

# Классы

## В питоне класс:

- ✓ объединяет вместе данные и функции для работы с ними; в ООП эти функции называют методами
- ✓ создаются динамически по мере необходимости
- ✓ создание нового класса создает новый тип объекта

## ● Оператор class

```
class ClassName(base classes):  
    class body
```

🚫 понятия `private`, `protected` – отсутствуют!

класс Point: точка на плоскости

```
class Point:
    def __init__(self,x=0,y=0):
        self.x = x
        self.y = y

    def dist_to_origin(self):
        return sqrt(self.x**2+self.y**2)

    def dist_to_point(self,other):
        return sqrt((self.x-other.x)**2+(self.y-other.y)**2)
```

```
p1=Point(1,1)           # 1-й объект класса Point
p2=Point(1,-1)          # 2-й объект класса Point
print( p1.dist_to_origin() ) # 1.41421356237
print( p1.dist_to_point(p2) ) # 2.0
```

## Пояснения к классу `Point`:

- ✓ переменная `self` — ссылка на объект класса к которому применяется метод; `self` — «условное имя» для первого аргумента метода
- ✓ две переменные «принадлежащие» классу `self.x` и `self.y`
- ✓ «конструктор класса»: функция `__init__()` для инициализации объекта
- ✓ два метода класса: функции `dist_to_origin()` и `dist_to_point()`

- 👉 в каждом методе ссылка на объект класса `self` первый аргумент!
- 👉 методы вызываются как `obj.method(...)`, то есть `obj` «подставляется» как `self`

## ● Специальные методы класса

- ✓ позволяют классам определять собственное поведение по отношению к операторам языка: «переопределения операторов»
- ✓ имеют специальные предопределенные имена:
  - `__init__(self[,...])` инициализация, **должен возвращать None**
  - `__str__(self)` вызывается функцией `str(object)`, `print()`, `format()` и должен возвращать `string`
  - `__eq__(self, other)`, (`ne`, `lt`, `le`, `gt`, `ge`) вызывается в сравнении `x==y`, (`!=`, `<`, `<=`, `>`, `>=`); должен возвращать `bool`
  - `__add__(self, other)`, (`sub`, `mul`, `truediv`, `floordiv`, `mod`, `pow`) вызывается в бинарной операции `+`, (`-`, `*`, `/`, `//`, `%`, `**`)
  - `__call__(self[,args])` позволяет использовать объект как функцию: выражение `x(args)` заменяется на вызов `type(x).__call__(x, args)`

### ● пример для `__str__`

```
_____ into class Point _____  
def __str__(self):  
    return "({}, {})".format(self.x,self.y)  
_____  
print( 'p2=',p2 )           # p2= (1, -1)
```

### ● пример для `__eq__`


```
_____ into class Point _____  
def __eq__(self,other):  
    return self.x == other.x and self.y == other.y  
_____  
if p1 != p2:  
    print('points are different')           # points are different
```

👉 `__ne__` по умолчанию вызывает `__eq__` и инвертирует результат

# Наследование классов

## Наследование (Inheritance)

- ✓ способ передать в класс наследник переменные и функции базового класса
- ✓ реализуется перечислением классов наследников в строке с оператором `class`
- ✓ прозрачный вызов методов базового класса, если они не переопределены

 Функция `super()` используется для вызова методов родительского класса

Класс **Circle**: окружность

```
class Circle(Point):
    def __init__(self,x=0,y=0,r=0):
        super(Circle, self).__init__(x,y)
        self.r = r

    def __str__(self):
        return "({}, {}, r={})".format(self.x,self.y,self.r)

    def area(self):
        return pi*self.r**2
```

```
c1=Circle(x=p1.x,y=p1.y,r=2)
print( 'c1=',c1 ) # c1= (1, 1, r=2)
print( c1.area() ) # 12.5663706144
print( c1.dist_to_origin() ) # 1.41421356237
print('p2=',p2,'dist(c1,p2)=',c1.dist_to_point(p2))
# p2= (1, -1) dist(c1,p2)= 2.0
```

## Переопределение метода `dist_to_point()` в `Circle`

```
_____ into class Circle _____  
    def dist_to_point(self, other):  
        dp=super(Circle, self).dist_to_point(other)  
        return dp-self.r  
  
print('p2=', p2, 'dist(c1,p2)=', c1.dist_to_point(p2))  
#  p2= (1, -1) dist(c1,p2)= 0.0  
  
p3=Point(1/2, 1/2) # inside the circle  
print('p3=', p3, 'dist(c1,p3)=', c1.dist_to_point(p3))  
#  p3= (0.5, 0.5) dist(c1,p3)= -1.2928932188
```



# Чтение и запись файлов


Открытие файла: `f=open(filename, mode)`

- `filename` – строинг, имя файла
- `mode` – строинг, как файл будет использоваться:
  - 'r' – чтение (по умолчанию), 'w' – запись, 'a' – запись в конец ...
- возвращает «файловый объект» `f`

пример: файл с текстовыми данными «poker\_5.txt»

```
5H 5C 6S 7S KD      2C 3S 8S 8D TD
5D 8C 9S JS AC      2C 5C 7D 8S QH
2D 9C AS AH AC      3D 6D 7D TD QD
4D 6S 9H QH QC      3D 6D 7H QD QS
2H 2D 4C 4D 4S      3C 3D 3S 9S 9D
```

```
f = open('poker_5.txt')
print(f) # <_io.TextIOWrapper ...
f.close()
print(f.closed) # True
```

 Нормальное завершение программы в python не гарантирует закрытие файла: поэтому всегда вызывайте `f.close()`

## Чтение всего файла

```
with open('poker_5.txt') as f:  
    contents = f.read()  
  
print(contents)
```

- ✓ `with` гарантирует вызов `f.close()` при любом завершении программы
- ✓ `f.read()` читает весь файл в строку
- ✓ `f.read(size)` читает не более `size` байт

- из файла всегда читается текст, затем используйте: `int`, `float`, ...
- `f.readline()` возвращает строку со строкой, включая `'\n'`
- `f.readlines()` возвращает список строк всего файла;  
`list(f)` – то же самое

👉 если при чтении вернулся пустой строка это означает конец файла

## «Лёгкое» чтение из файла по строкам: эффективно и быстро

```
with open('poker_5.txt') as f:
    for line in f:
        print(line)
```

```
with open('poker_5.txt') as f:
    # с нумерацией строк
    for i, line in enumerate(f):
        print(f'#{i+1} {line}')
```

## Еще два способа: тоже самое, но писать больше

```
with open('poker_5.txt') as f:
    for line in f.readlines():
        print(line)
```

```
with open('poker_5.txt') as f:
    line=f.readline()
    while line != '':
        print (line)
        line=f.readline()
```

## Запись в файл:

```
line='This is test'
i=10
with open('Test.txt','w') as fw:
    fw.write(line+' : '+str(i)+'\n')
```

👉 для записи в файл все объекты  
надо конвертировать в текст:  
`str()`

✓ `f.write(text)` – записывает  
`text` в файл

- запись буферизуется, поэтому реальное содержимое дискового файла «асинхронно» вызовом `write()`
- `f.flush()` – принудительная очистка буфера
- `close()` гарантированно вызывает `flush()`

# Обработка исключений

## Исключения (exceptions)

- Интерпретатор Python постоянно использует исключения для обработки ошибок:
  - использование несуществующей переменной: `NameError`
  - вызов неизвестной метода: `AttributeError`
  - обращение к несуществующему ключу словаря: `KeyError`
  - открытие на чтение несуществующего файла: `IOError`
- Перехват исключений выполняется с помощью `try ... except`

```
try:
    f=open('file.txt')
    i=int(f.read(2))
except IOError as e:
    print('I/O error({0}): {1}'.format(e.errno,e.strerror))
except:
    print('Unexpected error:", sys.exc_info()[0])
```

## Порядок работы try ... except:

- 1 выполняется код между `try` и `except`
- 2 при возникновении исключения выполнение тут же прерывается и вызывается блок `except` с соответствующим **ТИПОМ ИСКЛЮЧЕНИЯ**
- 3 если блок `except` не найден то исключение «вырывается» за пределы текущего блока обработки

## ВОЗМОЖНО ИСПОЛЬЗОВАТЬ `else`:

```
for file in file_list:
    try:
        f = open(file)
    except IOError:
        print('cannot open', file)
    else:
        # executed if no exeption
        print(file, 'has', len(f.readlines()), 'lines')
        f.close()
```

## Определение «очистки»: finally:

👉 Определяет действие которое должно быть выполнено в любом случае

```
def divide(x, y):  
    try:  
        result = x / y  
    except ZeroDivisionError:  
        print("division by zero!",end=''); return None  
    else:  
        print("result is", result,end=''); return result  
    finally:  
        print(" finally!")  
  
divide(2, 1)          # result is 2.0 finally!  
divide(2, 0)          # division by zero! finally!  
divide('x', 'y')      # finally! + TypeError exception
```

## Возбуждение исключений: оператор `raise`:

```
>>> raise NameError('hi')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: hi
```

## re-raise:

- если надо только определить наличие исключения, а обработка исключения планируется в другом месте:

```
try:
    raise NameError('hi')
except NameError:
    print('NameError exception!')
    raise # re-raise the exception
```



# Пользовательские исключения

- Определяются как класс наследуемый от «стандартного» класса `Exception`
- Наследование позволяет создавать разветвленную систему исключений

```
class MyErr(Exception):  
    def __init__(self, value):  
        self.value = value  
  
try:  
    raise MyErr(123)  
except MyErr as e:  
    print('MyErr({0}) exception'.format(e.value))  
  
# MyErr(123) exception
```

# Импорт модулей (библиотек)

`import` – подключения модуля к своей программе

- импорт в пространство имен совпадающих с именем модуля

```
import math
print(math.sin(1./math.pi))
```

- импорт с заменой «имени модуля»

```
import math as mm
print(mm.sin(1./mm.pi))
```

- импорт «всего» в текущее пространство имен **не рекомендуется!**

```
from math import *
print(sin(1./pi))
```

- импорт отдельных объектов с возможной заменой имени объекта

```
from math import sin as Sin, pi as Pi
print(Sin(1./Pi))
```

# Документация (Docstring) и система помощи

## Понятие о docstring

- `docstring` – это основа документирования кода в питон
- Встроенная функция `help()` выводит `docstring` на экран: `help(list)`:  
Help on class list in module builtins:  
class list(object)  
| list(iterable=(), /)...

## пример docstring в функции

```
def hello(name):  
    '''A simple function that says "Hello"''' # one line docstring  
    print(f'Hello {name}!')  
  
hello('Dubna')           # Hello Dubna!  
help(hello)              # Help on function hello in module __main__:  
                          # hello(name)  
                          #     A simple function that says "Hello"
```

## Правила написания docstring находятся в PEP 257

- Типы docstring:
  - Для отдельных скриптов и функций
  - Для классов, и методов класса
  - Для модулей и пакетов
- Однострочные и многострочные
- Существуют разные стили написания

## Встроенная функция dir() как часть системы помощи

- Возвращает список «атрибутов» объекта

```
import cmath    # импортировали модуль cmath
dir(cmath)      # список атрибутов модуль cmath
['__doc__', '__loader__', '__name__', '__package__', '__spec__', 'acos', 'acosh',
'asin', 'asinh', 'atan', 'atanh', 'cos', 'cosh', 'e', 'exp', 'inf', 'infj',
'isclose', 'isfinite', 'isinf', 'isnan', 'log', 'log10', 'nan', 'nanj', 'phase',
'pi', 'polar', 'rect', 'sin', 'sinh', 'sqrt', 'tan', 'tanh', 'tau']
help(cmath.phase) # Help on built-in function phase
```