Lecture 4:

In this lecture, we explore more details about Python language.

Defining functions

We can define our function by using `def` . A function contains a block of code that only executes when we call it. A function has a name for us to call. We can also pass variables into the function, they are called **parameters**.

In [ ]:
```python
def square(x):
    return x*x
```

In [ ]:
```python
square(3)
```

Exercise time: Try to write a function that given two number `a` and `b` , result the `sum` result.

In [ ]:
```python
def add(a, b):
    return None

print(add(1, 3))
print(add(100, 1))
```

Expected Result
4

101|

Comments and DocString

We can store multiple lines of string by using three quotes. Any string within a pair of `""""` .

For example, we can store a multiple-lines strings as following code example:

In [4]:
```python
source = """Alice was beginning to get very tired of sitting by her sister
So she was considering in her own mind (as well as she could, for the hot d
There was nothing so very remarkable in that; nor did Alice think it so ver

paragraphs = source.split("\n")
len(paragraphs)
```

Out[4]:
```
3
```

Beyond using multiple-lines in variables, we can also use it as **DocString**. DocString is a description on function, on module, on file. It provide a brief on what the block of code does.

In [ ]:
```python
# Comment begins with #

"""
Multi-lines comments
by using 3 quotes.
"""

def hello(message):
  """DocString:
      3-quote comments as first line of functions
      is called DocString.
  """
  return f"Hello {message}"
```

In [ ]:
```python
def square(x):
  """Return a square of input"""
  return x*x


help(square)

print(square.__doc__)
```

Exercise time: Given the following function that print today's date, try to write a Doc String for it.

In [ ]:
```python
def today():
    '''WRITE_YOUR_DOC_STRING_HERE'''
    import datetime
    return datetime.date.today().isoformat()
```

In [ ]:
```python
today.__doc__
```

In [8]:
```python
help(today)
```

```
Help on function today in module __main__:

today()
    WRITE_YOUR_DOC_STRING_HERE
```

List comprehension

We can create a list by transforming another list.

```
In [9]:  [x**2 for x in range(10)]

Out[9]:  [0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

Furthermore, we can have condition to include only the item we want into the new list.

```
In [1]:  [x**2 for x in range(10) if x**2 < 50]

Out[1]:  [0, 1, 4, 9, 16, 25, 36, 49]
```

Exercise time: Try creating a list that contains sequence of 10, 20, 30, ... 100.

```
In [ ]:
```

Expected Result

[10, 20, 30, 40, 50, 60, 70, 80, 90, 100]

Exercise time: Try creating a list that contains sequence of **square** of 10, 20, 30, ... 100.

```
In [ ]:
```

Expected Result

[100, 400, 900, 1600, 2500, 3600, 4900, 6400, 8100, 10000]

Using list comprehension with existing data list.

```
In [12]:  scores = [34, 65, 45, 67, 78, 56, 80]

          [x for x in scores if x >= 60]

Out[12]:  [65, 67, 78, 80]
```

Another example, given the following dataset.

```
In [4]:  contacts = [
             {'name': 'Thomas', 'email': 'thomas@example.com', 'tel':'66661234'},
             {'name': 'Susanna', 'email': 'susanna@example.com', 'tel':'66334455'},
             {'name': 'Dick', 'email': 'dick@example.com', 'tel':'66664321'},
             {'name': 'Tom', 'email': 'tom@example.com', 'tel':'67891230'},
         ]
```

Filtering by name

```
In [14]:  [x for x in contacts if x['name'][0]=='T' ]

Out[14]:  [{'name': 'Thomas', 'email': 'thomas@example.com', 'tel': '66661234'},
           {'name': 'Tom', 'email': 'tom@example.com', 'tel': '67891230'}]
```

Getting all emails from the list

```
In [15]:  emails = [ c['email'] for c in contacts]
          emails

Out[15]:  ['thomas@example.com',
           'susanna@example.com',
           'dick@example.com',
           'tom@example.com']
```

Displaying emails list in a string

```
In [17]:  print(", ".join(emails))

          thomas@example.com, susanna@example.com, dick@example.com, tom@example.com
```

Exercise time: Try to get all names into a list from the `contacts` dataset.

```
In [ ]:
```

Expected Result

['Thomas', 'Susanna', 'Dick', 'Tom']

Exercise time: Try to get all tels from the `contacts` dataset. We want the output to be comma-separated.

```
In [ ]:
```

Expected Result

66661234,66334455,66664321,67891230

Module import

We can import modules into the python script to extend the abilities. In our previous examples, we have imported `datetime` module, `random` module and a 3rd party `docx` module.

We can also import a custom external python script. Given the `test_helper.py` in the same folder of the python script, we can import it for us to use.

```
In [18]:  import test_helper

In [19]:  test_helper.today()

Out[19]:  '2020-08-13'
```

We may alias the module name by using `import...as...` .

```
In [20]:  import test_helper as helper

In [21]:  helper.today()

Out[21]:  '2020-08-13'
```

We may use `from...import...` syntax to import individual function from a module.

In [22]:

```python
from test_helper import download_file

download_file("https://xml.smg.gov.mo/c_7daysforecast.xml")
```

Easter Egg: import this

Let's try `import this` . Python gives us a poem of the Zen of Python. PEP20 has reference on this too.

In [23]:

```python
import this
```

```
The Zen of Python, by Tim Peters

Beautiful is better than ugly.
Explicit is better than implicit.
Simple is better than complex.
Complex is better than complicated.
Flat is better than nested.
Sparse is better than dense.
Readability counts.
Special cases aren't special enough to break the rules.
Although practicality beats purity.
Errors should never pass silently.
Unless explicitly silenced.
In the face of ambiguity, refuse the temptation to guess.
There should be one-- and preferably only one --obvious way to do it.
Although that way may not be obvious at first unless you're Dutch.
Now is better than never.
Although never is often better than *right* now.
If the implementation is hard to explain, it's a bad idea.
If the implementation is easy to explain, it may be a good idea.
Namespaces are one honking great idea -- let's do more of those!
```

Optional: Dunder methods

Dunder method stands for Double Underscore Method.

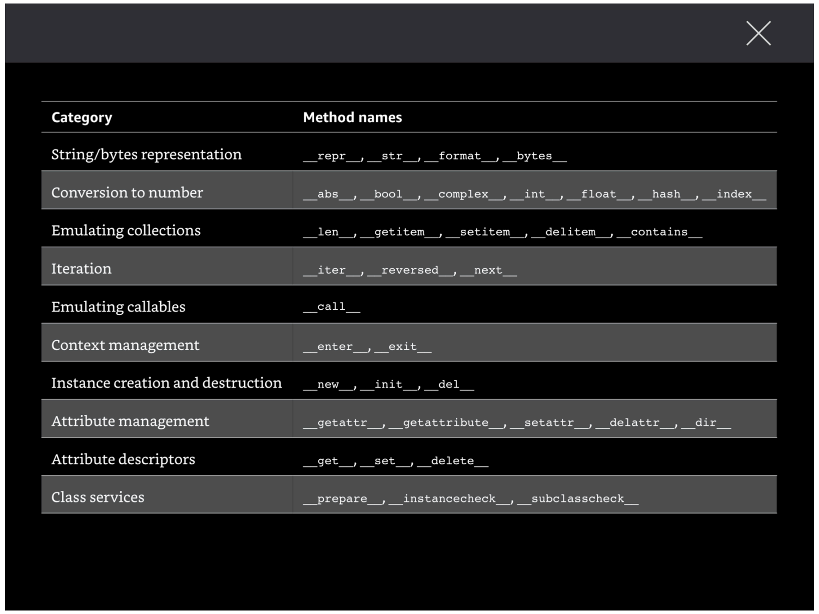
They are special usage for Python behind-the-scene.

Take `with` as example, it automatically calls the method `__enter__` and `__exit__` . When existing the indentation block, the `__exit__` is called.

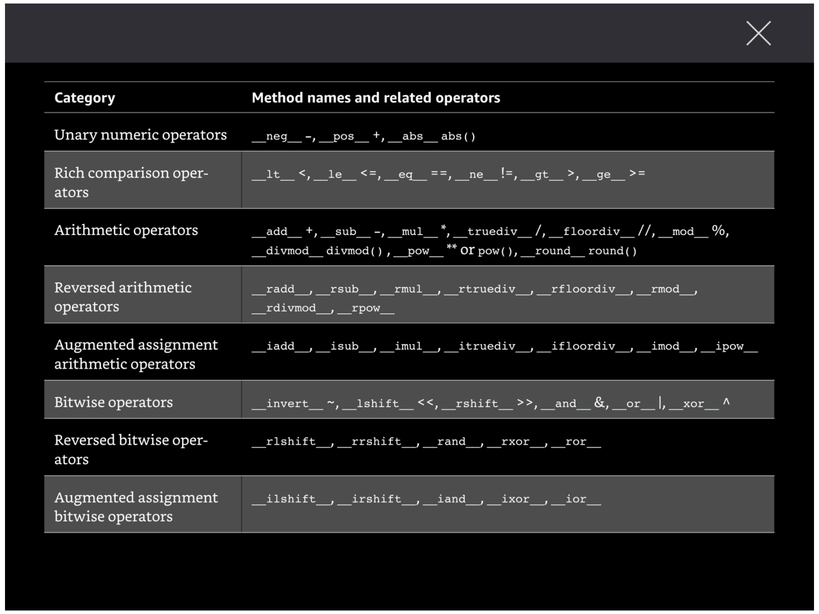
That's why when we `open` a file by using `with` block, Python handles the opening and closing for us.

Another example of dunder method is the `__doc__`

List of dunder methods:



Category	Method names
String/bytes representation	`__repr__, __str__, __format__, __bytes__`
Conversion to number	`__abs__, __bool__, __complex__, __int__, __float__, __hash__, __index__`
Emulating collections	`__len__, __getitem__, __setitem__, __delitem__, __contains__`
Iteration	`__iter__, __reversed__, __next__`
Emulating callables	`__call__`
Context management	`__enter__, __exit__`
Instance creation and destruction	`__new__, __init__, __del__`
Attribute management	`__getattr__, __getattribute__, __setattr__, __delattr__, __dir__`
Attribute descriptors	`__get__, __set__, __delete__`
Class services	`__prepare__, __instancecheck__, __subclasscheck__`



Category	Method names and related operators
Unary numeric operators	`__neg__ -`, `__pos__ +`, `__abs__ abs()`
Rich comparison operators	`__lt__ <`, `__le__ <=`, `__eq__ ==`, `__ne__ !=`, `__gt__ >`, `__ge__ >=`
Arithmetic operators	`__add__ +`, `__sub__ -`, `__mul__ *`, `__truediv__ /`, `__floordiv__ //`, `__mod__ %`, `__divmod__ divmod()`, `__pow__ **` or `pow()`, `__round__ round()`
Reversed arithmetic operators	`__radd__, __rsub__, __rmul__, __rtruediv__, __rfloordiv__, __rmod__, __rdivmod__, __rpow__`
Augmented assignment arithmetic operators	`__iadd__, __isub__, __imul__, __itruediv__, __ifloordiv__, __imod__, __ipow__`
Bitwise operators	`__invert__ ~`, `__lshift__ <<`, `__rshift__ >>`, `__and__ &`, `__or__ |`, `__xor__ ^`
Reversed bitwise operators	`__rlshift__, __rrshift__, __rand__, __rxor__, __ror__`
Augmented assignment bitwise operators	`__ilshift__, __irshift__, __iand__, __ixor__, __ior__`

Table clipped from Python Tricks.

Optional: Lambda function

Lambda function is a single line function.

In [24]:
```
def square(x):
    return x*x
```

In [25]:
```
square2 = lambda x: x*x
```

In [26]:
```
square2(3)
```

Out[26]:
```
9
```

Why single line of function?

Useful when we need an ad-hoc function to quick converting value, which can be expressed in one line of calculation.

We often use it with `map`

In [27]:
```
list(map(lambda x: x*x, range(10)))
```

Out[27]:
```
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

But indeed, we can use list comprehension to better express this transform:

Object-Oriented Programming

The concept of Object-oriented programming method is to think of the system components as objects. Objects communicate to each other. Objects have their own attributes and behaviors. Objects has definition. The definition defines what attributes this object should contain and what functions the object is able to do. Then we create instances from the definition.

Here is an example of defining Car class and extend the `Car` ability with `Motorcycle` and `ElectronicCar`.

In [28]:
```
class Car:
    '''A basic Car definition.'''
    wheels = 4
    gas = 100
    speed = 0
    def __repr__(self):
        return '{}: speed:{} gas:{} wheels:{}'.format(type(self).__name__,
    def refill(self):
        self.gas = 100
    def forward(self):
        self.speed += 10
        self.gas -= 1
    def brake(self):
        self.speed -= 10
    def stop(self):
        self.speed = 0
```

In [29]:
```
class Motorcycle(Car):
    '''Definition of Motocycle.'''
    wheels = 2

class ElectronicCar(Car):
    '''Electronic car definition'''
    gas = 0
    battery = 50
    def refill(self):
        self.battery = 500
    def forward(self):
        self.speed += 10
        self.battery -= 1
```

Using the `Car`, `ElectronicCar` and `Motorcycle`.

In [30]:
```
# use the Car
carA = Car()
print("carA: ", carA)
carA.forward()
print("carA: ", carA)
carA.refill()

carB = ElectronicCar()
carB.forward()
carB.forward()
print("carB: ", carB)
carB.refill()

carC = Motorcycle()
print("carC: ", carC)
carC.wheels = 3
print("carC: ", carC)
```

```
carA:  Car: speed:0 gas:100 wheels:4
carA:  Car: speed:10 gas:99 wheels:4
carB:  ElectronicCar: speed:20 gas:0 wheels:4
carC:  Motorcycle: speed:0 gas:100 wheels:2
carC:  Motorcycle: speed:0 gas:100 wheels:3
```

Date Helper class example

Version 1

In [31]:
```
import datetime

class DateHelper:
    '''Some helper functions to quickly calculate date.'''
    def today(self):
        '''Return today in YYYY-MM-DD format.'''
        date = datetime.date.today()
        return date.isoformat()
```

Version 2

In [32]:
```python
import datetime

class DateHelper:
    '''Some helper functions to quickly calculate date.'''

    today_date = datetime.date.today()

    def today(self):
        '''Return today in YYYY-MM-DD format.'''
        return self.today_date.isoformat()
    def tomorrow(self):
        '''Return tomorrow in YYYY-MM-DD format.'''
        date = self.today_date + datetime.timedelta(days=1)
        return date.isoformat()
    def yesterday(self):
        '''Return yesterday in YYYY-MM-DD format.'''
        date = self.today_date + datetime.timedelta(days=1)
        return date.isoformat()
```

Version 3

In [33]:
```python
import datetime

class DateHelper:
    '''Some helper functions to quickly calculate date.'''

    today_date = datetime.date.today()

    def days_later(self, days):
        '''Return days later in YYYY-MM-DD format.'''
        date = self.today_date + datetime.timedelta(days=days)
        return date.isoformat()
    def days_ago(self, days):
        '''Return days ago in YYYY-MM-DD format.'''
        date = self.today_date - datetime.timedelta(days=days)
        return date.isoformat()
    def today(self):
        '''Return today in YYYY-MM-DD format.'''
        return self.today_date.isoformat()
    def tomorrow(self):
        '''Return tomorrow in YYYY-MM-DD format.'''
        return self.days_later(1)
    def yesterday(self):
        '''Return yesterday in YYYY-MM-DD format.'''
        return self.days_ago(1)
```

Using the Date Helper

In [6]:
```python
from date_helper_v4 import DateHelper

helper = DateHelper()
print(helper.today())
print(helper.tomorrow())
print(helper.days_later(365))
print(helper.days_ago(10))
```

```
2020-08-13
2020-08-14
2021-08-13
2020-08-03
```

Iteration and Generator

We have learned that `range` creates a sequence of numbers.

In [ ]:
```python
for item in range(10):
    print(item)
```

range is actually an iterator:

In [ ]:
```python
iterator = iter(range(3))
print(next(iterator))
print(next(iterator))
print(next(iterator))
# print(next(iterator)) # Raise StopIteration Error
```

Repeater

Here is an example of defining our own iterator class.

In [ ]:
```python
class Repeater:
    def __init__(self, value):
        self.value = value
    def __iter__(self):
        return self
    def __next__(self):
        return self.value
```

Please uncomment the following line to run it. Beware that the following code never ends. You have to manually interrupt it.

In [ ]:
```python
#for item in Repeater('Keep saying Hello'):
#    print(item)
```

Repeater with max_repeat boundary

We improve the above example by setting a repeat limit. When the limit reaches, the `Repeater` raises the `StopIteration` event. For-loop syntax will then know when to end the loop.

In [ ]:

```python
class Repeater:
    def __init__(self, value, max_repeat=1):
        self.value = value
        self.count = 0
        self.max_repeat = max_repeat
    def __iter__(self):
        return self
    def __next__(self):
        if self.count >= self.max_repeat:
            raise StopIteration
        self.count += 1
        return self.value

# Using it
for item in Repeater('Keep saying Hello',10):
    print(item)
```

Generators

Another way to create iteration is to use `yield` . This is also known as *Generator*.

In [ ]:

```python
def repeat_twice(value):
    yield value
    yield value
```

In [ ]:

```python
for item in repeat_twice('I say only once, do I?'):
    print(item)
```

Repeater by using `yield` :

In [ ]:

```python
def repeater(value, max_repeat=1):
    count = 0
    while count < max_repeat:
        count += 1
        yield value

for item in repeater('Hey Yield', 3):
    print(item)

print(list(repeater('Hey Yield', 3)))
```

Example: Dice Randomizer

In [ ]:

```python
import random

def dice_randomizer(max_repeat=1):
    count = 0
    while count < max_repeat:
        count += 1
        yield random.randint(1,6)
```

In [ ]:

```python
list(dice_randomizer(10))
```

Example: Using yield to convert value of a given list

In [ ]:

```python
import math

def convert_by_formula(sequence):
    for x in sequence:
        yield math.sqrt(x[0]**2 + x[1]**2)
```

In [ ]:

```python
source = [(1,2),(2,2),(3,3),(5,6),(8,9)]
print( list( convert_by_formula(source) ))
```

Note that we can use Pandas to improve the performance of the above calculation.

Summary

We learned the behind-the-scene of the Python programming language.

- Defining functions
- Comment and DocString
- Lambda function
- List comprehension
- Dunder methods
- Object-Oriented Programming
- Date Helper Example
- Iteration and Generator
- Repeater
- Generators