_time()
    end = time.process_time()
    logging.info("Elapsed process time {}".format(end - start))
```

```
2024-02-27 21:11:21,166 - root - INFO - Elapsed process time 0.0
```

6.4.2 In Jupyter notebook

[53]:
```
# timeit in Jupyter Notebook
# -r: how many times to repeat, -n how many times to execute
# Below example is to show how many times it runs.
# Usually we avoid using print() because it slows things down other than the␣
 ↪calculation.

%timeit -r2 -n3 print('2x3=6')

# Faster without print()
%timeit -r2 -n3 ('2x3=6')
```

```
2x3=6
2x3=6
2x3=6
2x3=6
2x3=6
2x3=6
The slowest run took 16.57 times longer than the fastest. This could mean that
an intermediate result is being cached.
```

```
26.3 µs ± 23.4 µs per loop (mean ± std. dev. of 2 runs, 3 loops each)
133 ns ± 33.3 ns per loop (mean ± std. dev. of 2 runs, 3 loops each)
```

6.5 Use `logging` module to print out information.

```
{python} import logging logging.basicConfig(level=logging.INFO,
format='%(asctime)-15s %(levelname)-8s %(message)s') logging.info(f'[function
name] {variable} is good')
```

6.6 Use `__name__` in file

When we write code in .py file (not in Jupyter notebook), we shall use `if __name__ == '__main__'` to put our code to be run. This is like C++'s `int main()`.

"'{python} import os import numpy as numpy import logging

def a_func(): pass

if **name** == '**main**': # Place the logging setting here. logging.basicConfig(level=logging.INFO, format='%(asctime)-15s %(levelname)-8s %(message)s')

```
"""
Start to put your own lines
"""
logging.info(f'[function name] {variable} is good')
```

```
BAD example
```

import os import numpy as numpy

def a_func(): pass

" " " Start to put your own lines " " " print(" ") "'

6.7 Making a module

As soon as you start writing multiple Python source files, you can import and reuse your functions between - but beware of cyclic dependency. For example, if our solver function resides in a module file `decay.py`, another program may reuse of the function either by

In decay.py file

```
# decay.py
def solver(I, a, T, dt, theta):
    pass
```

In use_decay.py file

```
# use_decay.py
from decay import solver
u, t = solver(I=1, a=2, T=4, dt=0.2, theta=0.5)
```

First, create a directory, a folder is a module. All .py files under a directory belongs to the module.

Second, you shall not have "floating code", i.e. code block not in a function or class, except for variable definition.

Third, use `if __name__ == __main__` for where the program shall start running. Those code would not get to run when code is imported as a module.

Fourth, avoid cyclic dependency, In `a.py`, you have `import b` while in `b.py`, you have `import a`.

Note:

1. When you import a module in Jupyter notebook, if you have any changes, you need to restart the kernel to re-import the module.
2. You could use the autoreload magic from IPython to reload the module automatically. Reference

`{python} In [1]: %load_ext autoreload In [2]: %autoreload 2`

7. Assignment

Exercise 1: Find crossing points of two graphs

Consider two functions $f(x) = x$ and $g(x) = x^2$ on the interval $[-4, 4]$.

Write a program that, by trial and error, finds approximately for which values of x the two functions cross, i.e., $f(x) = g(x)$. Do this by considering `N` equally distributed points on the interval, at each point checking whether $|f(x) - g(x)| < \epsilon$, where ϵ is some small number. Let N and ϵ be user input to the function and let the result be printed to screen. Run your function run with $N = 400$ and $\epsilon = 0.01$. Explain the output from the function. Finally, try also other values of N, keeping the value of ϵ fixed. Explain your observations.

Exercise 2: Count occurrences of a string in a string

In the analysis of genes one encounters many problem settings involving searching for certain combinations of letters in a long string. For example, we may have a string like

```
gene = 'AGTCAATGGAATAGGCCAAGCGAATATTTGGGCTACCA'
```

We may traverse this string, letter by letter, by *for loop* for letter in gene. Use below code as the template

```
for g in gene:
    ...
```

The length of the string is given by `len(gene)`, so an alternative traversal over an index `i` is `for i in range(len(gene))`. Letter number `i` is reached through `gene[i]`, and a substring from index i up to, but not including j, is created by `gene[i:j]`.

a) create a class Gene like a str to store the list of gene, with below expected behavior.

g = Gene('ATGC') print(g) # 'ATGC' len(g) # 4

b) Write a function `freq(self, letter, text)` that returns the frequency of the letter letter in the string text, i.e., the number of occurrences of letter divided by the length of text. Call the function to determine the frequency of C and G in the gene string above. Compute the frequency by hand too.

c) Write a function `pairs(self, letter, text)` that counts how many times a pair of the letter letter (e.g., GG) occurs within the string text. Use the function to determine how many times the pair AA appears in the string gene above. Perform a manual counting too to check the answer. Note but that pair shall not overlap each other, AAA doesn't give count of 2 but 1.

d) Write a function `mystruct(self, text)` that counts the number of a certain structure in the string text. The structure is defined as G followed by A or T until a double GG (inclusive of GG). Perform a manual search for the structure too to control the computations by `mystruct`. The identified structure will not have overlapping, or embedding.

Appendix: timestamp

[54]:
```
from datetime import datetime

print(f"Generated on {datetime.now()}")
```

```
Generated on 2024-02-27 21:11:21.178395
```