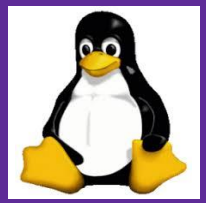
Introduction to Computation

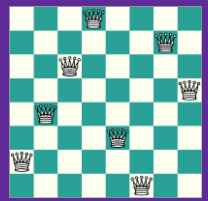
Autumn, 2023

Prof. Fan Cheng

Shanghai Jiao Tong University

chengfan85@gmail.com
https://github.com/ichengfan/itc

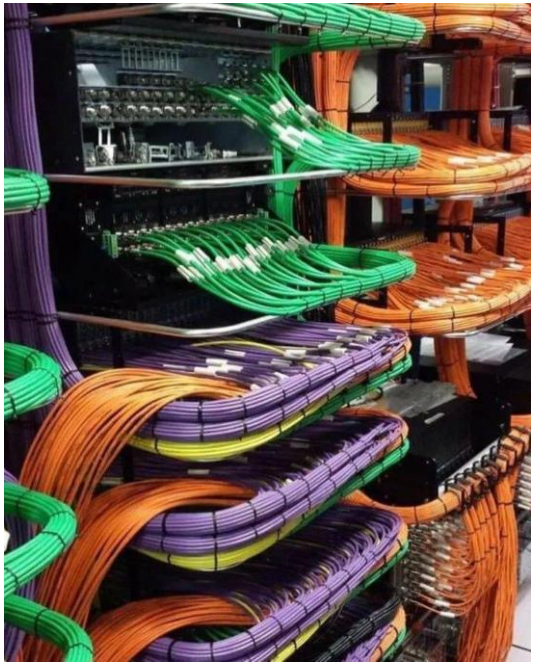




9 10

Outline

- Class
  - Inheritance
- Module
- Interpreter



清晰、有条理、减少bug

Class

Module, Class and Function: WHY?

Microsoft's Win10 has over 50,000,000 lines of codes

- **Name conflicts in variables and functions**
- **How to cooperate with other programmers**
- **How to maintain such a huge project**

We need a mechanism to deal with the complicated problems in real world

How to mange a city with a population of 26,000,000 (e.g., Shanghai)

- 分而治之：中央政府—省—市
- 城市分为各个部门：教育、卫生、环保等等 (Module)
- 部门：提供各种服务 (Class)

In Python

- Divide and conquer: module – class – data and function
  - 文件操作、网络模块、数据库、操作系统、科学计算
- Collaboration: modules can call each other and reuse their codes

根据问题的需求，将数据和函数封装（encapsulation）成类，把相关的类打包封装成模块

有结构，有组织，有协作，不是一盘散沙

Questions From Past Exercise

1. When we study trigonometry, we use a, b, c to denote the lengths of three edges of a triangle
2. In the same Python file, we also use a, b, c to denote coefficients of the quadratic polynomial with one variable
    - There will be some conflicts between a, b, c in two scenarios
3. We may also use a, b, c in a tuple unpack: a, b, c = t

- It is very messy for a single Python file. Chaos many lead to disaster
- We need a mechanism (Python Grammar) to resolve these issues
    - Encapsulate a, b, c and operations in a triangle
    - Encapsulate a, b, c and operations in a quadratic polynomial
    - Keep both (a, b, c) and operations above isolated from a, b, c in the tuple unpack
- We may also hope to
    - Reuse the code in the packs of Triangle and Polynomial
- That's what we called class
    - Data: Variables
    - Method: Member functions

Class

Recall Lec. 3

- 数据类型 int, float, complex, str
- Python中，不同类型type()就是不同的class
- class: 把数据和函数打包在一起，就构成一个class。好处之一是：可以和其它的数据和函数隔离开
  - 在三角形研究中，x, y, z 是三条边，可以定义三角形相关的函数
  - 在代数问题中，x, y, z是多项式的变量，可以定义多项式相关的函数
  - 为了避免同一文件中，可能的x, y, z冲突，用class将三角形和多项式分别打包为类，隔离开
- class中的函数和变量属于这个类所特有，不会和外面的同名函数或者变量冲突
- 用法 x.func(parm)
  - . 表示func是x中的函数，不是其它地方的 (先学会用，具体原理在第9讲)

```
1  print(type(1), type(1.0), type("1"), type(1j))
2
3  msg = "hello world"
4  print(msg.count('o'))
```

```
<class 'int'> <class 'float'> <class 'str'> <class 'complex'>
2
```

https://docs.python.org/3/library/math.html

Class and Object

类(class): 具有相同属性(attribute)和行为(method)的个体的集合

- 类的一个实例(instance)叫做对象(object)：
  - 类和对象的关系: 集合与个体
  - 中国人：姚明。球星：Messi。动物：狗。狗：哈士奇
- Python自带的类: str, int, float, list, tuple, dict, set
  - a = “hello” # a 是str的一个对象(实例)
  - b = 123 # b 是int的一个对象
  - c = [1, -1, [1]] # c 是list的一个对象
  - d = {1, 2, 3} #d 是set的一个对象
- 类方法，所有对象都具有的方法(以txt = “hello world”为例)
  - txt.find(‘o’)
  - txt.count(‘l’)
- 同义词
  - 属性、变量、成员变量
  - 函数、方法、行为、成员函数

物以类聚，人以群分
类class，对象object，实例instance
类：打包封装 数据+函数

Class and Type

- Values have their own types (class)
- Variables are defined by “=”: assignment expression
  - The type of a variable could be changed in python
- Functions are also a kind of data

```python
def test_type(x):
    print(type(x))


test_type(123)
test_type(12.3)
test_type("123")
test_type([1, 2, 3])
test_type(print)
test_type(test_type)

my_func = print  # no () after print
my_func("Hello world")

my_func = len
my_func([1, 2, 3])

my_func = test_type
my_func({1: 2})
```

Type == Class

```
<class 'int'>
<class 'float'>
<class 'str'>
<class 'list'>
<class 'builtin_function_or_method'>
<class 'function'>
Hello world
<class 'dict'>
<class '__main__.Empty'>
<class 'type'>
<class '__main__.Empty'>
```

Keyword: class

- class ClassName:
  # begin your code here
- Initialize an object: x = ClassName()

- Class Empty is the simplest class. All its details are default by python

```
111  class Empty:
112      pass
113
114  e = Empty()
115
116  print(e)
117  print(type(e))
118  print(id(e))
```

规范：ClassName

```
<__main__.Empty object at 0x000002046A02A2B0>
<class '__main__.Empty'>
2217981682352
```

```
1   a = int()
2   b = float()
3   c = str()
4   d = list()
5   e = tuple()
6   f = dict()
7   g = set()
8
9   print(a, b, c, d, e, f, g)
10  print(type(a), type(b), type(c), type(d), type(e), type(f), type(g))
```

```
1  0 0.0  [] () {} set()
2  <class 'int'> <class 'float'> <class 'str'> <class 'list'> <class 'tuple'> <class 'dict'> <class 'set'>
```

Class and Encapsulation (封装)

- Class := attributes + methods
  - Attribute：属性（变量），数据，例如三角形的三条边长度
  - Method：方法（函数），处理数据的各种函数，例如计算面积，计算各个角度，计算外接圆、内切圆等等
    - area(), angles()……
- class Triangle:

  ……
- Usage of attributes and methods: object.attr or object.method()
  - x = Triangle() print(x.a) print(x.area())
  - y = Triangle() print(y.a) print(y.area())
  - (x, y 表明范围, a和area表明操作)
    上海市. 闵行区. 东川路. 800号

```
c = str("Class and Object")
print(c.upper(), c.lower())
l = [123, 3.14, [1, 1, 1]]
print(l)
l.reverse()
print(l)
```

```
CLASS AND OBJECT class and object
[123, 3.14, [1, 1, 1]]
[[1, 1, 1], 3.14, 123]
```

. ： 表示从属关系

“.” operator

- x.y: y is a variable inside the scope of x. “.” could distinguish x.y from the y outside x
- x.y could also be defined by “=’

```
4208  class Empty:
4209      pass
4210
4211  def test_id(a, b, c):
4212      print("{} {} {}".format(id(a), id(b), id(c)))
4213
4214
4215  e1 = Empty()
4216
4217  e1.a = 1
4218  e1.b = 1
4219  e1.c = 1
4220
4221  e2 = Empty()
4222
4223  e2.a = 1
4224  e2.b = 1
4225  e2.c = 1
4226
4227  a, b, c = 1, 1, 1
4228  test_id(a, b, c)
4229  test_id(e1.a, e1.b, e1.c)
4230  test_id(e2.a, e2.b, e2.c)
```

```
140719763031728 140719763031728 140719763031728
140719763031728 140719763031728 140719763031728
140719763031728 140719763031728 140719763031728
```

```
4233  e1 = Empty()
4234
4235  e1.a = [1]
4236  e1.b = [1]
4237  e1.c = [1]
4238
4239  e2 = Empty()
4240
4241  e2.a = [1]
4242  e2.b = [1]
4243  e2.c = [1]
4244
4245  a, b, c = [1], [1], [1]
4246  test_id(a, b, c)
4247  test_id(e1.a, e1.b, e1.c)
4248  test_id(e2.a, e2.b, e2.c)
```

```
1655554537280 1655554536448 1655554545792
1655553194624 1655553197376 1655554545664
1655554545600 1655554545536 1655554544960
```

```
4250  e3 = Empty()
4251  test_id(e3.a, e3.b, e3.c)
```

```
Traceback (most recent call last):
  File "C:\Users\popeC\OneDrive\CS124计算导论\2021 秋季\lecture notes\course_code.py", line 4251, in <module>
    test_id(e3.a, e3.b, e3.c)
AttributeError: 'Empty' object has no attribute 'a'
```

- 对象内部的属性和外部的属性互不干扰
- 对象之间的属性也互不干扰
- Line 4215-4230，line 4233-4248: 1是不可修改对象，[1]是可修改对象，系统为了节约内存，有时候会共用不可修改对象。可修改对象一定不会共用
- e1, e2 都通过“.”添加了内部的变量a, b, c. e3没有添加，所以e3报错

如果我们需要所有Empty的对象都自动具有a, b, c三个属性呢？？

Class: Challenge

1. 如何保证同一个类所有的对象都具有相同的属性和方法
   - class Dog：狗都有颜色，一双眼睛、四条腿，一条尾巴
   - 每个狗都是单独的属性(数据)和方法(函数): 空间消耗
2. 如何共享程序: 方法(函数)调用
   - red_dog, white_dog, red_dog.bite(), white_dog.bite()
   - 调用bite()时，知道是white_dog还是red_dog
   - 不同的对象互相不干扰，独立运行
   - 如果bite()含有参数，知道是哪个参数

- __init__(): 定义对象时， 自动运行
- self: 类内部用来保存额外的信息，区分不同对象
  - Line 26, self = black_dog, 自动运行 __init__()
  - Line 27, self = white_dog, 自动运行 __init__()
  - Line 29, Line 30, 运行时可以正常区分开 white_dog 和 black_dog
    - black_dog.color
    - white_dog.color

```
18  class Dog:
19      def __init__(self, color):
20          self.color = color
21
22      def bite(self):
23          print(f"{self.color} dog bites you.")
24
25
26  black_dog = Dog('black')
27  white_dog = Dog('white')
28
29  black_dog.bite()
30  white_dog.bite()
```

```
black dog bites you.
white dog bites you.
```

权衡：引入一个新的机制往往也带来新的问题
1. 构造函数__init__(): 保证统一性
2. self，用来区分不同对象

Class design: __init__(self) 构造函数

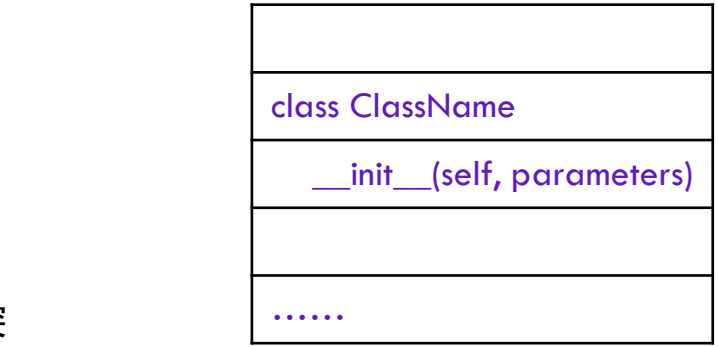
- def __init__(self): “__init__(self)” is a reserved method in python classes.
- All classes contain a __init__()
- This method is called automatically (自动运行) when an object is created from the class and it allow the class to initialize the attributes of a class
- It is known as a constructor（构造函数） in object oriented concepts.
- Use the __init__() function to assign values to object properties, or other operations that are necessary to do when the object is being created

```
class ClassName
    __init__(self, parameters)
.....
```

__init__(): 以**两个连续下划线** “_ _”开始和结束
init：initialization的缩写
Python中以__xxx__()形式的函数都具有特殊意义

Class: self

- self: myself, yourself, themselves ……
- The first parameter in methods __init__(self) must be self
- When define an object, self will point to that object
  - white_dog = Dog() # self = white_dog
  - black_dog = Dog() # self = red_dog
- __init__()保证统一性，self保证唯一性, 不和其它对象冲突
- 类的定义在程序里面也只是一段代码，只有具体定义一个对象时，才会使用。Self保证了程序可以区分开是哪个对象 (white_dog or red_dog)
- self不是keyword，可以用其它单词(this)代替它，但习惯上都用self.
- 第一个参数默认是self, __init__()没有参数会报错
- 定义对象的时候，不用用户传递self，系统会自己传递 (参数数量会少一个). 调用没有self的函数会报错




```
3944   class Empty:
3945       def __init__():
3946           pass
3947
3948   e = Empty()
```

```
Traceback (most recent call last):
  File "C:\Users\popeC\OneDrive\CS124计算导论\2021 秋季\lecture notes\course_code.py", line 3948, in <module>
    e = Empty()
TypeError: __init__() takes 0 positional arguments but 1 was given
```

定义: x = ClassName()
隐含: self = x

__init__() and self

```python
class TestInit:
    def __init__(self):
        print("__init__() is called automatically")
        print("self id: {}".format(id(self)))
        print("__init__() exit")


test1 = TestInit()
print(id(test1))

test2 = TestInit()
print(id(test2))
```

```
__init__() is called automatically
self id: 1655995151504
__init__() exit
1655995151504
__init__() is called automatically
self id: 1655995151024
__init__() exit
1655995151024
```

1. __init__()被在对象创建时，被自动调用
2. 定义: x = ClassName()
   隐含: self = x

对象和对象之间会共享函数，所以self保证了函数运行的互不干扰

self在对象调用函数时，不用传递。已经默认知道了

Member Attributes

```
3951  class Point:
3952      def __init__(self, x, y):
3953          self.x, self.y = x, y
3954
3955
3956
3957  def print_point_id(pt):
3958      print(id(pt), id(pt.x), id(pt.y))
3959
3960  def print_point(pt):
3961      print(pt.x, pt.y)
3962
3963
3964
3965  pt1 = Point(3, 4)
3966  pt2 = Point(3, 4)
3967  print_point(pt1)
3968  print_point(pt2)
3969  print_point_id(pt1)
3970  print_point_id(pt2)
3971
3972  print(pt1.x*pt2.y - pt1.y*pt2.x, pt1.x + pt2.y)
3973  pt1.x = -1
3974  pt2.y = "Hello"
3975
3976  print_point(pt1)
3977  print_point(pt2)
```

```
3 4
3 4
2119656160464 140710888465256 140710888465288
2119656169744 140710888465256 140710888465288
0 7
-1 4
3 Hello
```

```
3980  pt1 = Point([1], [2])
3981  pt2 = Point([1], [2])
3982  print_point_id(pt1)
3983  print_point_id(pt2)
```

3953: "=" 定义变量
3965: self = pt1
3966: self = pt2

```
2459846596352 2459845327296 2459845329792
2459846288528 2459846680384 2459846680320
```

- x, y是参数，self.x和self.y 表示将定义的对象的x 和y
  - Line 3965, 3966只有两个参数
- Line 3965—3970, pt1 and pt2 have the same position. Their address are different
- Line 3972—3977, you may use the attributes like general variables
- Line 3952 Vs. Line 3965, __init__ has 3 parameters. When initialize an object, self is not needed
- __init__ will be called when the object is created

Class Point: default parameters

```
3985  class Point:
3986      def __init__(self, x=0, y=0):
3987          self.x, self.y = x, y
3988
3989  pt1 = Point(3, 4)
3990  print(pt1.x, pt1.y)
3991
3992  pt2 = Point()
3993  print(pt2.x, pt2.y)
```

```
3 4
0 0
```

Class Dog

```python
class Dog:
    def __init__(self, name, color, age, weight):
        self.name = name
        self.color = color
        self.age = age
        self.weight = weight


def print_dog(dog):
    print(
        f"Name: {dog.name}, color: {dog.color}, age: {dog.age}, weight: {dog.weight}."
    )


dog = Dog("Fox", "Red", 3, 4)
print_dog(dog)

dog.age += 1
dog.weight /= 2
print_dog(dog)


def train_dog(dog):
    dog.name += " 1st"


train_dog(dog)
print_dog(dog)
```


```
Name: Fox, color: Red, age: 3, weight: 4.
Name: Fox, color: Red, age: 4, weight: 2.0.
Name: Fox 1st, color: Red, age: 4, weight: 2.0.
```

Member functions

- Member functions: the same with the general function definition, except that the first parameter should be self
- When called member functions, self is omitted

```python
class Dog:
    def __init__(self, name, color, age, weight):
        self.name = name
        self.color = color
        self.age = age
        self.weight = weight

    def set_name(self, name):
        self.name = name

    def set_color(self, color):
        self.color = color

    def set_age(self, age):
        self.age = age

    def set_weight(self, weight):
        self.weight = weight

    def print_dog(self):
        print(
            f"Name: {self.name}, color: {self.color}, age: {self.age}, weight: {self.weight}."
        )
```

```python
dg = Dog("Fox", "Red", 3, 4)
dg.print_dog()

dg.set_name("Tiger")
dg.set_color("White")
dg.set_age(10)
dg.set_weight(23)
dg.print_dog()

dg.name = "Kitty"
dg.color = "Pink"
dg.age = 2
dg.weight = 4
dg.print_dog()
```

```
Name: Fox, color: Red, age: 3, weight: 4.
Name: Tiger, color: White, age: 10, weight: 23.
Name: Kitty, color: Pink, age: 2, weight: 4.
```

不建议10-14这种写法

Member functions: no self

- Member functions without self cannot be called by an object, but called inside the class implementation via: ClassName.func()

```
4284 class Test:
4285     def __init__(self, x=0):
4286         self.x = x
4287
4288     def add(u,v):
4289         return u+v
4290
4291     def inc(self):
4292         self.x = self.add(self.x, 1)
4293
4294     def print(self):
4295         print(self.x)
4296
4297
4298
4299 test = Test(111)
4300 test.print()
4301
4302 test.inc()
4303 test.print()
```

Line 4288, u=self

```
111
Traceback (most recent call last):
  File "C:\Users\popeC\OneDrive\CS124计算导论\2021 秋季\lecture notes\course_code.py", line 4302, in <module>
    test.inc()
  File "C:\Users\popeC\OneDrive\CS124计算导论\2021 秋季\lecture notes\course_code.py", line 4292, in inc
    self.x = self.add(self.x, 1)
TypeError: add() takes 2 positional arguments but 3 were given
```

```
4262 class Test:
4263     def __init__(self, x=0):
4264         self.x = x
4265
4266     def add(u,v):
4267         return u+v
4268
4269     def inc(self):
4270         self.x = Test.add(self.x, 1)
4271
4272     def print(self):
4273         print(self.x)
4274
4275
4276
4277 test = Test(111)
4278 test.print()
4279
4280 test.inc()
4281 test.print()
4282
4283 print(Test.add(1, 1))
4284 print(test.add(3, 4))
```

```
111
112
2
```

Line 4284, error: add is not a function of self

class Triangle, self

```python
class Triangle:
    def __init__(self, a=0, b=0, c=0):
        self.a, self.b, self.c = a, b, c
```

```python
tr1 = Triangle(1, 1, 1)
tr2 = Triangle()
print(f"Tr1: {tr1.a}, {tr1.b}, {tr1.c}")
print(f"Tr2: {tr2.a}, {tr2.b}, {tr2.c}")
tr1.a, tr1.b, tr1.c = 4, 3, 5
tr2.a, tr2.b, tr2.c = 4, 3, 3
print(f"Tr1: {tr1.a}, {tr1.b}, {tr1.c}")
print(f"Tr2: {tr2.a}, {tr2.b}, {tr2.c}")
```

```
Tr1: 1, 1, 1
Tr2: 0, 0, 0
Tr1: 4, 3, 5
Tr2: 4, 3, 3
```

- tr1, tr2是两个独立的对象，虽然属于同一个类
- Tr1, tr2的属性和方法都可以直接使用，和普通变量一样
- . 运算符：表明作用范围

同一个类的不同对象
- 属性、方法独立，互相不干扰

__init__: 表明所有的Triangle对象都有a, b, c三个属性，并且默认为0

Q: 为什么有个self？为什么__init__()有四个参数，而tr1, tr2定义时只传递了三个参数

- 不同对象的属性，保存在不同的内存位置： tr1.a, tr2.a
- 为了节省内存空间。同一类的不同对象，共享方法（函数）的空间。执行的时候，都是跳转到同一段代码。问题：严重干扰，如何区分a是属于tr1还是tr2
- self是一个隐藏参数，当定义对象时，self被自动指向该对象
- tr1 = Triangle(1,1,1) # 系统默认 self=tr1
- tr2 = Triangle() #系统默认 self=tr2
- 类的方法调用的时候，self会自动传递过去
- 没有self的函数无法使用，使用会报错：参数数量不匹配
- 在类中使用属性和方法时，前面必须加self: self.xxx, self.xxx()
- self.x 和x，self.f()和f()是不同的方法和属性

class Triangle

```python
class Triangle:
    def __init__(self, a=0, b=0, c=0):
        self.a, self.b, self.c = a, b, c

    def is_valid(self):
        return (
            self.a > 0
            and self.b > 0
            and self.c > 0
            and self.a + self.b > self.c
            and self.b + self.c > self.a
            and self.c + self.a > self.b
        )

    def set_edges(self, a, b, c):
        self.a, self.b, self.c = a, b, c

    def area(self):
        q = (self.a + self.b + self.c) / 2
        return (q * (q - self.a) * (q - self.b) * (q - self.c)) ** 0.5

    def print(self):
        print(f"Triangle: {self.a}, {self.b}, {self.c}")

    def print_area(self):
        print(f"Area: {self.area():.3f}")  # area()会报错，必须加self
```

```python
t1 = Triangle()
t1.set_edges(3, 4, 5)

if t1.is_valid():
    t1.print()
    t1.print_area()

```

```
Triangle: 3, 4, 5
Area: 6.000
```

1. 类的成员函数和普通函数的定义一样，除了第一个参数是self
2. 类方法的执行规律和普通函数一样， 除了不需要传递self进去
3. 函数内使用成员属性，必须加self.

使用类自己的函数或者属性时，必须加self

Class HappyNumber

A happy number is a number defined by the following process:

1. Starting with any positive integer, replace the number by the sum of the squares of its digits.
2. Repeat the process until the number equals 1 (where it will stay), or it loops endlessly in a cycle which does not include 1.
3. Those numbers for which this process ends in 1 are happy.

```
4032  class HappyNumber:
4033      def __init__(self, number):
4034          self.number = number
4035
4036      def sum_of_digit_squares(self):
4037          return sum([int(x)**2 for x in str(self.number)]) # self is needed
4038
4039      def print_HN(self):
4040          print(self.number)
4041
4042      def is_happy(self):
4043          if self.number == 1:
4044              return True
4045
4046
4047          loop = {self.number}
4048          while loop:
4049              nn = self.sum_of_digit_squares()
4050              if nn == 1:
4051                  print(loop)
4052                  return True
4053
4054              if nn in loop:
4055                  print(loop)
4056                  return False
4057
4058              loop.add(nn)
4059              self.number = nn
4060
4061
```

```
4062  hn = HappyNumber(1234567890987654321)
4063  hn.print_HN()
4064  print(hn.sum_of_digit_squares())
4065  print(hn.is_happy())
4066
4067  hn = HappyNumber(19)
4068  hn.print_HN()
4069  print(hn.sum_of_digit_squares())
4070  print(hn.is_happy())
4071
4072  hn = HappyNumber(2)
4073  hn.print_HN()
4074  print(hn.sum_of_digit_squares())
4075  print(hn.is_happy())
```

```
1234567890987654321
570
{65, 58, 37, 4, 74, 42, 16, 1234567890987654321, 145, 20, 89, 570, 61}
False
19
82
{100, 82, 19, 68}
True
2
4
{2, 4, 37, 42, 16, 145, 20, 89, 58}
False
```

Scope of functions and variables

```python
def area(a, b, c):
    q = (a + b + c) / 2
    return (q * (q - a) * (q - b) * (q - c)) ** 0.5


class Test:
    def __init__(self, a=0, b=0, c=0):
        self.a, self.b, self.c = a, b, c

    def area(self):
        return area(self.a, self.b, self.c)


print("test")
a, b, c = 3, 4, 5
print(area(a, b, c))
test = Test(3, 3, 3)
print(test.area())
```

```
test
6.0
3.897114317029974
```

- area()和xxx.area()是两个函数
- 类里面的函数要加self
- 不同作用范围的两个物体是隔离的

Without __init__

```python
class Test1:
    def area(self):
        return area(self.a, self.b, self.c)

    def print(self):
        print(f"Triangle: {self.a}, {self.b}, {self.c}")


t1 = Test1()
t1.a, t1.b, t1.c = 1.1, 2.1, 1.1
t1.print()
print(t1.area())

t2 = Test1()
t2.print()
print(t2.area())
```

没有__init__()函数

- 类Test1的对象没有共同的属性
- Python自动推导变量的类型
- t1有自己的的属性a, b, c
- t2没有属性a, b, c

```
Triangle: 1.1, 2.1, 1.1
0.34426552252585607
Traceback (most recent call last):
  File "/Users/fancheng/OneDrive/CS124计算导论/2020 秋季/lecture notes/course_code.py", line 769, in <module>
    t2.print()
  File "/Users/fancheng/OneDrive/CS124计算导论/2020 秋季/lecture notes/course_code.py", line 761, in print
    print("Triangle: {}, {}, {}".format(self.a,self.b, self.c))
AttributeError: 'Test1' object has no attribute 'a'
```

Python classes/objects

- Python is an object oriented programming language
- Almost everything in Python is an object, with its properties and methods
- A Class is like an object constructor, or a "blueprint" for creating objects
- Procedural programming paradigm: write functions that operate data
- Python is an object-oriented Programming (OOP) Language
- OOP has its root in 1960s. In 1980s, it became the main programming paradigm
    - Rapid growing software size and complexity
    - The focus is object that contains both data and functionality
    - Each object definition corresponds to some object or concept in the real world, and the functions that operate on that object correspond to the ways real-world objects interact
- Commonly known OO Languages: C++, Java, Python
    - Class and object
- For more details: https://en.wikipedia.org/wiki/Object-oriented_programming

如果没有OO，所有东西都写在一起?

Exercise

- 实现Triangle类
- 实现vector类
  - 属性：一个n维的list
  - 方法
    - 长度length(self)
    - print_length()
    - 方向angle(self)
    - Print_angle()
    - 点积dot(self, other)
    - 伸缩scale(self, ratio)

Class Variables

- In general, instance variables are for data unique to each instance
- class variables are for attributes shared by all instances of the class

```python
class SomeClass:
    variable_1 = ["This is a class variable", 2018]
    variable_2 = 100    #this is also a class variable.

    def __init__(self, param1, param2):
        self.instance_var1 = param1
        #instance_var1 is a instance variable
        self.instance_var2 = param2
        #instance_var2 is a instance variable

obj1 = SomeClass("some thing", 18)
obj2 = SomeClass(28, 6)

print(obj1.variable_1, id(obj1.variable_1))
print(obj2.variable_1, id(obj2.variable_1))
print(obj1.instance_var1)
print(obj2.instance_var1)
```

```
['This is a class variable', 2018] 61361488
['This is a class variable', 2018] 61361488
some thing
28
```

类变量 Vs. 对象变量
所有人都有自己的姓名(instance)，但只有一个国籍(class)

Class Methods

- Class methods are methods shared by all instances of the class.
- Without even instantiating an object, we can access class methods

```
class SomeClassA:

    def create_arr(self):# An instance method
        self.arr = []

    def insert_to_arr(self, value):  #An instance method
        self.arr.append(value)

    @classmethod
    def class_method(cls):
        print("the class method was called")

SomeClassA.class_method()
```

```
the class method was called
```

```
1  class Example:
2      cnt = 0
3
4      @classmethod
5      def class_method(cls):
6          cls.cnt += 1
7          print("Test", cls.cnt)
8
9
10 Example.class_method()
11 Example.class_method()
12 Example.class_method()
```

```
Test 1
Test 2
Test 3
```

Class Variable and Class method

- Instance variables/methods are owned by the instance itself. No interference!
  - You must define an instance first to use them
- Class variables/methods are shared by all instances of the class
  - To use a class variables/methods, you don't need to define an instance first

```python
class ClassVariable:
    number = 0

    def __init__(self):
        print("init")
        ClassVariable.number += 1 # don't use self.
        self.x = 1

    @classmethod
    def print_num(cls):
        print(cls.number)

print(ClassVariable.number)
ClassVariable.print_num()

v1 = ClassVariable()
v2 = ClassVariable()
print(v1.number, v2.number)

print(ClassVariable.number)
ClassVariable.print_num()
```

```
0
0
init
init
2 2
2
2
```

类对象/函数实例化之前就可以使用
Classname.xxx

类属性/方法：类的所有对象共享同一份
- 不用实例化对象

实例属性/方法：各个对象各有一份

Private Methods _, __

- "Private" instance variables that cannot be accessed except from inside an object don't exist in Python
- However, there is a convention that is followed by most Python code:
  - a name prefixed with an underscore (e.g. _spam) should be treated as a non-public part of the API (whether it is a function, a method or a data member). It should be considered an implementation detail and subject to change without notice
  - To define a private method prefix the member name with the double underscore "__"

The double underscore "__" does not mean a "private variable". You use it to define variables which are "class local" and which can not be easily overridden by subclasses. It mangles the variables name.

For example:

```
class A(object):
  def __init__(self):
    self.__foobar = None # Will be automatically mangled to self._A__foobar

class B(A):
  def __init__(self):
    self.__foobar = 1 # Will be automatically mangled to self._B__foobar
```

```
class Mapping:
    def __init__(self, iterable):
        self.items_list = []
        self.__update(iterable)

    def update(self, iterable):
        for item in iterable:
            self.items_list.append(item)

    __update = update   # private copy of original update() method

class MappingSubclass(Mapping):

    def update(self, keys, values):
        # provides new signature for update()
        # but does not break __init__()
        for item in zip(keys, values):
            self.items_list.append(item)
```

https://stackoverflow.com/questions/1641219/does-python-have-private-variables-in-classes

non-public：不建议（禁止）使用

Inheritance

Evolution and Inheritance (继承)

软件的生命周期：开发、部署、维护

- Inheritance: Classes can inherit from other classes
- A class can inherit attributes and behaviour methods from another class, called the superclass（超类）. Superclasses are sometimes called ancestors as well
- A class which inherits from a superclass is called a subclass（子类）, also called heir class or child class
- Evolution: Subclass may have their own attributes and methods
- Inheritance and Evolution — polymorphism

Inheritance Syntax

- The syntax for a subclass definition looks like this:

  class DerivedClassName(BaseClassName1, BaseClassName2, …):

```python
class Person:
    def __init__(self, name, job=None, pay=0):
        print("Init: Person " + name)
        self.name = name
        self.job = job
        self.pay = pay

    def get_last_name(self):
        return self.name.split()[-1]

    def give_raise(self, percent):
        self.pay = int(self.pay * (1 + percent))

    def __repr__(self):  # added method
        return f"[Person: {self.name}, {self.pay}]"  # string to print
```

```python
class Manager(Person):
    pass


roger = Manager("Roger Chen", "Manager", 100000)
print(roger.name, roger.job, roger.pay)
print(roger.get_last_name())
roger.give_raise(percent=0.1)
print(roger)
```

```
Init: Person Roger Chen
Roger Chen Manager 100000
Chen
[Person: Roger Chen, 110000]
```

Person类
属性：name, job, pay
方法：get_last_name(), give_raise(), __repr__()
__str__() for print()

Manager类
继承Person: 具有Person的所有属性和方法
自己的方法和属性

继承：拥有超类所有的属性和方法

Functions with the same name

- Python中，函数必须在使用前定义
- Python中，两个同名函数， 无论参数是否相同，后面的函数会覆盖前面的函数。即，python中~~没有重载~~，只有重写

```python
def func1(name, age):
    print(f"name = {name}, age = {age}")


func1("Hello", 28)


def func1(name, age, country):
    print(f"name = {name}, age = {age}, country = {country}")


# func1("Hello", 28) # error
func1("Hello", 28, "CN")
```

```
name = Hello, age = 28
name = Hello, age = 28, country = CN
```

Overriding (重写)

- Method overriding is an object-oriented programming feature that allows a subclass to provide a different implementation of a method that is already defined by its superclass or by one of its superclasses
- The implementation in the subclass overrides the implementation of the superclass by providing a method with the same name as the method of the parent class

```
1  class Manager(Person):
2      def give_raise(self, percent, bonus=0.1):
3          Person.give_raise(self, percent + bonus)
4          # self.pay = int(self.pay * (1+percent)) # Bad: cut and paste
5          # double maintance
6
7      def __repr__(self):  # added method
8          return "[Manager: {:s} {:d}]".format(self.name, self.pay)  # string to print
9
10
11 roger = Manager("Roger Chen", "Manager", 100000)
12 print(roger)
13 roger.give_raise(percent=0.1)
14 print(roger)
```

```
Init: Person Roger Chen
[Manager: Roger Chen 100000]
[Manager: Roger Chen 120000]
```

super().giveRaise()可以起到和Person.giveRaise()相同的功能，但是不推荐用super()

- Person.giveRaise()比super更清晰
- 一个类可以继承多个类，super()有歧义

重写：子类覆盖超类
“年轻人有自己的想法”

Overriding __init__()

- To add new members to the attributes.

```
1  class Manager(Person):
2      def __init__(self, name, pay, id):
3          print("Init: Manager " + name)
4          Person.__init__(self, name, "mgr", pay)
5          self.id = id
6
7      def give_raise(self, percent, bonus=0.1):
8          Person.give_raise(self, percent + bonus)
9
10     def __repr__(self):
11         return f"[Manager: {self.name}, {self.id} {self.pay}]"
12
13
14 roger = Manager(name="Roger Chen", pay=100000, id="CN007")
15 print(roger)
16 roger.give_raise(percent=0.1)
17 print(roger)
```

```
Init: Manager Roger Chen
Init: Person Roger Chen
[Manager: Roger Chen, CN007 100000]
[Manager: Roger Chen, CN007 120000]
```

注意：删除Person.__init__(self, name, 'mgr', pay)则manager将没有name, job, pay.
必须先调用父类的__init__()

issubclass() and isinstance()

- Python (from version 3.x), object is root of all classes.
- Python provides a function issubclass() that directly tells us if a class is subclass of another class. isinstance() will tell us if a variable is an object of a class

```
class Base(object):
    pass   # Empty Class

class Derived(Base):
    pass   # Empty Class

# Driver Code
print(issubclass(Derived, Base))
print(issubclass(Base, Derived))

d = Derived()
b = Base()

# b is not an instance of Derived
print(isinstance(b, Derived))

# But d is an instance of Base
print(isinstance(d, Base))
```

```
True
False
False
True
```

== VS. is

- The == operator tests value equivalence. Python performs an equivalence test, comparing all nested objects recursively
- The is operator tests object identity (身份证). Python tests whether the two are really the same object (i.e., live at the same address in memory)

```
L1 = [1, ("a", 3)]  # Same value, unique objects
L2 = [1, ("a", 3)]
print(L1 == L2, L1 is L2)
```

```
True False
```

```
S1 = "spam"
S2 = "spam"
print(S1 == S2, S1 is S2)

S1 = "xyz a longer string+_)(*&^%$#@!"
S2 = "xyz a longer string+_)(*&^%$#@!"
print(S1 == S2, S1 is S2)
```

```
True True
True True
```

is 依赖于系统

```
x = 100
y = 100
print(x == y, x is y)

x = 2**64
y = 2**64
print(x == y, x is y)

x = 2**128
y = 2**128
print(x == y, x is y)
```

```
True True
True True
True False
```

```
x = "Hello"
y = "SJTU"
z = x + y
w = "HelloSJTU"

print(z is w, z == w, z)
```

```
False True HelloSJTU
```

```
import sys

x = str(2**1)
y = str(2**1)
print(x == y, x is y)

x = sys.intern(x)
y = sys.intern(y)
print(x is y, x == y, x)
```

```
True False
True True 2
```

Introspection Tools

- Special attributes and functions that give us access to some of the internals of objects' implementations.
- The built-in instance.__class__ attribute provides a link from an instance to the class from which it was created. Classes in turn have a __name__, and a __bases__ sequence that provides access to superclasses.
- The built-in object.__dict__ attribute provides a dictionary with one key/value pair for every attribute attached to a namespace object (including modules, classes, and instances). (dir() for functions)

```
1  roger = Manager(name="Roger Chen", pay=100000, id="CN007")
2  print(roger)
3
4  print(roger.__class__)
5  print(roger.__class__.__name__)
6  print(list(roger.__dict__.keys()))
7
8  for key in roger.__dict__:
9      print(key, "=>", roger.__dict__[key])
10
11 for key in roger.__dict__:
12     print(roger, "=>", getattr(roger, key))
```

```
1  Init: Manager Roger Chen
2  Init: Person Roger Chen
3  [Manager: Roger Chen, CN007 100000]
4  <class '__main__.Manager'>
5  Manager
6  ['name', 'job', 'pay', 'id']
7  name => Roger Chen
8  job => mgr
9  pay => 100000
10 id => CN007
11 [Manager: Roger Chen, CN007 100000] => Roger Chen
12 [Manager: Roger Chen, CN007 100000] => mgr
13 [Manager: Roger Chen, CN007 100000] => 100000
14 [Manager: Roger Chen, CN007 100000] => CN007
```

```
1  x = 100
2  print(dir(x))
3
4  x = "SJTU"
5  print(dir(x))
```

object class

- Python (from version 3.x), object is root of all classes.
- 所有的类都继承了object类
  - 类定义的头 class ClassName, 其实是 class ClassName (object):
- 所有的对象都可以调用 len(), print(), id(), type()
- object.__len__(self)
- object.__str__(self): Called by str(object) and the built-in functions format() and print() to compute the “informal” or nicely printable string representation of an object. The return value must be a string object
- 如何让自定义的类能够使用基本的运算符： +,-,*,/, +=, >=, 等等
  - object.__lt__(self, other) # x<y lt: less than
  - object.__le__(self, other) # x<=y less equal
  - object.__eq__(self, other) # x==y equal
  - object.__ne__(self, other) # x!=y not equal
  - object.__gt__(self, other) # x>y greater than
  - object.__ge__(self, other) # x>=y greater equal
- These are the so-called “rich comparison” methods. The correspondence between operator symbols and method names is as follows:
  - x<y calls x.__lt__(y),
  - x<=y calls x.__le__(y),
  - x==y calls x.__eq__(y),
  - x!=y calls x.__ne__(y),
  - x>y calls x.__gt__(y),
  - x>=y calls x.__ge__(y)

1. https://docs.python.org/3/reference/datamodel.html

Triangle: overloading object

See P23

```python
def __len__(self):
    return self.a + self.b + self.c

def __str__(self):
    return ", ".join((str(self.a), str(self.b), str(self.c)))

def __lt__(self, other):  # x<y                    lt: large than
    return self.a < other.a and self.b < other.b and self.c < other.c

def __le__(self, other):  # x<=y                  less equal
    return self.a <= other.a and self.b <= other.b and self.c <= other.c

def __eq__(self, other):  # x==y                  equal
    return self.a == other.a and self.b <= other.b and self.c <= other.c

def __ne__(self, other):  # x!=y                    not equal
    return self.a != other.a or self.b != other.b or self.c != other.c

def __gt__(self, other):  # x>y                     greater than
    return self.a > other.a and self.b > other.b and self.c > other.c

def __ge__(self, other):  # x>=y                  greater equal
    return self.a >= other.a and self.b >= other.b and self.c >= other.c
```

```python
t = Triangle()
t.set_edges(3, 4, 5)
print(len(t))
print(t)

t1 = Triangle()
t1.set_edges(3, 4, 5)
print("<     <=   !=    ==    >    >=")
print(t < t1, t <= t1, t != t1, t == t1, t > t1, t >= t1)

t1 = Triangle()
t1.set_edges(4, 5, 6)
print("<     <=   !=    ==    >    >=")
print(t < t1, t <= t1, t != t1, t == t1, t > t1, t >= t1)
```

```
12
3, 4, 5
<     <=   !=    ==    >    >=
False True False True False True
<     <=   !=    ==    >    >=
True True True False False False
```

此处的大小关系是自定义的

[]: __getitem__, __setitem__

To access an element via its index

- __getitem__(self, index)
- __setitem__(self, index, value)

```
1  class vector:
2      def __init__(self, n):
3          self.data = [1] * n
4
5      def __getitem__(self, index):
6          return self.data[index]
7
8      def __setitem__(self, index, value):
9          self.data[index] = value
10
11
12 v = vector(100)
13 v.data = [i for i in range(100)]
14 print(v[67])
15 v[67] = -100
16 print(v[67])
```

```
1  67
2  -100
```

+, +=

```python
class Number:
    def __init__(self, start):
        self.data = start

    def __add__(self, other):
        return Number(self.data + other)

    def __radd__(self, other):
        return Number(self.data + other)

    def __iadd__(self, other):  # __iadd__ explicit: x += y
        self.data += other  # Usually returns self
        return self

    def __str__(self):
        return str(self.data)


x = Number(1)
print(x)
y = x + 1
z = -9 + x
x += 100
print(x, y, z)
```

```
1
101 2 -8
```

__add__: y = x + 1
__radd__: z = -9 + x
__iadd__: x += 100

repr(): __repr__(), __str__()

- __str__() inside class X will be used for print(x)
- __repr(object)__: Called by the repr() built-in function and by string conversions (reverse quotes) to compute the “official” string representation of an object
  - This is typically used for debugging, so it is important that the representation is information-rich and unambiguous.
  - In short, the goal of __repr__ is to be unambiguous and __str__ is to be readable.
- 如果没有定义__str__, 将自动使用__repr__
- My rule of thumb: “__repr__ is for developers, __str__ is for customers.”
  - __repr__比__str__包含更多的信息

```
1  s = "Hello world"
2  print(str(s), repr(s))
3
4  print(2.0 / 11.0, repr(2.0 / 11.0))
5
6  import datetime
7
8  today = datetime.datetime.now()
9
10 print(str(today))
11 print(repr(today))
```

```
Hello world 'Hello world'
0.18181818181818182 0.18181818181818182
2023-10-19 23:30:38.127104
datetime.datetime(2023, 10, 19, 23, 30, 38, 127104)
```

Repr：reproduce,包含更多信息

1. https://stackoverflow.com/questions/1436703/difference-between-str-and-repr
2. https://docs.python.org/3.8/reference/datamodel.html

All are Object

- Functions are also objects
  - dir(func)
  - func.__name__
  - func.__code__

```
def printer(str1):
    print("Hello, world")

print(dir(printer))
print(type(printer))
print(printer.__module__)
print(printer.__name__)
```

```
['__annotations__', '__call__', '__class__', '__closure__', '__code__', '__d
at__', '__ge__', '__get__', '__getattribute__', '__globals__', '__gt__', '__
, '__module__', '__name__', '__ne__', '__new__', '__qualname__', '__reduce__
classhook__']
<class 'function'>
__main__
printer
```

Duck Typing

"If it walks like a duck, and it quacks like a duck, then it must be a duck."

- Duck typing is a concept related to dynamic typing, where the type or the class of an object is less important than the methods it defines. When you use duck typing, you do not check types at all. Instead, you check for the presence of a given method or attribute
- Duck typing is a concept that says that the "type" of the object is a matter of concern only at runtime and you don't need to explicitly mention the type of the object before you perform any kind of operation on that object, unlike normal typing where the suitability of an object is determined by its type
- In Python, we have the concept of Dynamic typing i.e. we can mention the type of variable/object later. The idea is that you don't need a type in order to invoke an existing method on an object if a method is defined on it, you can invoke it

Polymorphism (多态)

- Polymorphism in Computer Science is the ability to present the same interface for differing underlying forms. We can have in some programming languages polymorphic functions or methods, for example. Polymorphic functions or methods can be applied to arguments of different types, and they can behave differently depending on the type of the arguments to which they are applied.

```
def f(x, y):
    print("values: ", x, y)

f(42, 43)
f(42, 43.7)
f(42.3, 43)
f(42.0, 43.9)
```

```
values:  42 43
values:  42 43.7
values:  42.3 43
values:  42.0 43.9
```

```
def printer(x):
    print(x)

bob = Person('Bob Smith')
sue = Person('Sue Jones', job='dev', pay=100000)
roger = Manager("Roger Chen", "manager", 100000)

printer(1)
printer("SJTU")
printer(list(range(4)))
printer(bob)
printer(sue)
printer(roger)
```

```
1
SJTU
[0, 1, 2, 3]
[Person: Bob Smith, 0]
[Person: Sue Jones, 100000]
[Manager: Roger Chen, 100000]
```

Object: len, print, type, id
__str__() for print()

多态：根据不同的参数类型，运行时自动调用相应的方法，不用重复编写相同的代码

Machinery of attribute inheritance

```
4112  class Ape:
4113      pass
4114
4115  class Human(Ape):
4116      pass
4117
4118  class Chinese(Human):
4119      pass
4120
4121  class Greece(Human):
4122      pass
4123
4124  class Students(Greece, Chinese):
4125      pass
4126
4127  stt = Students()
4128  print(type(stt))
```

```
<class '__main__.Students'>
```

Classes support factoring and customization of code better than any other language tool we've seen so far.

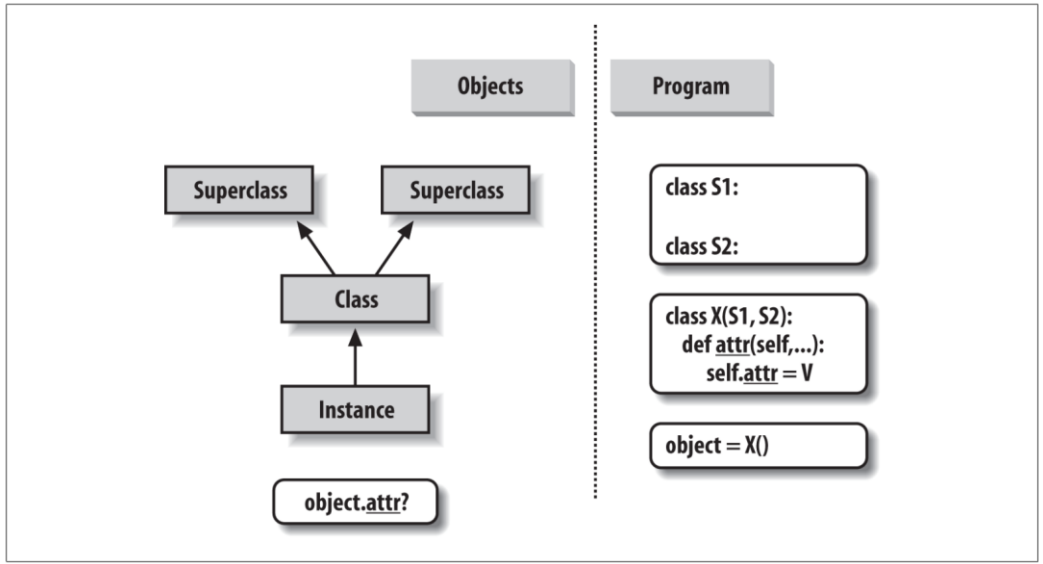
- they allow us to minimize code redundancy (and so reduce maintenance costs) by factoring operations into a single, shared implementation;
- they allow us to program by customizing what already exists, rather than changing it in place or starting from scratch.

Machinery of attribute inheritance

- Superclasses are listed in parentheses in a class header.
  - To make a class inherit attributes from another class, just list the other class in parentheses in the new
- Classes inherit attributes from their superclasses.
  - Just as instances inherit the attribute names defined in their classes, classes inherit all of the attribute names defined in their superclasses
- Instances inherit attributes from all accessible classes.
  - Each instance gets names from the class it's generated from, as well as all of that class's superclasses.
  - When looking for a name, Python checks the instance, then its class, then all superclasses.
- Each object.attribute reference invokes a new, independent search.
  - Python performs an independent search of the class tree for each attribute fetch expression.
- Logic changes are made by subclassing, not by changing superclasses.
  - By redefining superclass names in subclasses lower in the hierarchy (class tree), subclasses replace and thus customize inherited behavior.

Inheritance Tree

- Instance attributes are generated by assignments to self attributes in methods
- Class attributes are created by statements (assignments) in class statements
- Superclass links are made by listing classes in parentheses in a class statement header



The net result is a tree of attribute namespaces that leads from an instance, to the class it was generated from, to all the superclasses listed in the class header. Python searches upward in this tree, from instances to superclasses

Multiple inheritance 多重继承

- Sub class may inherit several sup class

```
class A:
    attrA = "A"

class B:
    attrB = "B"

class C(A, B):
    attrC = "C"

x = C()
print(x.attrA, x.attrB, x.attrC)
```

```
A B C
```

Chapter 31: Designing with Classes
P. 956 Multiple Inheritance: “Mix-in” Classes

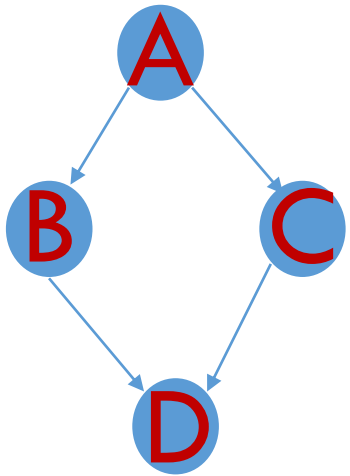
- Questions:
  - super()的困扰

```
class A1:
    attr = "A1"

class A2:
    attr = "A2"

class A3(A1, A2):
    attr = "A3"

x = A3()
print(x.attr)
```

```
class A:
    attr = "A"

class B(A):
    attr = "B"

class C(A):
    attr = "C"

class D(B, C):
    pass

x = D()
print(x.attr)
```

```
class A:
    attr = "A"

class B(A):
    attr = "B"

class C(A):
    attr = "C"

class D(C, B):
    pass

x = D()
print(x.attr)
```

```
A3
B
C
```



仅供了解
MRO
BFS

Example

- Triangle
- Vector
- Point2D
- nDPoint
- ComplexNumber
- Happy Number
- Isomorphic String

Reading

Learning Python: Part VI. Class and OOP

- []
- for
- ……
- Super() P. 1403
- 体会class的优点：代码界限清晰，互相不干扰；接口统一，配合方便
- 除了实现上层类的接口，一般不要用继承

不需要死记，要用的时候再翻看手册

Module

The big picture

- Python module:
  - the highest-level program organization unit, which packages program code and data for reuse, and provides self contained namespaces that minimize variable name clashes across your programs.
- In concrete terms, modules typically correspond to Python program files:
  - Each file is a module, and modules import other modules to use the names they define.
- Modules are processed with two statements and one important function:
  - import: Lets a client (importer) fetch a module as a whole
  - from: Allows clients to fetch particular names from a module
  - imp.reload (reload in 2.X): Provides a way to reload a module's code without stopping Python)
- In short, modules provide an easy way to organize components into a system by serving as self-contained packages of variables known as namespaces (名空间).

一个单独的文件就是一个module

import

- import math
  print(math.sin(math.pi/3))
- 语法：import moduleName
  - 由于moduleName就是文件名，所以python的文件名必须符合python的变量定义规则
- 在文件b.py中定义函数spam(). 在文件a.py中import b并且调用b.spam(). 运行a.py
- 在文件my_geometry.py中定义Triangle类. 在另一文件中import并运行

```
def spam(text): # File b.py
    print(text, 'spam')
```

```
import b # File a.py

b.spam("Test")
```

```
Test spam
```

```
import my_geometry

tt = my_geometry.Triangle(6,7,8)
tt.print()
```

```
my_geometry.Triangle: 6, 7, 8
```

b.py和my_geometry.py都是python文件
都可以作为module导入(import)

Top-level file (顶层文件)

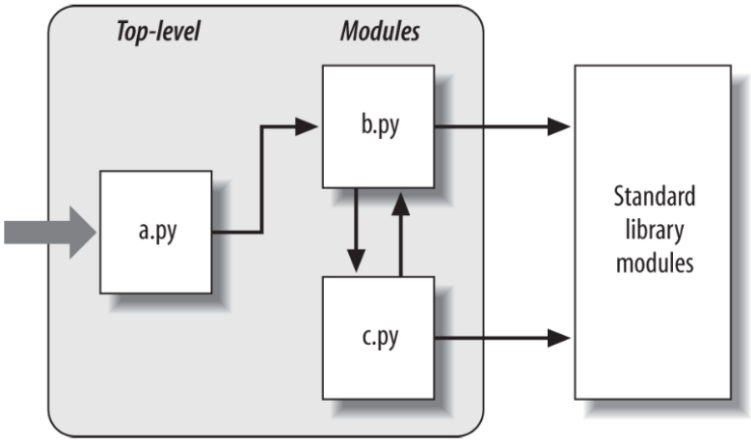
- 模块之间import，其执行顺序和函数调用类似
- At a base level, a Python program consists of text files containing Python statements, with
  a) one main top-level file (当前运行的文件)
  b) zero or more supplemental files known as modules (被import的文件)
- The top-level (a.k.a. script) file contains the main flow of control of your program—this is the file you run to launch your application.
- The module files are libraries of tools used to collect components used by the top-level file, and possibly elsewhere. Top-level files use tools defined in module files, and modules use tools defined in other modules.
- A file imports a module to gain access to the tools it defines, which are known as its attributes—variable names attached to objects such as functions. Ultimately, we import modules and access their attributes to use their tools.

```
import math
math.pi
math.sin(x)
```

Top-level: 自己当家作主，调用其他模块
Imported module: 打工人

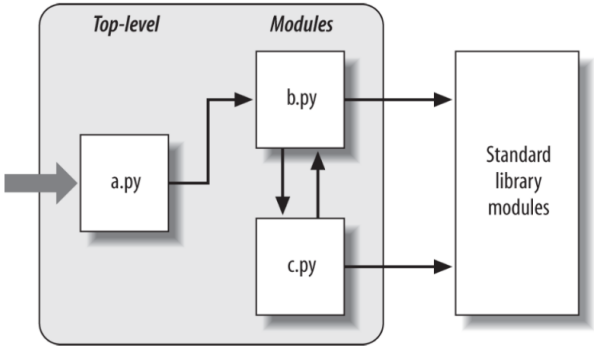
Python Program Architecture

- Program architecture in Python.
  - A program is a system of modules
  - one top-level (顶层) script file (launched to run the program)
  - and multiple module files (imported libraries of tools)
- Scripts and modules are both text files containing Python statements, though the statements in modules usually just create objects to be used later
- Python's standard library provides a collection of precoded modules



Python Program Architecture: import

- The file a.py is chosen to be the top-level file; it will be a simple text file of statements, which is executed from top to bottom when launched.
- The files b.py and c.py are modules; they are simple text files of statements as well, but they are not usually launched directly. Instead, as explained previously, modules are normally imported by other files that wish to use the tools the modules define
- The first of these, a Python import statement, gives the file a.py access to everything defined by top-level code in the file b.py.
- The code import b roughly means:
  - Load the file b.py (unless it's already loaded), and give me access to all its attributes through the name b.
- In fact, the module name used in an import statement serves two purposes: it identifies the external file to be loaded, but it also becomes a variable assigned to the loaded module.
- The code b.spam means:
  - Fetch the value of the name spam that lives within the object b.



import

Objects defined by a module are also created at runtime, as the import is executing:
import literally runs statements in the target file one at a time to create its contents.

```
1   print("Enter module1.py")  # module1.py
2
3   name = "SJTU"
4
5   def printer(x):
6       print(x)
7
8   print(name)
9   print("Exit module1.py")
```

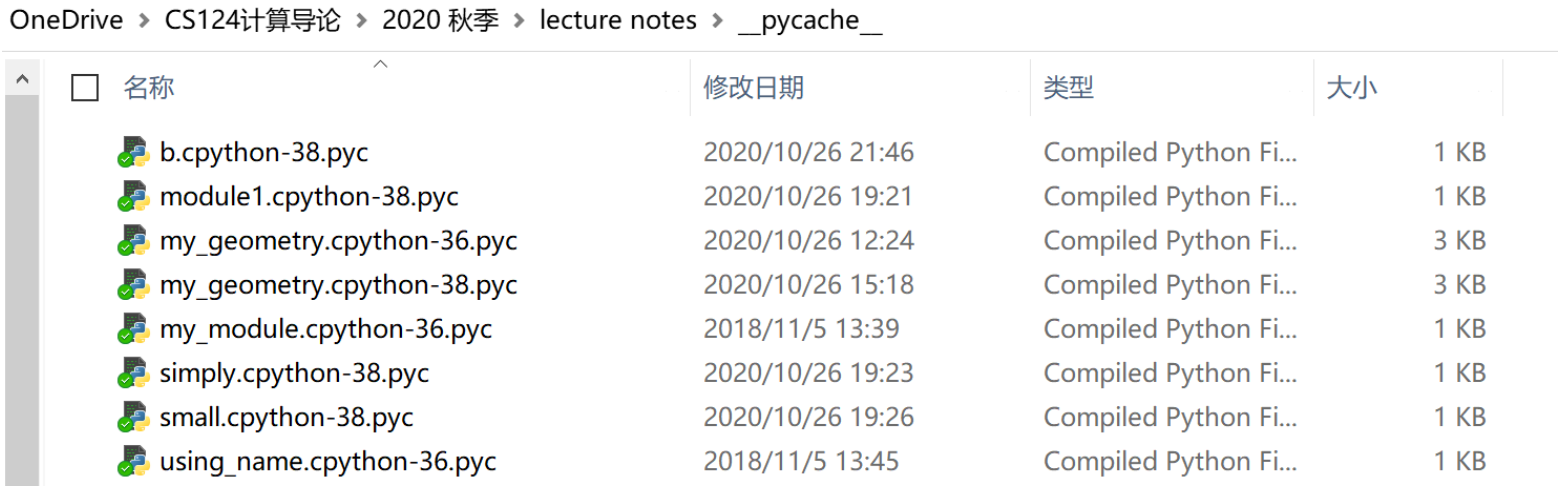
```
import module1
module1.printer([1, 2, 4])
```

```
Enter module1.py
SJTU
Exit module1.py
[1, 2, 4]
```

被import的文件的每一条语句都会被执行一遍

How import works

- They are really runtime operations that perform three distinct steps the first time a program imports a given file:
  1. Find the module's file.
  2. Compile it to byte code (if needed). Byte Code Files: __pycache__ in Python
  3. Run the module's code to build the objects it defines.



The module search path

import时寻找模块的步骤

1. The home directory of the program
2. PYTHONPATH directories (if set)
3. Standard library directories
4. The contents of any .pth files (if present)
5. The site-packages home of third-party extensions

import, from … import

- import moduel1
  module1.printer(‘Hello world!’)
- from moduel1 import printer
  printer('Hello world!’)
- from module1 import *
  printer('Hello world!’)

*：通配符，代表“所有，全部，任意”

```python
print("Enter module1.py")  # module1.py

name = "SJTU"

def printer(x):
    print(x)

print(name)
print("Exit module1.py")
```

```python
from module1 import printer

printer("test")
```

```
Enter module1.py
SJTU
Exit module1.py
test
```

vs code和Jupyter有差异，以vs code为准

from xxx import xxx的文件的每一条语句也都会被执行一遍

Import Happen Only Once

- Modules are loaded and run on the first import or from, and only the first.
  - This is on purpose—because importing is an expensive operation, by default Python does it just once per file, per process. Later import operations simply fetch the already loaded module object.

```
print('hello') # File simple.py
spam = 1
```

```
import simple
print(simple.spam)

simple.spam = 2

import simple
print(simple.spam)
```

```
hello
1
2
```

Import and from

- At least conceptually, a from statement like this one:

  ```
  from module import name1, name2 # Copy these two names out (only)
  ```

  is similar to this statement sequence:

  ```
  import module # Fetch the module object
  name1 = module.name1 # Copy names out by assignment
  name2 = module.name2
  del module # Get rid of the module name
  ```

- Like all assignments, the from statement creates new variables in the importer, which initially refer to objects of the same names in the imported file. Only the names are copied out, though, not the objects they reference, and not the name of the module itself. When we use the from * form of this statement (from module import *), the equivalence is the same, but all the top-level names in the module are copied over to the importing scope this way.

Reload modules

In a nutshell:

- Imports (via both import and from statements) load and run a module's code only the first time the module is imported in a process
- Later imports use the already loaded module object without reloading or rerunning the file's code
- The reload function forces an already loaded module's code to be reloaded and rerun
- Unlike import and from:
  - reload is a function in Python, not a statement.
  - reload is passed an existing module object, not a new name.
  - reload lives in a module in Python 3.X and must be imported itself.
- from imp import reload
  reload(moduleName)

as (别名)

- import modulename as name # And use name, not modulename

is equivalent to the following, which renames the module in the importer's scope only (it's still known by its original name to other files):

```
import modulename
name = modulename
del modulename # Don't keep original name
```

- After such an import, you can—and in fact must—use the name listed after the as to refer to the module. This works in a from statement, too, to assign a name imported from a file to a different name in the importer's scope; as before you get only the new name you provide, not its original:

```
from modulename import attrname as name
import reallylongmodulename as name # Use shorter nickname
name.func()
from module1 import utility as util1 # Can have only 1 "utility"
from module2 import utility as util2
util1()
util2()
```

import numpy as np

__future__

- Changes to the language that may potentially break existing code are usually introduced gradually in Python. They often initially appear as optional extensions, which are disabled by default. To turn on such extensions, use a special import statement of this form:

  from __future__ import featurename

- When used in a script, this statement must appear as the first executable statement in the file (possibly following a docstring or comment), because it enables special compilation of code on a per-module basis. It’s also possible to submit this statement at the interactive prompt to experiment with upcoming language changes; the feature will then be available for the remainder of the interactive session.
- 必须第一行
- 列子：在py2.0环境下使用py3.0的语法

__name__ and __main__

- Top-level: 自己当家作主，调用其他模块
- Imported module: 打工人
- Both import a file as a module and run it as a standalone program, is widely used in Python files.
- Each module has a built-in attribute called __name__, which Python creates and assigns automatically as follows:
  - If the file is being run as a top-level program file, __name__ is set to the string "__main__" when it starts.
  - If the file is being imported instead, __name__ is set to the module's name as known by its clients.
- The upshot is that a module can test its own __name__ to determine whether it's being run or imported.
- In effect, a module's __name__ variable serves as a usage mode flag, allowing its code to be leveraged as both an importable library and a top-level script.

Top-level Vs. Imported

```python
print("Enter module1.py")  # module1.py

name = "SJTU"

def printer(x):
    print(x)

print(name)
print("Exit module1.py")

if __name__ == "__main__":
    print("module1 main")
```

```
Enter module1.py
SJTU
Exit module1.py
module1 main
```

```python
import module1 # test.py
module1.printer([1, 2, 4])
```

```
Enter module1.py
SJTU
Exit module1.py
[1, 2, 4]
```

import

From Lec. 8

- Imports are always put at the top of the file, just after any module comments and docstrings, and before module globals and constants.
- Imports should be grouped in the following order:
  - Standard library imports.
  - Related third party imports.
  - Local application/library specific imports.
- You should put a blank line between each group of imports.
- Absolute imports are recommended, as they are usually more readable and tend to be better behaved
- Wildcard imports (from <module> import *) should be avoided

```
# Correct:
import os
import sys
```

```
# Wrong:
import sys, os
```

```
# Correct:
from subprocess import Popen, PIPE
```

Must Know Useful Module

Top 2

- Matplotlib : plot figures
- Numpy: numerical computing

Others

- Scipy: scientific computing
- Pandas: data analysis
- Scrapy: Crawler
- Beautiful soup: pulling data out of HTML and XML files
- Game
  - Pygame: pip3 install pygame
    To see if it works, run one of the included examples:
    python3 -m pygame.examples.aliens

Numpy

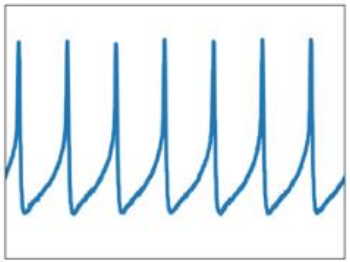
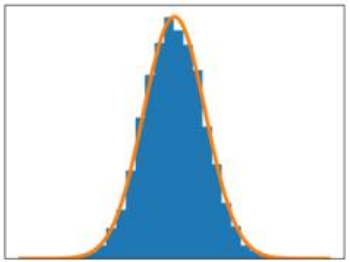
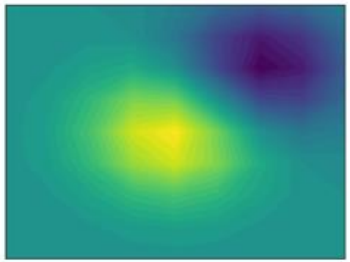
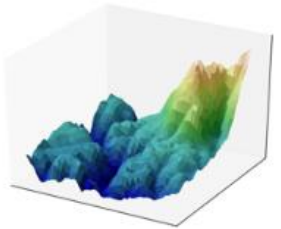
- NumPy is the fundamental package for scientific computing with Python.
- It contains among other things:
  - a powerful N-dimensional array object
  - sophisticated (broadcasting) functions
  - tools for integrating C/C++ and Fortran code
  - useful linear algebra, Fourier transform, and random number capabilities
- https://numpy.org/

```
>>> import numpy as np
>>> a = np.arange(15).reshape(3, 5)
>>> a
array([[ 0,  1,  2,  3,  4],
       [ 5,  6,  7,  8,  9],
       [10, 11, 12, 13, 14]])
>>> a.shape
(3, 5)
>>> a.ndim
2
>>> a.dtype.name
'int64'
>>> a.itemsize
8
>>> a.size
15
>>> type(a)
<type 'numpy.ndarray'>
>>> b = np.array([6, 7, 8])
>>> b
array([6, 7, 8])
>>> type(b)
<type 'numpy.ndarray'>
```

务必自学

Matplotlib

- Matplotlib is a Python 2D plotting library which produces publication quality figures in a variety of hardcopy formats and interactive environments across platforms. Matplotlib can be used in Python scripts, the Python and IPython shells, the Jupyter notebook, web application servers, and four graphical user interface toolkits.
- https://matplotlib.org/index.html

Scipy

- SciPy (pronounced “Sigh Pie”) is a Python-based ecosystem of open-source software for mathematics, science, and engineering. In particular, these are some of the core packages:

NumPy

Base N-dimensional array package

SciPy library

Fundamental library for scientific computing

Matplotlib

Comprehensive 2-D plotting

IP[y]: IPython

IPython

Enhanced interactive console

SymPy

Symbolic mathematics

pandas

Data structures & analysis

- https://www.scipy.org/

Pip: Module Management

- Package installer for Python: https://pypi.org/project/pip/
- pip is a de facto standard package-management system used to install and manage software packages written in Python
- Many packages can be found in the default source for packages and their dependencies — Python Package Index
- Most distributions of Python come with pip preinstalled
- xxx.whl: A WHL file is a package saved in the Wheel format, which is the standard built-package format used for Python distributions
- Quickstart: https://pip.pypa.io/en/stable/quickstart/
- Usage
  - pip install SomePackage
  - pip install --upgrade SomePackage
  - pip uninstall SomePackage

```
PS C:\Users\popeC> pip install beautifulsoup4
Collecting beautifulsoup4
  Downloading beautifulsoup4-4.9.3-py3-none-any.whl (115 kB)
     |████████████████████████████████| 115 kB 152 kB/s
Collecting soupsieve>1.2; python_version >= "3.0"
  Downloading soupsieve-2.0.1-py3-none-any.whl (32 kB)
Installing collected packages: soupsieve, beautifulsoup4
Successfully installed beautifulsoup4-4.9.3 soupsieve-2.0.1
```

```
from bs4 import BeautifulSoup
```

```
    import beautifulsoup
ModuleNotFoundError: No module named 'beautifulsoup'
```

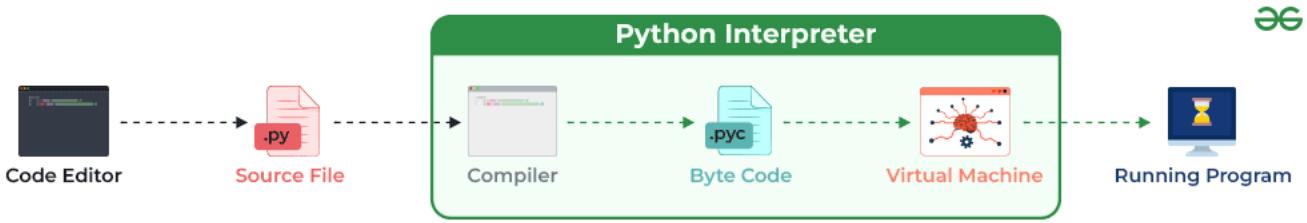
pip install xxx 网络安装
pip install xxx.whl 本地安装

Reading

- Learning Python: Part V. Modules and Packages

Interpreter

Python Interpreter



- Compiler — Translate the whole source code into assembly language, then execute assembly language
  - C, C++, Java, C#
- Interpreter — Your source code is translated into native CPU instructions as the program runs
  - JS
- Python is often described as an interpreted language——but this is only partially correct.
- Python, like many interpreted languages, actually compiles source code to a set of instructions for a virtual machine, and the Python interpreter is an implementation of that virtual machine.
- This intermediate format is called “bytecode”. The file extension is *.pyc.
- The standard implementation of Python is called “CPython”. It is the default and widely used implementation of Python. More: CPython, Cython, Jython, PyPy, Nuitka, Coco

https://www.geeksforgeeks.org/internal-working-of-python/

CPython

- When you type python at the console or install a Python distribution from python.org, you are running CPython.
  - The unique thing about CPython is that it contains both a runtime and the shared language specification that all Python runtimes use. CPython is the “official,” or reference implementation of Python
- The C in CPython is a reference to the C programming language, implying that this Python distribution is written in the C language
- The compiler in CPython is written in pure C. However, many of the standard library modules are written in pure Python or a combination of C and Python
  - Programs/python.c is a simple entry point
  - Modules/main.c contains the code to bring together the whole process, loading configuration, executing code and clearing up memory
  - Python/initconfig.c loads the configuration from the system environment and merges it with any command-line flags

Python: 岁月静好
C: 负重前行
C, not C++

https://realpython.com/cpython-source-code-guide/
https://github.com/python/cpython

Bytecode

- Byte Code is automatically created in the same directory as .py file, when a module of python is imported for the first time, or when the source is more recent than the current compiled file.
  - Next time, when the program is run, python interpreter use this file to skip the compilation step.
- Running a script is not considered an import and no .pyc file will be created.
  - Running a script is not considered an import and no .pyc file will be created. For instance, let's write a script file abc.py that imports another module xyz.py. Now run abc.py file, xyz.pyc will be created since xyz is imported, but no abc.pyc file will be created since abc.py isn't being imported.
- Advantage of bytecode
  - Portability: Bytecode can be executed on any platform with a Python interpreter, ensuring that Python code is cross-platform.
  - Performance: While Python is an interpreted language, the use of bytecode allows for some level of optimization, making code execution faster than if it were interpreted directly from the source.
  - Security: Bytecode is harder to reverse-engineer than source code, offering a layer of protection against code theft.

Bytecode: compile

- The compileall and py_compile module is part of the python standard library, so there is no need to install anything extra to use it.
    1. Using py_compile.compile function: The py_compile module can manually compile any module.
    2. Using py_compile.main() function: It compiles several files at a time.
    3. Using compileall.compile_dir() function: It compiles every single python file present in the directory supplied.
- Using py_compile in Terminal:
    - python -m py_compile File1.py File2.py File3.py ...
- For Interactive Compilation of files
    - python -m py_compile -
        File1.py
        File2.py
        File3.py
- Using compileall in Terminal: This command will automatically go recursively into sub directories and make .pyc files for all the python files it finds.
    - python -m compileall

Compile

lec2.py

```
def add(x, y):
    return x + y + x*y

```

lec3.py

```
import lec2

def add(x, y):
    return x*y - x - y


print(lec2.add(1, 2))
print(lec2.add(-1, 1))
print(lec2.add(3.1, 2.7))

print(add(1, 2))
print(add(-1, 1))
print(add(3.1, 2.7))


```

lec4.py

```
import lec2
import lec3

def add(x, y):
    return x + y


print(lec2.add(1, 2))
print(lec2.add(-1, 1))
print(lec2.add(3.1, 2.7))

print(lec3.add(1, 2))
print(lec3.add(-1, 1))
print(lec3.add(3.1, 2.7))


print(add(1, 2))
print(add(-1, 1))
print(add(3.1, 2.7))

```

- 直接运行lec4.py, 会出现lec2.py和lec3.py的pyc文件，但是不会出现lec4的pyc文件。pyc文件在__pycache__文件夹

```
import py_compile
py_compile.compile('lec4.py')
```

```
import compileall
compileall.compile_dir('.')
```

python -m compileall

compile() Built-in Function: compile a source code string or a file into a code object that can be executed by the Python interpreter.

Disassemble bytecode

- dis module: 反汇编
- co_consts is a tuple of any literals that occur in the function body
- co_varnames is a tuple containing the names of any local variables used in the function body
- co_names is a tuple of any non-local names referenced in the function body

```python
import dis


def hello():
    print("Hello, World!")


dis.dis(hello)
```

```
7           0 RESUME                   0

8           2 LOAD_GLOBAL              1 (NULL + print)
           14 LOAD_CONST               1 ('Hello, World!')
           16 PRECALL                  1
           20 CALL                     1
           30 POP_TOP
           32 LOAD_CONST               0 (None)
           34 RETURN_VALUE
```

```python
print(hello.__code__)
print(hello.__code__.co_consts)
print(hello.__code__.co_varnames)
print(hello.__code__.co_names)
```

```
<code object hello at 0x000001A3732FD0D0, file "c:\Users\popeC\OneDrive\CS124计算导论\2023 秋季\course_code.py", line 7>
(None, 'Hello, World!')
()
('print',)
```

exec()可以直接执行pyc文件

Reference

- Inside the Python Virtual Machine by Obi Ike-Nwosu is a free online book that does a deep dive into the Python interpreter, explaining in detail how Python actually works.
- A Python Interpreter Written in Python by Allison Kaptur is a tutorial for building a Python bytecode interpreter in—what else—Python itself, and it implements all the machinery to run Python bytecode.
- Finally, the CPython interpreter is open source and you can read through it on GitHub. The implementation of the bytecode interpreter is in the file Python/ceval.c. Here's that file for the Python 3.6.4 release; the bytecode instructions are handled by the switch statement beginning on line 1266.
- To learn more, attend James Bennett's talk, A Bit about Bytes: Understanding Python Bytecode, at PyCon Cleveland 2018.
- Anthony Shaw, “CPython Internals”
- http://www.python.org
- https://jython-devguide.readthedocs.io/en/latest/compiler.html
- https://docs.python.org/3/library/py_compile.html
- https://docs.python.org/2/library/compileall.html
- https://devguide.python.org/internals/compiler/
- https://devguide.python.org/internals/interpreter/