Divide and Conquer

We can divide the code block into functions, modules, files and even folders. By separating the code blocks, we focus on specific concern each time. By grouping different functions together, we can better organize our code for future reference and readability.

Functions

```
def square(x):
  """Return a square of input"""
  return x*x
```

List Comprehension

```
[x**2 for x in range(10)]
# [0, 1, 4, 9, 16, 25, 36, 49, 64, 81]

[x**2 for x in range(10) if x**2 < 50]
# [0, 1, 4, 9, 16, 25, 36, 49]
```

Another example on list comprehension

Given the following contacts dataset.

```
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 data with condition

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

Getting specific attributes from dataset.

```
tels = [ c['tel'] for c in contacts ]
print( ",".join(tels) )
```

Class example

```
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 DateHelper class

helper = DateHelper()
print(helper.today()) # 2020-08-13
print(helper.tomorrow()) # 2020-08-14
print(helper.days_later(365)) # 2021-08-13
print(helper.days_ago(10)) # 2020-08-03
```

Modules import

Importing a built-in module.

```
import datetime
```

Import a file named *test_helper.py* in same folder.

```
import test_helper
```

Import a file and rename the module.

```
import test_helper as helper
```

Import everything from a module.

```
from test_helper import *
```

Import a 3rd party module and rename it.

```
import pandas as pd
```

The *import this* easter egg (PEP20)

```
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!

Dunder methods

Double underscore methods. They are special usage for Python behind-the-scene.

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__`

The table is clipped from Python Tricks book.