Nesne Tabanlı Programlama

&

Python

Dr. Cahit Karakuş

Esenyurt Üniversitesi
Bilgisayar Mühendisliği

2017

İçindekiler

Giriş

Programlama dili kavramı; doğrusal programlama, yapısal programlama ve nesne tabanlı programlama olarak üç başlık altında incelenir.

Doğrusal programlama, alt alta yazılan ve çoğu zaman aynı satırlardan oluşan, start, goto, end gibi ifadelerin kullanıldığı, program büyüdükçe karmaşıklaşan ve geliştirilmesi yeniden yazılması kadar bir zaman gerektiren bir yapısı olduğundan yapısal programlama kavramı geliştirilmiştir.

Yapısal programlama da kodlar işlevselliklerine göre mantıksal olarak parçalara bölünebilirler ve bu sayede uzun ve karmaşık programların küçük parçalar haline getirilip yazılmasını mümkün kılmaktadır.

Bu yapı üzerinde çalışmak için metotlar kullanılırken nesneye yönelimli programlamada ise sınıflar içinde metot yapısını veri yapısı ile ilişkilendirilir, nesneler her biri kendi içlerinde veri işleyebilir ve diğer nesneler ile çift yönlü veri transferinde bulunabilir, yapısal programlamada ise programlar sadece bir komut dizisi olarak görülebilmektedir. Bu nesne yönelimli programlamayı diğer yapılardan öne çıkaran en önemli özelliklerden biridir. Yüz binlerce satır kod nesne yönelimli bir dil ile yazıldığında yönetilmesi daha kolay olacaktır, çünkü kendi içerisinde yüzlerce nesneye bölünecektir. Amaç geliştirilen programın mümkün olduğu kadar küçük ve anlamlı parçalara bölüp, her parçayı bir türde nesne haline getirip daha sonra bu nesneleri gerektiği yerde kullanmaktır. Özet olarak OOP reusable kodların yazılması, bu da yazılan programın esnetilebilmesi ve geliştirilebilmesinde büyük kolaylık sağlamaktadır.

Which of these languages do you know?
Assembly, C or C++, Java, Perl, Scheme, Fortran, Python, Matlab

Programlamanın temelleri

- **code** or **source code**: The sequence of instructions in a program.
- **syntax**: The set of legal structures and commands that can be used in a particular programming language.
- **output**: The messages printed to the user by a program.
- **console**: The text box onto which output is printed.
  - Some source code editors pop up the console as an external window, and others contain their own console window.

1. Matematiksel İşlemler

Python daki matematiksel operatörler: /, *, -, +

+ addition
- subtraction
* multiplication
/ division
** exponentiation
// integer division
% modulo (remainder)

Say you have three variables x, y, and z, and you want to calculate the average of their values. To expression x+y+z/3 would not work. Because division comes before addition, you would actually be calculating x + y + z/3 instead of (x+y+z)/3 .

```
>>>a=5+7
>>>print a
12
```

Aynı işlemler değişkenler ile de yapılabilir:

```
>>>a=5
>>>b=a+7
>>>print b
12
```

Karmaşık işlemler:

```
>>>a=(3+4+21)/7
>>>b=(9*4)/(2+1)-6
>>>print(a*b)-(a+b)
14
```

İki integer sayıyı böldüğümüzde eğer bir kalan oluşursa Python bunu göz ardı eder.

```
>>>13/6
2
```

Kalanlı işlemler noktalıdır:

```
>>>13/6.0
2.1666666666666665
```

String:

```
>>>a=”Hel”
>>>b=”lo”
>>>c=a+b
>>>print c
'Hello'
>>>c=c+ “world”
>>>print c
'Hello world'
```

Üs alma ve mod işlemi.

```
>>>13%2
1
>>>2**3
8
>>>3**2
9
```

Mantıksal Operatörler

== (eşit mi?), != (farklı mı), < (küçük mü?), > (büyük mü?), <= (küçükeşit mi?), >= (büyükeşit mi?) gibi işaretlerdir. Bu operatörler basit karşılaştırma işlemleri yaparlar ve yaptıkları işlemin sonucunda 'true' ya da 'false' değerleri döndürürler.

```
>>>4>=9
False

>>>’abc’!=’def’
True

>>>x=’abc’
>>>x==’abc’
True

>>>14!=9
True
>>>
```

+=, -=, *=, /= c operatörlerini ele alalım.Bu operatörler hızlı bir şekilde değişkenin değerini işleme tabi tutup ardından tekrar değişkene atamak için kullanılır.

```
>>> x = 5
>>> x = x + 6
```

```
>>> print x
11

>>> y = 5
>>> y += 6
>>> print y
11
```

Matematiksel Fonksiyonlar ve math Modülü

Matematiksel Fonksiyonlar ve math Modülü kullanabilmemiz için öncelikle çalışmamıza import etmemiz gerekiyor:

```
>>>import math
>>>

>>>dir(math)       #yazarak veya
>>>help(math)     #yazarak görebilirsiniz.
```

ceil(x), Parametreye en yakın büyük tam sayıyı verir.

```
>>>math.ceil(3.56843)
4.0
>>>math.ceil(-3.56843)
-3.0
```

copysign(x, y), y nin işaretini x e kopyalar diyebiliriz. Geri dönüş değeri yine x tir.

```
>>>math.copysign(2.56,-3)
-2.56
```

fabs(x), Girilen değerin mutlak değerini verir.

```
>>>math.fabs(-3.26)
3.26
```

factorial(x), x! i bulur. Negatif yazılmamalı

```
>>>math.factorial(4)
24
```

floor(x), aldığı değeri en yakın küçük tam sayıya yuvarlar.

```
>>> math.floor(3.56843)
3.0
>>> math.floor(-3.56843)
-4.0
```

fmod(x,y), x sayısının y sayısına bölümünden kalanı veren bir fonksiyondur. a%b işlemiyle aynıdır ama fmod ile ondalıklı sayılarıda kullanabiliriz.

```
>>> math.fmod(5,3)
2.0
>>> math.fmod(6,3.2)
2.8
```

frexp(x), değerin karakteristiğini ve mantisini veriyor. (m,k) şeklinde bir değer döndürür. Burada m mantis, k karakteristiktir.

```
>>>math.frexp(7)
(0.875, 3)
```

hypot(x,y), değerlere karşılık gelen hipotenüs uzunluğunu bulur. Öklid'in dik üçgen kanunundaki sqrt(x*x+y*y) işleminin sonucudur.

```
>>>math.hypot(3,4)
5.0
```

isinf(x), bir sayının sonsuz olup olmadığını kontrol ederek true ya da false değer döndürür.

isnan(x), girilen parametrenin belirsizlik durumunu kontrol eder. Yani (sonsuz/sonsuz), (sonsuz*0) gibi.

ldexp(x,y), x*(2**y) işlemini döndürür. (** işlemi üs alma işlemi)

```
>>>math.ldexp(3,2)
12.0
```

modf(x), Girilen değerin ondalık kısmını ve tam kısmını birbirinden ayırıp geri döndürür.Girilen sayı tam sayı da ondalıklı sayı da olabilir.

```
>>> math.modf(6.5)
(0.5, 6.0)
```

sqrt(x), karekök alır.

```
>>>math.sqrt(81)
9.0
```

trunc(x), girilen sayının ondalık kısmını atıp elinde kalan değeri geri döndürür.

```
>>>math.trunc(12.6)
12
```

>>>math.trunc(-12.6)
-12

Üssel işlemler:
pow(x,y), x^y işlemini yapan fonksiyonumuzdur. Bu fonksiyon math modülü import etmeden de kullanılabilir.
>>>math.pow(5,3)
125

>>>pow(2,3)
8

exp(x)
Bu fonksiyon bize e^x işlemini yapmaktadır.
>>>math.exp(2)
7.38905609893065

expm1(x), Bu fonksiyonun exp(x) ten tek farkı sonucu 1 çıkarıp vermesidir. Yani e^x-1 işlemini yapar.
>>>math.expm1(2)
6.38905609893065

Trigonometrik fonksiyonlar:
radians(x), Girilen açı değerinin radyan karşılığını verir:
>>> math.radians(180)
3.141592653589793

degrees(x), Radyan cinsinden girilen değerin açı olarak verir.
>>>math.degrees(1.5707963267948966)
90.0

>>>math.degrees(0.5235987755982988)
30.0

tan(x), tanjant alır.
>>>math.tan(1)
1.5574077246549023

tanh(x), Hiperbolik tanjant fonksiyonudur.
>>>math.tanh(0)

0.0

sin(x), sinüs fonksiyonudur. Değer olarak radyan girilir.
>>> math.sin(1.0471975511965976)
0.8660254037844386

sinh(x), Hiperbolik sinüs fonksiyonudur. Sinüs te nasıl değeri radyan cinsinden alıyorsa burada da radyan cinsinden geri döndürür.
>>>math.sinh(1)
1.1752011936438014

cos(x), kosinüs fonksiyonu. Burada parametre olarak radyan değer girilir.
>>>math.cos(0.5235987755982988)
0.8660254037844387

cosh(x)
Hiperbolik cosinüs fonksiyonudur.
>>>math.cosh(1)
1.5430806348152437

acos(x), arccosinüs (cosinüs fonk. tersi) fonksiyonudur ve dönen değer radyan cinsindendir.

>>>math.acos(0)
1.5707963267948966

acosh(x), Hiperbolik kosinüs fonksiyonunun tersidir ve bu da radyan cinsinden bir değer döndürür.

>>>math.acosh(1)
0.0

asin(x), Arcsinüs fonksiyonudur. Dönüş değeri radyandır.
>>>math.asin(1)
1.5707963267948966

asinh(x), Hiperbolik arcsinüs fonksiyonudur.
>>>math.asinh(0)
0.0

atan(x), arctanjant fonksiyonudur.
>>>math.atan(1)

0.7853981633974483

atan2(y,x), y/x in arctanjantını verir. Neden atan(y/x) bize yetmiyor mu derseniz atan2(y,x) fonksiyonu işaret karışıklığını gidermek için kullanılabilir.

>>>atan2(5,5) # 5/5=1 olduğundan
0.7853981633974483

atanh(x), X değerinin Hiberbolik arctanjantını radyan cinsinden veren fonksiyondur.
>>>math.atanh(0)
0.0

Logaritmik işlemler:
log(x,y), Logaritması alınacak sayı x tir. Taba ise y değeridir.
>>>math.log(8,2)
3.0

log10(x), Girilen parametrenin 10 tabanındaki logaritmasını alır.
>>>math.log10(1000)
3.0

log1p(x), 1+x değerinin e tabanındaki logaritmasını alır:
>>> math.log1p(4)
1.6094379124341003

Sabitler:
e = 2.718281828459045
pi = 3.141592653589793

Random numbers
Rasgele modülden ihtiyaç duyacağımız yalnızca randint adı verilen bir fonksiyon var. Bu fonksiyonu yüklemek için aşağıdaki ifadeyi kullanın:

from random import randint

Randint'i kullanmak basittir: randint (a, b) hem a hem de b'yi içeren, a ve b arasında rasgele bir tam sayı üretir.

```
from random import randint
x = randint(1,10)
print('A random number between 1 and 10: ', x)
```

Rastgele sayı, programı her çalıştırdığımızda farklı olacaktır.

2. Komutlar

2.1. For döngüsü

The program below asks the user for a number and prints its square, then asks for another number and prints its square, etc. It does this three times and then prints that the loop is done.

```
for i in range(3):
num = eval(input('Enter a number: '))
print ('The square of your number is', num*num)
print('The loop is now done.')
```

Değişkenlere daha açıklayıcı bir isim vermek için iyi bir neden yoksa, i, j ve k harflerini döngü değişkenleri için kullanmak, programlamada bir kuraldır.

The *range* function

Range fonksiyonuna koyduğumuz değer, kaç defa döneceğimizi belirler. Çalışma şekli, sıfırdan eksi bir değerine kadar bir sayı listesi üretmesidir. Örneğin, aralık (5) beş değer üretir: 0, 1, 2, 3 ve 4.

```
sum = 0          # Initialize sum
for i in range(1, 100):
        sum += i
print(sum)
```

The following examples show how range can be used to produce a variety of sequences:

range(10)	*0,1,2,3,4,5,6,7,8,9*
range(3,7)	*3,4,5,6*
range(2,15,3)	*2,5,8,11,14*
range(9,2,-1)	*9,8,7,6,5,4,3*

- range(1, 10, 2) 1,3,5,7,9
- range(10, 0, -1) 10,9,8,7,6,5,4,3,2,1
- range(10, 0, -2) 10,8,6,4,2
- range(2, 11, 2) 2,4,6,8,10
- range(-5, 5) −5,−4,−3,−2,−1,0,1,2,3,4
- range(1, 2) 1
- range(1, 1) (empty)
- range(1, -1) (empty)
- range(1, -1, -1) 1,0
- range(0) (empty)

Example:

10 adet sayı okusun. Toplamını ve ortalamasını bulsun.

```
s = 0
for i in range(10):
    num = eval(input('Enter a number: '))
    s = s + num
print('The average is', s/10)
```

Örnek:

Çarpım tablosu

```
# Print a multiplication table to 10 x 10
 # Print column heading
print("  1  2  3  4  5  6  7  8  9 10")
print(" +---------------------------------------")
for row in range(1, 11): # 1 <= row <= 10, table has 10 rows
    if row < 10: # Need to add space?
        print(" ", end="")
    print(row, "| ", end="") # Print heading for this row.
    for column in range(1, 11): # Table has 10 columns.
        product = row*column; # Compute product
        if product < 100: # Need to add space?
            print(end=" ")
        if product < 10: # Need to add another space?
            print(end=" ")
        print(product, end=" ") # Display product
    print() # Move cursor to next row
```

Örnek, Permute abc

```
# File permuteabc.py
# The first letter varies from A to C
for first in 'ABC':
    for second in 'ABC': # The second varies from A to C
        if second != first: # No duplicate letters allowed
            for third in 'ABC': # The third varies from A to C
                # Don't duplicate first or second letter
                if third != first and third != second:

                print(first + second + third)
```

Notice how the if statements are used to prevent duplicate letters within a given string. The output of
Listing 5.10 (permuteabc.py) is all six permutations of ABC:

ABC
ACB
BAC
BCA
CAB
CBA

	1	2	3	4	5	6	7	8	9	10	
1 |	1	2	3	4	5	6	7	8	9	10	
2 |	2	4	6	8	10	12	14	16	18	20	
3 |	3	6	9	12	15	18	21	24	27	30	
4 |	4	8	12	16	20	24	28	32	36	40	
5 |	5	10	15	20	25	30	35	40	45	50	
6 |	6	12	18	24	30	36	42	48	54	60	
7 |	7	14	21	28	35	42	49	56	63	70	
8 |	8	16	24	32	40	48	56	64	72	80	
9 |	9	18	27	36	45	54	63	72	81	90	
10		10	20	30	40	50	60	70	80	90	00

+----------

2.2. While döngüsü

Örnek: Fibonacci dizisinin ilk 20 terimini ekrana yazalım.

```
a, b = 1, 1
print a
print b
i = 2
while i<=20:
   a,b = b, a+b
   print b
   i += 1
```

Örnek: *x* değişkenini 0 ile 1 arasında 0.1 adımlarla artırarak, *x* ve *sin(x)* değerlerini bir tablo olarak yazın.

```
import math
x = 0.0
dx = 0.1
print "%6s  %6s" % ("x", "sin x")
while x <= 1.0:
  print "%1.4f  % 1.4f" % (x, math.sin(x))
   x += dx
```

Not: %6s, altı karakterlik yere bir dizenin (sağa yaslanarak) yazılacağını, %1.4f ise bir noktalı sayının virgülden önce bir, virgülden sonra ise dört basamaklı yazılacağını belirtir.

Örnek: ve *x* 0 ile 1 arasında değişirken, her satırda *x*, $f_1(x)$, $f_2(x)$, $f_3(x)$, $\sin x$ değerlerini ekrana yazın.

```
import math
x = 0.0
dx = 0.1
print "%6s %6s %6s %6s %6s" % ("x", "f1", "f2", "f3", "sin x")
while x <= 1.0:
   f1 = x - x**3 /6
   f2 = f1 + x**5/120
   f3 = f2 + x**7/5040
   print "%1.4f %1.4f %1.4f %1.4f %1.4f" % (x, f1, f2, f3, math.sin(x))
   x += dx
```

2.3. IF

The comparison operators are ==, >, <, >=, <=, and !=. That last one is for not equals. There are three additional operators used to construct more complicated conditions: and, or, and not. *if x=1:* ifadesi hatalı; *if x==1:* ifadesi doğrudur.

if grade>=80 and <90: ifadesi bir sözdizimi hatasına (syntax error) neden olacaktır. Açık ifade etmeliyiz. Doğru ifade: *if grade>=80 and grade<90:*

```
grade = eval(input('Enter your score: '))
if grade>=90:
print('A')
if grade>=80 and grade<90:
print('B')
if grade>=70 and grade<80:
print('C')
if grade>=60 and grade<70:
print('D')
if grade<60:
print('F')
```

```
grade = eval(input('Enter your score: '))
if grade>=90:
print('A')
elif grade>=80:
print('B')
elif grade>=70:
print('C'):
elif grade>=60:
print('D'):
else:
print('F')
```

Example:

This program gets 10 numbers from the user and counts how many of those numbers are greater than 10.

```
count = 0
for i in range(10):
        num = eval(input('Enter a number: '))
        if num>10:
                count=count+1
print('There are', count, 'numbers greater than 10.')
```

Example:

1^{2}'den 100^{2}' ün karesine ondalık mod bölümünde 4 kalanı olan kaç sayı vardır?

```
count = 0
for i in range(1,101):
        if (i**2)%10==4:
                count = count + 1
print(count)
```

2.4. Swapping – Yer değiştirme

```
x = y
y = x
```

ifadesi ile x'in değeri y'ye, y'nin değeri x'e atanmaz.

```
k=x
x=y
y=k
```

şeklinde, ya da Python'da aşağıdaki şekilde ifade edilir,

```
x,y = y,x
```

2.5. Strings - Diziler

The in operator

The *in* operator is used to tell if a string contains something. For example:

```
if 'a' in string:
        print('Your string contains the letter a.')
```

You can combine in with the not operator to tell if a string does not contain something:

```
if ';' not in string:
        print('Your string does not contain any semicolons.')
```

Example In the previous section we had the long if condition

```
if t=='a' or t=='e' or t=='i' or t=='o' or t=='u':
```

Using the "in" operator, we can replace that statement with the following:

```
if t in 'aeiou':
```

Indexing

```
s='Python'
s[0]            P first character of s
```

s[1] *y second character of s*
s[-1] *n last character of s*
s[-2] *o second-to-last character of s*

Slices - dilimler

s='abcdefghij'
index: 0 1 2 3 4 5 6 7 8 9
letters: a b c d e f g h i j

Code	Result	Description
s[2:5]	*cde*	*characters at indices 2, 3, 4*
s[:5]	*abcde*	*first five characters*
s[5:]	*fghij*	*characters from index 5 to the end*
s[-2:]	*ij*	*last two characters*
s[:]	*abcdefghij*	*entire string*
s[1:7:2]	*bdf*	*characters from index 1 to 6, by twos*
s[: :-1]	*jihgfedcba*	*a negative step reverses the string*

Changing individual characters of a string
Suppose we have a string called *s* and we want to change the character at index 4 of *s* to 'X'. It is tempting to try s[4]='X', but that unfortunately will not work. Python strings are immutable (değişmez), which means we can't modify any part of them.

If we want to change a character of *s*, we have to instead build a new string from s and reassign it to *s*. Here is code that will change the character at index 4 to 'X':

s = s[:4] + 'X' + s[5:]

The idea of this is we take all the characters up to index 4, then X, and then all of the characters after index 4.

lower(), returns a string with every letter of the original in lowercase
upper(), returns a string with every letter of the original in uppercase
replace(x,y), returns a string with every occurrence of x replaced by y
count(x), counts the number of occurrences of x in the string
index(x), returns the location of the first occurrence of x
isalpha(), returns True if every character of the string is a letter

İmportant note: If you want to change a string, s, to all lowercase, it is not enough to just use *s.lower().* You need to do the following:

s = s.lower()

2.6. Lists - Listeler

L = [1,2,3]

The empty list is []. It is the list equivalent of 0 or ''.

len— The number of items in L is given by len(L).
in— The in operator tells you if a list contains something.

[7,8]+[3,4,5], [7,8,3,4,5]
*[7,8]*3, [7,8,7,8,7,8]*
*[0]*5, [0,0,0,0,0]*

len returns the number of items in the list
sum returns the sum of the items in the list
min returns the minimum of the items in the list
max returns the maximum of the items in the list

append(x), adds x to the end of the list
sort(), sorts the list
count(x), returns the number of times x occurs in the list
index(x), returns the location of the first occurrence of x
reverse(), reverses the list
remove(x), removes first occurrence of x from the list
pop(p), removes the item at index p and returns its value
insert(p,x), inserts x at index p of the list

Important note: There is a big difference between list methods and string methods: String methods do not change the original string, but list methods do change the original list. To sort a list L, just use L.sort() and not L=L.sort(). In fact, the latter will not work at all.

Two-dimensional lists

```
L =   [ [1,2,3],
        [4,5,6],
        [7,8,9]]
```

Shortcuts

```
count=count+1,          count+=1
total=total-5,          total-=5
prod=prod*2,            prod*=2

a = 0
b = 0
c = 0
A nice shortcut is:
a = b = c = 0

x = L[0]
y = L[1]
z = L[2]
Instead, we can do this:
x,y,z = L
```

Similarly, we can assign three variables at a time like below:

```
x,y,z = 1,2,3
```

And, as we have seen once before, we can swap variables using this kind of assignment.

```
x,y,z = y,z,x

if a==0 and b==0 and c==0:      if a==b==c==0:
if 1<a and a<b and b<5:         if 1<a<b<5:
```

String formatting

```
a = 23.60 * .25
print('The tip is {:.2f}'.format(a))
```

To format integers, the formatting code is {:d}.

```
print('{:3d}'.format(2))
print('{:3d}'.format(25))
print('{:3d}'.format(138))
```

To center integers instead of right-justifying, use the ^ character, and to left-justify, use the < character.

```
print('{:^5d}'.format(2))
print('{:^5d}'.format(222))
print('{:^5d}'.format(13834))
```

The ^ and < characters center and left-justify floats. Formatting strings To format strings, the formatting code is {:s}. Here is an example that centers some text:

```
print('{:^10s}'.format('Hi'))
print('{:^10s}'.format('there!'))
```

To right-justify a string, use the > character:

```
print('{:>6s}'.format('Hi'))
print('{:>6s}'.format('There'))
```

2.7. Dictionaries - Sözlükler

Here is a dictionary of the days in the months of the year:

```
days = {'January':31, 'February':28, 'March':31, 'April':30,
        'May':31, 'June':30, 'July':31, 'August':31,
        'September':30, 'October':31, 'November':30, 'December':31}
```

Example: You can use a dictionary as an actual dictionary of definitions:

```
d = {'dog' : 'has a tail and goes woof!',
        'cat' : 'says meow',
        'mouse' : 'chased by cats'}
```

Here is an example of the dictionary in use:

```
word = input('Enter a word: ')
print('The definition is:', d[word])

Enter a word: mouse
The definition is: chased by cats
```

2.8. Why Doesn't Python Have Switch/Case?

Unlike every other programming language I've used before, Python does not have a switch or case statement. To get around this fact, we use dictionary mapping:
Daha önce kullandığımız diğer programlama dillerinin aksine Python'da bir “switch” veya “case” ifadesi yok. Bu gerçeği çözmek için sözlük haritalama kullanıyoruz:

The switch statement can be very useful sometimes, and it is pretty easy to handle multiple conditions instead of writing a lot of if else. Here is the example of writing switch statement in most high-level programming languages:

```
switch(n)
{
 case 0:
        print("You selected blue.");
        break;
 case 1:
        print("You selected yellow.");
        break;
 case 2:
        print("You selected green.");
        break;
}
```

Type the following code into the window — pressing Enter after each line:

```
def PrintBlue():
        print("You chose blue!rn")
def PrintRed():
        print("You chose red!rn")
def PrintOrange():
        print("You chose orange!rn")
def PrintYellow():
        print("You chose yellow!rn")
```

Type the following code into the window — pressing Enter after each line:

```
ColorSelect = {
 0: PrintBlue,
 1: PrintRed,
 2: PrintOrange,
 3: PrintYellow
}
```

This code is the dictionary. Each key is like the case part of the switch statement. The values specify what to do. In other words, this is the switch structure. The functions that you created earlier are the action part of the switch — the part that goes between the case statement and the break clause.

Type the following code into the window — pressing Enter after each line:

```
Selection = 0
while (Selection != 4):
        print("0. Blue")
        print("1. Red")
        print("2. Orange")
        print("3. Yellow")
        print("4. Quit")
        Selection = int(input("Select a color option: "))
        if (Selection >= 0) and (Selection < 4):
                ColorSelect[Selection]()
```

Finally, you see the user interface part of the example. The code begins by creating an input variable, Selection. It then goes into a loop until the user enters a value of 4.

During each loop, the application displays a list of options and then waits for user input. When the user does provide input, the application performs a range check on it. Any value between 0 and 3 selects one of the functions defined earlier using the dictionary as the switching mechanism.

2.9. Fonksiyonlar

```
def print_hello(n):
        print('Hello ' * n)
        print()

print_hello(3)
print_hello(5)
times = 2
print_hello(times)

def convert(t):
        return t*9/5+32

print(convert(20))
```

Example: Let us write our own sine function that works in degrees.

```
from math import pi, sin

def deg_sin(x):
        return sin(pi*x/180)
```

Example:
x = (de- b f)=(ad - bc) and
y = (a f- ce)=(ad - bc). We need our function to return both the x and y solutions.

```
        def solve(a,b,c,d,e,f):
                x = (d*e-b*f)/(a*d-b*c)
                y = (a*f-c*e)/(a*d-b*c)
                return [x,y]

xsol, ysol = solve(2,3,4,1,2,5)
print('The solution is x = ', xsol, 'and y = ', ysol)

The solution is x = 1.3 and y = -0.2
```

First-class functions

Python functions are said to be first-class functions, which means they can be assigned to variables, copied, used as arguments to other functions, etc., just like any other object.

Copying functions Here is an example where we make a copy of a function:

```
def f(x):
return x*x
g = f

print('f(3) =', f(3), 'g(3) =', g(3), sep = '\n')
f(3) = 9
g(3) = 9
```

Lists of functions:

```
def f(x):
return x**2
def g(x):
return x**3
funcs = [f, g]
print(funcs[0](5), funcs[1](5), sep='\n')

25
125
231
```

3. Sınıf (Class)

Python nesne tabanlı bir dildir. Python da nesneleri (Object) olusturmak için sınıflar (Class) kullanılır. Sınıflar, değiskenlerden ve fonksiyonlardan oluşur.

Fonksiyonlar, değişkenler gibi herhangi bir bilgiyi saklamaz - her fonksiyon çalıştırıldığında yeni başlar. ***Bununla birlikte, bazı işlevler ve değişkenler birbirleriyle çok yakından ilgilidir ve birbirleriyle çok fazla etkileşime girmesi gerekir.***

Gerekli olan, birbirleriyle etkileşim kurabilmeleri için birbirine yakın bir yerde bulunan fonksiyonları ve değişkenleri gruplamanın bir yoludur.

Class (Sınıf):
Herhangi bir nesnenin özelliklerini (attributes) içeren kullanıcının tanımladığı bir prototip koddur. Bu özellikler members (elemanları) ve methods (fonksiyonları) olarak geçer ve bunlara erişim sağlanırken nesneden sonra bir nokta kullanılır.

Class variable (Sınıf değişkeni):
Bu değişken sınıftan türetilmiş tüm örnekler tarafından kullanılan ortak değişkendir. Class'ın içerisinde tanımlanır. Class'ın dışında onun metodları (alt fonksiyonları) tarafından tanımlanamaz. Class değişkenleri ondan türetilmiş örnekler (instance) tarafından kullanılamaz.

Instance variable (Örnek değişkeni):
Bu değişken bir metodun içinde tanımlanan ve sadece sınıfın o anki örneğine ait olan değişkendir.

Instance (örnek):
Belli bir sınıftan türetilmiş bağımsız bir birey nesnedir. Bir nesne hangi sınıfa ait ise onun bir örnek nesnesidir.

Method (metod):
Bir sınıf içinde tanımlanan özel bir fonksiyon türüdür.

Object (Nesne):
Class tarafından oluşturulan örnek bir veri yapısıdır. Yani bir kopyasıdır.

Inheritance (Miras):
Bir Class'ın özelliklerinin başka bir Class'a aktarılmasıdır.

Defining a class:

```
# Defining a class
class class_name:
   [statement 1]
   [statement 2]
   [statement 3]
   [etc.]
```

Example of a Class:

Here is an example that creates the definition of a Shape:

```
#An example of a class
class Shape:

   def __init__(self, x, y):
      self.x = x
      self.y = y
      self.description = "This shape has not been described yet"
      self.author = "Nobody has claimed to make this shape yet"

   def area(self):
      return self.x * self.y

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

   def describe(self, text):
      self.description = text

   def authorName(self, text):
      self.author = text

   def scaleSize(self, scale):
      self.x = self.x * scale
      self.y = self.y * scale
```

What you have created is a description of a shape (That is, the variables) and what operations you can do with the shape (That is, the functions). This is very important - you have not made an actual shape, simply the description of what a shape is.

The shape has a width (x), a height (y), and an area and perimeter (area(self) and perimeter(self)). No code is run when you define a class - you are simply making functions and variables.

The function called __init__ is run when we create an instance of Shape - that is, when we create an actual shape, as opposed to the 'blueprint' we have here, __init__ is run. You will understand how this works later.

self is how we refer to things in the class from within itself. self is the first parameter in any function defined inside a class. Any function or variable created on the first level of indentation (that is, lines of code that start one TAB to the right of where we put class Shape) is automatically put into self. To access these functions and variables elsewhere inside the class, their name must be preceeded with self and a full-stop (e.g. self.variable_name).

Creating a class:
It's all well and good that we can make a class, but how do we use one? Here is an example of what we call "creating an instance of a class". Assume that the code in example has already been run:

rectangle = Shape(100, 45)

What has been done? It takes a little explaining...
The __init__ function really comes into play at this time. We create an instance of a class by first giving its name (in this case, Shape) and then, in brackets, the values to pass to the __init__ function. The init function runs (using the parameters you gave it in brackets) and then spits out an instance of that class, which in this case is assigned to the name "rectangle".

Accessing attributes from outside an instance:
Think of our class instance, rectangle, as a self-contained collection of variables and functions. In the same way that we used self to access functions and variables of the class instance from within itself, we use the name that we assigned to it now (rectangle) to access functions and variables of the class instance from outside of itself. Following on from the code we ran above, we would do this:

#finding the area of your rectangle:
print(rectangle.area())

#finding the perimeter of your rectangle:
print(rectangle.perimeter())

```
#describing the rectangle
rectangle.describe("A wide rectangle, more than twice\
 as wide as it is tall")

#making the rectangle 50% smaller
rectangle.scaleSize(0.5)

#re-printing the new area of the rectangle
print(rectangle.area())
```

As you see, where self would be used from within the class instance, its assigned name is used when outside the class. We do this to view and change the variables inside the class, and to access the functions that are there.

More than one instance:
We aren't limited to a single instance of a class - we could have as many instances as we like. I could do this:

```
long_rectangle = Shape(120,10)
fat_rectangle = Shape(130,120)
```

Both long_rectangle and fat_rectangle have their own functions and variables contained inside them - they are totally independent of each other. There is no limit to the number of instances I could create.

Lingo:
Object-oriented programming has a set of lingo that is associated with it. It's about time that we have this all cleared up:

- when we first describe a class, we are defining it (like with functions)
- the ability to group similar functions and variables together is called encapsulation
- the word 'class' can be used when describing the code where the class is defined (like how a function is defined), and it can also refer to an instance of that class - this can get confusing, so make sure you know in which form we are talking about classes
- a variable inside a class is known as an 'attribute'
- a function inside a class is known as a 'method'
- a class is in the same category of things as variables, lists, dictionaries, etc. That is, they are objects
- a class is known as a 'data structure' - it holds data, and the methods to process that data.
-

Inheritance - Miras:

Sınıfların, nitelikler ve yöntemler olarak bilinen değişkenleri ve işlevleri nasıl bir araya getirdiklerini biliyoruz. Oluşturduğumuz her yeni nesne için yeni bir kod yazmak zorunda kalmamak için, o sınıfın herhangi bir örneğini yaratabiliriz. Pekikoda ekstra özellikler eklemek için ne yapmak gerekir? Mirasın başladığı yer burasıdır.

Python kalıtım işlemini gerçekten kolaylaştırır. Başka bir 'ana' sınıfına dayanan yeni bir sınıf tanımlıyoruz. Yeni sınıfımız herşeyi ebeveynden getiriyor ve buna başka şeyler de ekleyebiliyoruz. Herhangi bir yeni nitelik veya yöntem, ana sınıfımızda bir nitelik veya yöntemle aynı isme sahipse, üst sınıf yerine kullanılır.

The Shape class

```
class Shape:
    def __init__(self,x,y):
        self.x = x
        self.y = y
        self.description = "This shape has not been described yet"
        self.author = "Nobody has claimed to make this shape yet"
    def area(self):
        return self.x * self.y
    def perimeter(self):
        return 2 * self.x + 2 * self.y
    def describe(self,text):
        self.description = text
    def authorName(self,text):
        self.author = text
    def scaleSize(self,scale):
        self.x = self.x * scale
        self.y = self.y * scale
```

Using inheritance

If we wanted to define a new class, let's say a square, based on our previous Shape class, we would do this:

```
class Square(Shape):
    def __init__(self,x):
        self.x = x
        self.y = x
```

It is just like normally defining a class, but this time we put the parent class that we inherited from in brackets after the name. As you can see, this lets us describe a square really quickly. That's because we inherited everything from the shape class, and changed only what needed to be changed. In this case we redefined the __init__ function of Shape so that the X and Y values are the same. Let's take from what we have learnt, and create another new class, this time inherited from Square. It will be two squares, one immediately left of the other:

DoubleSquare class

```
# The shape looks like this:
# _________
#|   |   |
#|   |   |
#|___|___|

class DoubleSquare(Square):
    def __init__(self,y):
        self.x = 2 * y
        self.y = y
    def perimeter(self):
        return 2 * self.x + 3 * self.y
```

This time, we also had to redefine the perimeter function, as there is a line going down the middle of the shape. Try creating an instance of this class. As a helpful hint, the IDLE command line starts where your code ends - so typing a line of code is like adding that line to the end of the program you have written.

Pointers and Dictionaries of Classes

Thinking back, when you say that one variable equals another, e.g. variable2 = variable1, the variable on the left-hand side of the equal-sign takes on the value of the variable on the right. With class instances, this happens a little differently - the name on the left becomes the class instance on the right. So in instance2 = instance1, instance2 is 'pointing' to instance1 - there are two names given to the one class instance, and you can access the class instance via either name.

In other languages, you do things like this using pointers, however in python this all happens behind the scenes.

The final thing that we will cover is dictionaries of classes. Keeping in mind what we have just learnt about pointers, we can assign an instance of a class to an entry in a list or dictionary. This allows for virtually any amount of class instances to exist when our program is run. Lets have a look at the example below, and see how it describes what I am talking about:

Dictionaries of Classes

```
# Again, assume the definitions on Shape,
# Square and DoubleSquare have been run.
# First, create a dictionary:
dictionary = {}

# Then, create some instances of classes in the dictionary:
dictionary["DoubleSquare 1"] = DoubleSquare(5)
dictionary["long rectangle"] = Shape(600,45)

#You can now use them like a normal class:
print(dictionary["long rectangle"].area())

dictionary["DoubleSquare 1"].authorName("The Gingerbread Man")
print(dictionary["DoubleSquare 1"].author)
```

Example:

Classes are a way of combining information and behavior. For example, let's consider what you'd need to do if you were creating a rocket ship in a game, or in a physics simulation. One of the first things you'd want to track are the x and y coordinates of the rocket. Here is what a simple rocket ship class looks like in code:

```
class Rocket():
    # Rocket simulates a rocket ship for a game,
    #  or a physics simulation.

    def __init__(self):
        # Each rocket has an (x,y) position.
        self.x = 0
        self.y = 0
```

One of the first things you do with a class is to define the ***_init_*()** method. The __init__() method sets the values for any parameters that need to be defined when an object is first created. The *self* part will be explained later; basically, it's a syntax that allows you to access a variable from anywhere else in the class.

The Rocket class stores two pieces of information so far, but it can't do anything. The first behavior to define is a core behavior of a rocket: moving up. Here is what that might look like in code:

```
class Rocket():
    # Rocket simulates a rocket ship for a game,
    #  or a physics simulation.

    def __init__(self):
        # Each rocket has an (x,y) position.
        self.x = 0
        self.y = 0

    def move_up(self):
        # Increment the y-position of the rocket.
        self.y += 1
```

The Rocket class can now store some information, and it can do something. But this code has not actually created a rocket yet. Here is how you actually make a rocket:

```
class Rocket():
    # Rocket simulates a rocket ship for a game,
    #  or a physics simulation.

    def __init__(self):
        # Each rocket has an (x,y) position.
        self.x = 0
        self.y = 0

    def move_up(self):
        # Increment the y-position of the rocket.
        self.y += 1

# Create a Rocket object.
my_rocket = Rocket()
print(my_rocket)
```

```
<__main__.Rocket object at 0x7f6f50c39190>
```

To actually use a class, you create a variable such as *my_rocket*. Then you set that equal to the name of the class, with an empty set of parentheses. Python creates an **object** from the class. An object is a single instance of the Rocket class; it has a copy of each of the class's variables, and it can do any action that is defined for the class. In this case, you can see that the variable my_rocket is a Rocket object from the __main__ program file, which is stored at a particular location in memory.

Once you have a class, you can define an object and use its methods. Here is how you might define a rocket and have it start to move up:

```
class Rocket():
    # Rocket simulates a rocket ship for a game,
    #  or a physics simulation.

    def __init__(self):
        # Each rocket has an (x,y) position.
        self.x = 0
        self.y = 0

    def move_up(self):
        # Increment the y-position of the rocket.
        self.y += 1

# Create a Rocket object, and have it start to move up.
my_rocket = Rocket()
print("Rocket altitude:", my_rocket.y)

my_rocket.move_up()
print("Rocket altitude:", my_rocket.y)

my_rocket.move_up()
print("Rocket altitude:", my_rocket.y)
```

```
Rocket altitude: 0
Rocket altitude: 1
Rocket altitude: 2
```

To access an object's variables or methods, you give the name of the object and then use *dot notation* to access the variables and methods. So to get the y-value of *my_rocket*, you use *my_rocket.y*. To use the move_up() method on my_rocket, you write *my_rocket.move_up()*.

Once you have a class defined, you can create as many objects from that class as you want. Each object is its own instance of that class, with its own separate variables. All of the objects are capable of the same behavior, but each object's particular actions do not affect any of the other objects. Here is how you might make a simple fleet of rockets:

```
class Rocket():
    # Rocket simulates a rocket ship for a game,
    #  or a physics simulation.

    def __init__(self):
        # Each rocket has an (x,y) position.
        self.x = 0
        self.y = 0

    def move_up(self):
        # Increment the y-position of the rocket.
        self.y += 1

# Create a fleet of 5 rockets, and store them in a list.
my_rockets = []
for x in range(0,5):
    new_rocket = Rocket()
    my_rockets.append(new_rocket)

# Show that each rocket is a separate object.
for rocket in my_rockets:
    print(rocket)
```

```
<__main__.Rocket object at 0x7f6f5073cd10>
<__main__.Rocket object at 0x7f6f5073cc90>
<__main__.Rocket object at 0x7f6f5073cbd0>
<__main__.Rocket object at 0x7f6f5077e410>
<__main__.Rocket object at 0x7f6f5077ead0>
```

You can see that each rocket is at a separate place in memory. By the way, if you understand list comprehensions, you can make the fleet of rockets in one line:

```
class Rocket():
    # Rocket simulates a rocket ship for a game,
    #  or a physics simulation.

    def __init__(self):
        # Each rocket has an (x,y) position.
        self.x = 0
        self.y = 0

    def move_up(self):
        # Increment the y-position of the rocket.
        self.y += 1

# Create a fleet of 5 rockets, and store them in a list.
my_rockets = [Rocket() for x in range(0,5)]

# Show that each rocket is a separate object.
for rocket in my_rockets:
    print(rocket)
```

output:

```
<__main__.Rocket object at 0x7f6f50789190>
<__main__.Rocket object at 0x7f6f50763450>
<__main__.Rocket object at 0x7f6f507634d0>
<__main__.Rocket object at 0x7f6f50763510>
<__main__.Rocket object at 0x7f6f50763550>
```

You can prove that each rocket has its own x and y values by moving just one of the rockets:

```
class Rocket():
    # Rocket simulates a rocket ship for a game,
    #  or a physics simulation.

    def __init__(self):
        # Each rocket has an (x,y) position.
        self.x = 0
        self.y = 0

    def move_up(self):
        # Increment the y-position of the rocket.
        self.y += 1

# Create a fleet of 5 rockets, and store them in a list.
my_rockets = [Rocket() for x in range(0,5)]

# Move the first rocket up.
my_rockets[0].move_up()

# Show that only the first rocket has moved.
for rocket in my_rockets:
    print("Rocket altitude:", rocket.y)
```

output:

```
Rocket altitude: 1
Rocket altitude: 0
Rocket altitude: 0
Rocket altitude: 0
Rocket altitude: 0
```

The syntax for classes may not be very clear at this point, but consider for a moment how you might create a rocket without using classes. You might store the x and y values in a dictionary, but you would have to write a lot of ugly, hard-to-maintain code to manage even a small set of rockets. As more features become incorporated into the Rocket class, you will see how much more efficiently real-world objects can be modeled with classes than they could be using just lists and dictionaries.

When you write a class in Python 2.7, you should always include the word object in parentheses when you define the class. This makes sure your Python 2.7 classes act like Python 3 classes, which will be helpful as your projects grow more complicated.

The simple version of the rocket class would look like this in Python 2.7:

```
class Rocket(object):
    # Rocket simulates a rocket ship for a game,
    #  or a physics simulation.

    def __init__(self):
        # Each rocket has an (x,y) position.
        self.x = 0
        self.y = 0
```

This syntax will work in Python 3 as well.

Rocket With No Class

- Using just what you already know, try to write a program that simulates the above example about rockets.
    - Store an x and y value for a rocket.
    - Store an x and y value for each rocket in a set of 5 rockets. Store these 5 rockets in a list.
- Don't take this exercise too far; it's really just a quick exercise to help you understand how useful the class structure is, especially as you start to see more capability added to the Rocket class.

Object-Oriented terminology

Classes are part of a programming paradigm called **object-oriented programming**. Object-oriented programming, or OOP for short, focuses on building reusable blocks of code called classes. When you want to use a class in one of your programs, you make an **object** from that class, which is where the phrase "object-oriented" comes from. Python itself is not tied to object-oriented programming, but you will be using objects in most or all of your Python projects. In order to understand classes, you have to understand some of the language that is used in OOP.

General terminology

A **class** is a body of code that defines the **attributes** and **behaviors** required to accurately model something you need for your program. You can model something from the real world, such as a rocket ship or a guitar string, or you can model something from a virtual world such as a rocket in a game, or a set of physical laws for a game engine.

An **attribute** is a piece of information. In code, an attribute is just a variable that is part of a class.

A **behavior** is an action that is defined within a class. These are made up of **methods**, which are just functions that are defined for the class.

An **object** is a particular instance of a class. An object has a certain set of values for all of the attributes (variables) in the class. You can have as many objects as you want for any one class.

There is much more to know, but these words will help you get started. They will make more sense as you see more examples, and start to use classes on your own.

A closer look at the Rocket class

Now that you have seen a simple example of a class, and have learned some basic OOP terminology, it will be helpful to take a closer look at the Rocket class.

The __init__() method

Here is the initial code block that defined the Rocket class:

```
class Rocket():
    # Rocket simulates a rocket ship for a game,
    #  or a physics simulation.

    def __init__(self):
        # Each rocket has an (x,y) position.
        self.x = 0
        self.y = 0
```

The first line shows how a class is created in Python. The keyword **class** tells Python that you are about to define a class. The rules for naming a class are the same rules you learned about naming variables, but there is a strong convention among Python programmers that classes should be named using CamelCase. If you are unfamiliar with CamelCase, it is a convention where each letter that starts a word is capitalized, with no underscores in the name. The name of the class is followed by a set of parentheses. These parentheses will be empty for now, but later they may contain a class upon which the new class is based.

It is good practice to write a comment at the beginning of your class, describing the class. There is a more formal syntax for documenting your classes, but you can wait a little bit to get that formal. For now, just write a comment at the beginning of your class summarizing what you intend the class to do. Writing more formal documentation for your classes will be easy later if you start by writing simple comments now.

Function names that start and end with two underscores are special built-in functions that Python uses in certain ways. The __init()__ method is one of these special functions. It is called automatically when you create an object from your class. The __init()__ method lets you make sure that all relevant attributes are set to their proper values when an object is created from the class, before the object is used. In this case, The __init__() method initializes the x and y values of the Rocket to 0.

The **self** keyword often takes people a little while to understand. The word "self" refers to the current object that you are working with. When you are writing a class, it lets you refer to certain attributes from any other part of the class. Basically, all methods in a class need the *self* object as their first argument, so they can access any attribute that is part of the class.

Now let's take a closer look at a **method**. Here is the method that was defined for the Rocket class:

```
class Rocket():
    # Rocket simulates a rocket ship for a game,
    #  or a physics simulation.

    def __init__(self):
        # Each rocket has an (x,y) position.
        self.x = 0
        self.y = 0

    def move_up(self):
        # Increment the y-position of the rocket.
        self.y += 1
```

A method is just a function that is part of a class. Since it is just a function, you can do anything with a method that you learned about with functions. You can accept positional arguments, keyword arguments, an arbitrary list of argument values, an arbitrary dictionary of arguments, or any combination of these. Your arguments can return a value or a set of values if you want, or they can just do some work without returning any values.

Each method has to accept one argument by default, the value **self**. This is a reference to the particular object that is calling the method. This *self*argument gives you access to the calling object's attributes. In this example, the self argument is used to access a Rocket object's y-value. That value is increased by 1, every time the method move_up() is called by a particular Rocket object. This is probably still somewhat confusing, but it should start to make sense as you work through your own examples.

If you take a second look at what happens when a method is called, things might make a little more sense:

```
class Rocket():
    # Rocket simulates a rocket ship for a game,
    #  or a physics simulation.

    def __init__(self):
        # Each rocket has an (x,y) position.
        self.x = 0
        self.y = 0

    def move_up(self):
        # Increment the y-position of the rocket.
        self.y += 1

# Create a Rocket object, and have it start to move up.
my_rocket = Rocket()
print("Rocket altitude:", my_rocket.y)

my_rocket.move_up()
print("Rocket altitude:", my_rocket.y)

my_rocket.move_up()
print("Rocket altitude:", my_rocket.y)
```

output:

```
Rocket altitude: 0
Rocket altitude: 1
Rocket altitude: 2
```

In this example, a Rocket object is created and stored in the variable my_rocket. After this object is created, its y value is printed. The value of the attribute *y* is accessed using dot notation. The phrase *my_rocket.y* asks Python to return "the value of the variable y attached to the object my_rocket".

After the object my_rocket is created and its initial y-value is printed, the method move_up() is called. This tells Python to apply the method move_up() to the object my_rocket. Python finds the y-value associated with my_rocket and adds 1 to that value. This process is repeated several times, and you can see from the output that the y-value is in fact increasing.

Making multiple objects from a class

One of the goals of object-oriented programming is to create reusable code. Once you have written the code for a class, you can create as many objects from that class as you need. It is worth mentioning at this point that classes are usually saved in a separate file, and then imported into the program you are working on. So you can build a library of classes, and use those classes over and over again in different programs. Once you know a class works well, you can leave it alone and know that the objects you create in a new program are going to work as they always have.

You can see this "code reusability" already when the Rocket class is used to make more than one Rocket object. Here is the code that made a fleet of Rocket objects:

```
class Rocket():
    # Rocket simulates a rocket ship for a game,
    #  or a physics simulation.

    def __init__(self):
        # Each rocket has an (x,y) position.
        self.x = 0
        self.y = 0

    def move_up(self):
        # Increment the y-position of the rocket.
        self.y += 1

# Create a fleet of 5 rockets, and store them in a list.
my_rockets = []
for x in range(0,5):
    new_rocket = Rocket()
    my_rockets.append(new_rocket)

# Show that each rocket is a separate object.
for rocket in my_rockets:
    print(rocket)
```

output:

```
<__main__.Rocket object at 0x7f6f50763090>
<__main__.Rocket object at 0x7f6f5077ead0>
<__main__.Rocket object at 0x7f6f50763990>
<__main__.Rocket object at 0x7f6f50763a10>
<__main__.Rocket object at 0x7f6f50763a50>
```

If you are comfortable using list comprehensions, go ahead and use those as much as you can. I'd rather not assume at this point that everyone is comfortable with comprehensions, so I will use the slightly longer approach of declaring an empty list, and then using a for loop to fill that list. That can be done slightly more efficiently than the previous example, by eliminating the temporary variable *new_rocket*:

```
class Rocket():
    # Rocket simulates a rocket ship for a game,
    #  or a physics simulation.

    def __init__(self):
        # Each rocket has an (x,y) position.
        self.x = 0
        self.y = 0

    def move_up(self):
        # Increment the y-position of the rocket.
        self.y += 1

# Create a fleet of 5 rockets, and store them in a list.
my_rockets = []
for x in range(0,5):
    my_rockets.append(Rocket())

# Show that each rocket is a separate object.
for rocket in my_rockets:
    print(rocket)
```

Output

```
<__main__.Rocket object at 0x7f6f50763990>
<__main__.Rocket object at 0x7f6f50763a10>
<__main__.Rocket object at 0x7f6f50763750>
<__main__.Rocket object at 0x7f6f506fd8d0>
<__main__.Rocket object at 0x7f6f506fd6d0>
```

What exactly happens in this for loop? The line *my_rockets.append(Rocket())* is executed 5 times. Each time, a new Rocket object is created and then added to the list my_rockets. The __init__() method is executed once for each of these objects, so each object gets its own x and y value. When a method is called on one of these objects, the *self* variable allows access to just that object's attributes, and ensures that modifying one object does not affect any of the other objecs that have been created from the class.

Each of these objects can be worked with individually. At this point we are ready to move on and see how to add more functionality to the Rocket class. We will work slowly, and give you the chance to start writing your own simple classes.

A quick check-in

If all of this makes sense, then the rest of your work with classes will involve learning a lot of details about how classes can be used in more flexible and powerful ways. If this does not make any sense, you could try a few different things:

- Reread the previous sections, and see if things start to make any more sense.
- Type out these examples in your own editor, and run them. Try making some changes, and see what happens.
- Try the next exercise, and see if it helps solidify some of the concepts you have been reading about.
- Read on. The next sections are going to add more functionality to the Rocket class. These steps will involve rehashing some of what has already been covered, in a slightly different way.

Classes are a huge topic, and once you understand them you will probably use them for the rest of your life as a programmer. If you are brand new to this, be patient and trust that things will start to sink in.

Your Own Rocket

- Without looking back at the previous examples, try to recreate the Rocket class as it has been shown so far.
    - Define the Rocket() class.
    - Define the __init__() method, which sets an x and a y value for each Rocket object.
    - Define the move_up() method.
    - Create a Rocket object.
    - Print the object.
    - Print the object's y-value.
    - Move the rocket up, and print its y-value again.
    - Create a fleet of rockets, and prove that they are indeed separate Rocket objects.

Refining the Rocket class

The Rocket class so far is very simple. It can be made a little more interesting with some refinements to the __init__() method, and by the addition of some methods.

Accepting parameters for the __init__() method

The __init__() method is run automatically one time when you create a new object from a class. The __init__() method for the Rocket class so far is pretty simple:

```
class Rocket():
    # Rocket simulates a rocket ship for a game,
    #  or a physics simulation.

    def __init__(self):
        # Each rocket has an (x,y) position.
        self.x = 0
        self.y = 0

    def move_up(self):
        # Increment the y-position of the rocket.
        self.y += 1
```

All the __init__() method does so far is set the x and y values for the rocket to 0. We can easily add a couple keyword arguments so that new rockets can be initialized at any position:

```
class Rocket():
    # Rocket simulates a rocket ship for a game,
    #  or a physics simulation.

    def __init__(self, x=0, y=0):
        # Each rocket has an (x,y) position.
        self.x = x
        self.y = y

    def move_up(self):
        # Increment the y-position of the rocket.
        self.y += 1
```

Now when you create a new Rocket object you have the choice of passing in arbitrary initial values for x and y:

```
class Rocket():
    # Rocket simulates a rocket ship for a game,
    #  or a physics simulation.

    def __init__(self, x=0, y=0):
        # Each rocket has an (x,y) position.
        self.x = x
        self.y = y

    def move_up(self):
        # Increment the y-position of the rocket.
        self.y += 1

# Make a series of rockets at different starting places.
rockets = []
rockets.append(Rocket())
rockets.append(Rocket(0,10))
rockets.append(Rocket(100,0))

# Show where each rocket is.
for index, rocket in enumerate(rockets):
    print("Rocket %d is at (%d, %d)." % (index, rocket.x, rocket.y))
```

hide output

```
Rocket 0 is at (0, 0).
Rocket 1 is at (0, 10).
Rocket 2 is at (100, 0).
```

Accepting parameters in a method

The __init__ method is just a special method that serves a particular purpose, which is to help create new objects from a class. Any method in a class can accept parameters of any kind. With this in mind, the move_up() method can be made much more flexible. By accepting keyword arguments, the move_up() method can be rewritten as a more general move_rocket() method. This new method will allow the rocket to be moved any amount, in any direction:

```
class Rocket():
    # Rocket simulates a rocket ship for a game,
    #  or a physics simulation.

    def __init__(self, x=0, y=0):
        # Each rocket has an (x,y) position.
        self.x = x
        self.y = y

    def move_rocket(self, x_increment=0, y_increment=1):
        # Move the rocket according to the paremeters given.
        #  Default behavior is to move the rocket up one unit.
        self.x += x_increment
        self.y += y_increment
```

The paremeters for the move() method are named x_increment and y_increment rather than x and y. It's good to emphasize that these are changes in the x and y position, not new values for the actual position of the rocket. By carefully choosing the right default values, we can define a meaningful default behavior. If someone calls the method move_rocket() with no parameters, the rocket will simply move up one unit in the y-direciton. Note that this method can be given negative values to move the rocket left or right:

```
class Rocket():
    # Rocket simulates a rocket ship for a game,
    #  or a physics simulation.

    def __init__(self, x=0, y=0):
        # Each rocket has an (x,y) position.
        self.x = x
        self.y = y

    def move_rocket(self, x_increment=0, y_increment=1):
        # Move the rocket according to the paremeters given.
        #  Default behavior is to move the rocket up one unit.
        self.x += x_increment
        self.y += y_increment

# Create three rockets.
rockets = [Rocket() for x in range(0,3)]

# Move each rocket a different amount.
rockets[0].move_rocket()
rockets[1].move_rocket(10,10)
rockets[2].move_rocket(-10,0)

# Show where each rocket is.
for index, rocket in enumerate(rockets):
    print("Rocket %d is at (%d, %d)." % (index, rocket.x, rocket.y))
```

Output

```
Rocket 0 is at (0, 1).
Rocket 1 is at (10, 10).
Rocket 2 is at (-10, 0).
```

Adding a new method

One of the strengths of object-oriented programming is the ability to closely model real-world phenomena by adding appropriate attributes and behaviors to classes. One of the jobs of a team piloting a rocket is to make sure the rocket does not get too close to any other rockets. Let's add a method that will report the distance from one rocket to any other rocket.

If you are not familiar with distance calculations, there is a fairly simple formula to tell the distance between two points if you know the x and y values of each point. This new method performs that calculation, and then returns the resulting distance.

```
from math import sqrt

class Rocket():
    # Rocket simulates a rocket ship for a game,
    #  or a physics simulation.

    def __init__(self, x=0, y=0):
        # Each rocket has an (x,y) position.
        self.x = x
        self.y = y

    def move_rocket(self, x_increment=0, y_increment=1):
        # Move the rocket according to the paremeters given.
        #  Default behavior is to move the rocket up one unit.
        self.x += x_increment
        self.y += y_increment

    def get_distance(self, other_rocket):
        # Calculates the distance from this rocket to another rocket,
        #  and returns that value.
        distance = sqrt((self.x-other_rocket.x)**2+(self.y-other_rocket.y)**2)
        return distance

# Make two rockets, at different places.
rocket_0 = Rocket()
rocket_1 = Rocket(10,5)

# Show the distance between them.
distance = rocket_0.get_distance(rocket_1)
print("The rockets are %f units apart." % distance)
```

hide output

```
The rockets are 11.180340 units apart.
```

Hopefully these short refinements show that you can extend a class' attributes and behavior to model the phenomena you are interested in as closely as you want. The rocket could have a name, a crew capacity, a payload, a certain amount of fuel, and any number of other attributes. You could define any behavior you want for the rocket, including interactions with other rockets and launch facilities, gravitational fields, and whatever you need it to! There are techniques for managing these more complex interactions, but what you have just seen is the core of object-oriented programming.
At this point you should try your hand at writing some classes of your own. After trying some exercises, we will look at object inheritance, and then you will be ready to move on for now.

Inheritance
One of the most important goals of the object-oriented approach to programming is the creation of stable, reliable, reusable code. If you had to create a new class for every kind of object you wanted to model, you would hardly have any reusable code. In Python and any other language that supports OOP, one class can **inherit** from another class. This means you can base a new class on an existing class; the new class *inherits* all of the attributes and behavior of the class it is based on. A new class can override any undesirable attributes or behavior of the class it inherits from, and it can add any new attributes or behavior that are appropriate. The original class is called the **parent** class, and the new class is a **child** of the parent class. The parent class is also called a **superclass**, and the child class is also called a **subclass**.

The child class inherits all attributes and behavior from the parent class, but any attributes that are defined in the child class are not available to the parent class. This may be obvious to many people, but it is worth stating. This also means a child class can override behavior of the parent class. If a child class defines a method that also appears in the parent class, objects of the child class will use the new method rather than the parent class method.
To better understand inheritance, let's look at an example of a class that can be based on the Rocket class.

The SpaceShuttle class
If you wanted to model a space shuttle, you could write an entirely new class. But a space shuttle is just a special kind of rocket. Instead of writing an entirely new class, you can inherit all of the attributes and behavior of a Rocket, and then add a few appropriate attributes and behavior for a Shuttle.
One of the most significant characteristics of a space shuttle is that it can be reused. So the only difference we will add at this point is to record the number of flights the shutttle has completed. Everything else you need to know about a shuttle has already been coded into the Rocket class.

Here is what the Shuttle class looks like:

```
from math import sqrt

class Rocket():
    # Rocket simulates a rocket ship for a game,
    #  or a physics simulation.

    def __init__(self, x=0, y=0):
        # Each rocket has an (x,y) position.
        self.x = x
        self.y = y

    def move_rocket(self, x_increment=0, y_increment=1):
        # Move the rocket according to the paremeters given.
        #  Default behavior is to move the rocket up one unit.
        self.x += x_increment
        self.y += y_increment

    def get_distance(self, other_rocket):
        # Calculates the distance from this rocket to another rocket,
        #  and returns that value.
        distance = sqrt((self.x-other_rocket.x)**2+(self.y-other_rocket.y)**2)
        return distance

class Shuttle(Rocket):
    # Shuttle simulates a space shuttle, which is really
    #  just a reusable rocket.

    def __init__(self, x=0, y=0, flights_completed=0):
        super().__init__(x, y)
        self.flights_completed = flights_completed

shuttle = Shuttle(10,0,3)
print(shuttle)
```

output:

```
<__main__.Shuttle object at 0x7f1e62ba6cd0>
```

When a new class is based on an existing class, you write the name of the parent class in parentheses when you define the new class:

```
class NewClass(ParentClass):
```

The __init__() function of the new class needs to call the __init__() function of the parent class. The __init__() function of the new class needs to accept all of the parameters required to build an object from the parent class, and these parameters need to be passed to the __init__() function of the parent class. The *super().__init__()* function takes care of this:

```
class NewClass(ParentClass):
    def __init__(self, arguments_new_class, arguments_parent_class):
    super().__init__(arguments_parent_class)
    # Code for initializing an object of the new class.
```

The *super()* function passes the *self* argument to the parent class automatically. You could also do this by explicitly naming the parent class when you call the __init__() function, but you then have to include the *self* argument manually:

```
class Shuttle(Rocket):
  # Shuttle simulates a space shuttle, which is really
  #  just a reusable rocket.

  def __init__(self, x=0, y=0, flights_completed=0):
    Rocket.__init__(self, x, y)
    self.flights_completed = flights_completed
```

This might seem a little easier to read, but it is preferable to use the *super()* syntax. When you use *super()*, you don't need to explicitly name the parent class, so your code is more resilient to later changes. As you learn more about classes, you will be able to write child classes that inherit from multiple parent classes, and the *super()* function will call the parent classes' __init__() functions for you, in one line. This explicit approach to calling the parent class' __init__() function is included so that you will be less confused if you see it in someone else's code.

The output above shows that a new Shuttle object was created. This new Shuttle object can store the number of flights completed, but it also has all of the functionality of the Rocket class: it has a position that can be changed, and it can calculate the distance between itself and other rockets or shuttles. This can be demonstrated by creating several rockets and shuttles, and then finding the distance between one shuttle and all the other shuttles and rockets. This example uses a simple function called randint, which generates a random integer between a lower and upper bound, to determine the position of each rocket and shuttle:

```
from math import sqrt
from random import randint
```

```
class Rocket():
    # Rocket simulates a rocket ship for a game,
    #  or a physics simulation.

    def __init__(self, x=0, y=0):
        # Each rocket has an (x,y) position.
        self.x = x
        self.y = y

    def move_rocket(self, x_increment=0, y_increment=1):
        # Move the rocket according to the paremeters given.
        #  Default behavior is to move the rocket up one unit.
        self.x += x_increment
        self.y += y_increment

    def get_distance(self, other_rocket):
        # Calculates the distance from this rocket to another rocket,
        #  and returns that value.
        distance = sqrt((self.x-other_rocket.x)**2+(self.y-other_rocket.y)**2)
        return distance

class Shuttle(Rocket):
    # Shuttle simulates a space shuttle, which is really
    #  just a reusable rocket.

    def __init__(self, x=0, y=0, flights_completed=0):
        super().__init__(x, y)
        self.flights_completed = flights_completed


# Create several shuttles and rockets, with random positions.
#  Shuttles have a random number of flights completed.
shuttles = []
for x in range(0,3):
    x = randint(0,100)
    y = randint(1,100)
    flights_completed = randint(0,10)
    shuttles.append(Shuttle(x, y, flights_completed))

rockets = []
for x in range(0,3):
```

```
    x = randint(0,100)
    y = randint(1,100)
    rockets.append(Rocket(x, y))

# Show the number of flights completed for each shuttle.
for index, shuttle in enumerate(shuttles):
    print("Shuttle %d has completed %d flights." % (index, shuttle.flights_completed))

print("\n")
# Show the distance from the first shuttle to all other shuttles.
first_shuttle = shuttles[0]
for index, shuttle in enumerate(shuttles):
    distance = first_shuttle.get_distance(shuttle)
    print("The first shuttle is %f units away from shuttle %d." % (distance, index))

print("\n")
# Show the distance from the first shuttle to all other rockets.
for index, rocket in enumerate(rockets):
    distance = first_shuttle.get_distance(rocket)
    print("The first shuttle is %f units away from rocket %d." % (distance, index))
```

output:

```
Shuttle 0 has completed 7 flights.
Shuttle 1 has completed 10 flights.
Shuttle 2 has completed 7 flights.

The first shuttle is 0.000000 units away from shuttle 0.
The first shuttle is 25.806976 units away from shuttle 1.
The first shuttle is 40.706265 units away from shuttle 2.

The first shuttle is 25.079872 units away from rocket 0.
The first shuttle is 60.415230 units away from rocket 1.
The first shuttle is 35.468296 units away from rocket 2.
```

Inheritance is a powerful feature of object-oriented programming. Using just what you have seen so far about classes, you can model an incredible variety of real-world and virtual phenomena with a high degree of accuracy. The code you write has the potential to be stable and reusable in a variety of applications.

Inheritance in Python 2.7

The *super()* method has a slightly different syntax in Python 2.7:

```
class NewClass(ParentClass):

    def __init__(self, arguments_new_class, arguments_parent_class):
        super(NewClass, self).__init__(arguments_parent_class)
        # Code for initializing an object of the new class.
```

Notice that you have to explicitly pass the arguments *NewClass* and *self* when you call *super()* in Python 2.7. The SpaceShuttle class would look like this:

```
from math import sqrt

class Rocket(object):
    # Rocket simulates a rocket ship for a game,
    #  or a physics simulation.

    def __init__(self, x=0, y=0):
        # Each rocket has an (x,y) position.
        self.x = x
        self.y = y

    def move_rocket(self, x_increment=0, y_increment=1):
        # Move the rocket according to the paremeters given.
        #  Default behavior is to move the rocket up one unit.
        self.x += x_increment
        self.y += y_increment

    def get_distance(self, other_rocket):
        # Calculates the distance from this rocket to another rocket,
        #  and returns that value.
        distance = sqrt((self.x-other_rocket.x)**2+(self.y-other_rocket.y)**2)
        return distance
```

```
class Shuttle(Rocket):
    # Shuttle simulates a space shuttle, which is really
    #  just a reusable rocket.

    def __init__(self, x=0, y=0, flights_completed=0):
        super(Shuttle, self).__init__(x, y)
        self.flights_completed = flights_completed

shuttle = Shuttle(10,0,3)
print(shuttle)
```
hide output
```
<__main__.Shuttle object at 0x7f1e60080810>
```
This syntax works in Python 3 as well.

Modules and classes

Now that you are starting to work with classes, your files are going to grow longer. This is good, because it means your programs are probably doing more interesting things. But it is bad, because longer files can be more difficult to work with. Python allows you to save your classes in another file and then import them into the program you are working on. This has the added advantage of isolating your classes into files that can be used in any number of different programs. As you use your classes repeatedly, the classes become more reliable and complete overall.

Storing a single class in a module

When you save a class into a separate file, that file is called a **module**. You can have any number of classes in a single module. There are a number of ways you can then import the class you are interested in.

Start out by saving just the Rocket class into a file called *rocket.py*. Notice the naming convention being used here: the module is saved with a lowercase name, and the class starts with an uppercase letter. This convention is pretty important for a number of reasons, and it is a really good idea to follow the convention.

```
# Save as rocket.py
from math import sqrt

class Rocket():
    # Rocket simulates a rocket ship for a game,
    #  or a physics simulation.

    def __init__(self, x=0, y=0):
        # Each rocket has an (x,y) position.
        self.x = x
        self.y = y

    def move_rocket(self, x_increment=0, y_increment=1):
        # Move the rocket according to the paremeters given.
        #  Default behavior is to move the rocket up one unit.
        self.x += x_increment
        self.y += y_increment

    def get_distance(self, other_rocket):
        # Calculates the distance from this rocket to another rocket,
        #  and returns that value.
        distance = sqrt((self.x-other_rocket.x)**2+(self.y-other_rocket.y)**2)
        return distance
```

Make a separate file called *rocket_game.py*. If you are more interested in science than games, feel free to call this file something like *rocket_simulation.py*. Again, to use standard naming conventions, make sure you are using a lowercase_underscore name for this file.

```
# Save as rocket_game.py
from rocket import Rocket

rocket = Rocket()
print("The rocket is at (%d, %d)." % (rocket.x, rocket.y))
```

output:

```
The rocket is at (0, 0).
```

This is a really clean and uncluttered file. A rocket is now something you can define in your programs, without the details of the rocket's implementation cluttering up your file. You don't have to include all the class code for a rocket in each of your files that deals with rockets; the code defining rocket attributes and behavior lives in one file, and can be used anywhere.

The first line tells Python to look for a file called *rocket.py*. It looks for that file in the same directory as your current program. You can put your classes in other directories, but we will get to that convention a bit later. Notice that you do not When Python finds the file *rocket.py*, it looks for a class called *Rocket*. When it finds that class, it imports that code into the current file, without you ever seeing that code. You are then free to use the class Rocket as you have seen it used in previous examples.

Storing multiple classes in a module

A module is simply a file that contains one or more classes or functions, so the Shuttle class actually belongs in the rocket module as well:

```
# Save as rocket.py
from math import sqrt

class Rocket():
    # Rocket simulates a rocket ship for a game,
    #  or a physics simulation.

    def __init__(self, x=0, y=0):
        # Each rocket has an (x,y) position.
        self.x = x
        self.y = y

    def move_rocket(self, x_increment=0, y_increment=1):
        # Move the rocket according to the paremeters given.
        #  Default behavior is to move the rocket up one unit.
        self.x += x_increment
        self.y += y_increment

    def get_distance(self, other_rocket):
        # Calculates the distance from this rocket to another rocket,
        #  and returns that value.
        distance = sqrt((self.x-other_rocket.x)**2+(self.y-other_rocket.y)**2)
        return distance

class Shuttle(Rocket):
    # Shuttle simulates a space shuttle, which is really
    #  just a reusable rocket.

    def __init__(self, x=0, y=0, flights_completed=0):
        super().__init__(x, y)
        self.flights_completed = flights_completed
```

Now you can import the Rocket and the Shuttle class, and use them both in a clean uncluttered program file:

```
# Save as rocket_game.py
from rocket import Rocket, Shuttle

rocket = Rocket()
print("The rocket is at (%d, %d)." % (rocket.x, rocket.y))
```

```
shuttle = Shuttle()
print("\nThe shuttle is at (%d, %d)." % (shuttle.x, shuttle.y))
print("The shuttle has completed %d flights." % shuttle.flights_completed)
```

output:

```
The rocket is at (0, 0).

The shuttle is at (0, 0).
The shuttle has completed 0 flights.
```

The first line tells Python to import both the *Rocket* and the *Shuttle* classes from the *rocket* module. You don't have to import every class in a module; you can pick and choose the classes you care to use, and Python will only spend time processing those particular classes.

A number of ways to import modules and classes

There are several ways to import modules and classes, and each has its own merits.

```
import module_name
```

The syntax for importing classes that was just shown:
from module_name import ClassName
is straightforward, and is used quite commonly. It allows you to use the class names directly in your program, so you have very clean and readable code. This can be a problem, however, if the names of the classes you are importing conflict with names that have already been used in the program you are working on. This is unlikely to happen in the short programs you have been seeing here, but if you were working on a larger program it is quite possible that the class you want to import from someone else's work would happen to have a name you have already used in your program. In this case, you can use simply import the module itself:

```
# Save as rocket_game.py
import rocket

rocket_0 = rocket.Rocket()
print("The rocket is at (%d, %d)." % (rocket_0.x, rocket_0.y))

shuttle_0 = rocket.Shuttle()
print("\nThe shuttle is at (%d, %d)." % (shuttle_0.x, shuttle_0.y))
print("The shuttle has completed %d flights." % shuttle_0.flights_completed)
```

hide output

```
The rocket is at (0, 0).

The shuttle is at (0, 0).
The shuttle has completed 0 flights.
```

The general syntax for this kind of import is:
import module_name
After this, classes are accessed using dot notation:
module_name.ClassName

This prevents some name conflicts. If you were reading carefully however, you might have noticed that the variable name *rocket* in the previous example had to be changed because it has the same name as the module itself. This is not good, because in a longer program that could mean a lot of renaming.

import *module_name* as *local_module_name*

There is another syntax for imports that is quite useful:
import module_name as local_module_name
When you are importing a module into one of your projects, you are free to choose any name you want for the module in your project. So the last example could be rewritten in a way that the variable name *rocket* would not need to be changed:

```
# Save as rocket_game.py
import rocket as rocket_module

rocket = rocket_module.Rocket()
print("The rocket is at (%d, %d)." % (rocket.x, rocket.y))

shuttle = rocket_module.Shuttle()
print("\nThe shuttle is at (%d, %d)." % (shuttle.x, shuttle.y))
print("The shuttle has completed %d flights." % shuttle.flights_completed)
```

output:

```
The rocket is at (0, 0).

The shuttle is at (0, 0).
The shuttle has completed 0 flights.
```

This approach is often used to shorten the name of the module, so you don't have to type a long module name before each class name that you want to use. But it is easy to shorten a name so much that you force people reading your code to scroll to the top of your file and see what the shortened name stands for. In this example,
import rocket as rocket_module
leads to much more readable code than something like:
import rocket as r
from *module_name* import *

There is one more import syntax that you should be aware of, but you should probably avoid using. This syntax imports all of the available classes and functions in a module:
from module_name import *
This is not recommended, for a couple reasons. First of all, you may have no idea what all the names of the classes and functions in a module are. If you accidentally give one of your variables the same name as a name from the module, you will have naming conflicts. Also, you may be importing way more code into your program than you need.

If you really need all the functions and classes from a module, just import the module and use the module_name.ClassName syntax in your program.

You will get a sense of how to write your imports as you read more Python code, and as you write and share some of your own code.

A module of functions

You can use modules to store a set of functions you want available in different programs as well, even if those functions are not attached to any one class. To do this, you save the functions into a file, and then import that file just as you saw in the last section. Here is a really simple example; save this is *multiplying.py*:

```
# Save as multiplying.py
def double(x):
    return 2*x

def triple(x):
    return 3*x

def quadruple(x):
    return 4*x
```

Now you can import the file *multiplying.py*, and use these functions. Using the from module_name import function_name syntax:

```
from multiplying import double, triple, quadruple

print(double(5))
print(triple(5))
print(quadruple(5))
```

output:

```
10
15
20
```

Using the import module_name syntax:

```
import multiplying

print(multiplying.double(5))
print(multiplying.triple(5))
print(multiplying.quadruple(5))
```

output:

```
10
15
20
```

Using the import module_name as local_module_name syntax:

import multiplying as m

print(m.double(5))
print(m.triple(5))
print(m.quadruple(5))

Output
10
15
20

Using the from module_name import * syntax:
from multiplying import *

print(double(5))
print(triple(5))
print(quadruple(5))
hide output
10
15
20

4. Nesne Tabanlı Programlama'nın temel bileşenleri

Bir dilin kendisini nesne yönelimli olarak tanımlayabilmesi için, üç kavramı desteklemesi gerekir: nesneler, sınıflar ve kalıtım - objects, classes, and inheritance. Ancak nesne yönelimli diller; Encapsulation (sarmalama), Inheritance (Miras Alma) ve Polymorphism (Çok Biçimlilik) üçlüsü üzerine oturtulmuş diller olarak düşünülür. Tanımlamadaki bu değişimin nedeni, geçen yıllar boyunca encapsulation ve polymorphism'in class ve inheritance gibi nesne yönelimli sistemler oluşturmanın bir iç parçası olduğunun farkına varılmasıdır.

Nedir bu Nesne? Her şey nesnedir! Etrafınıza bir bakın, ilk gözünüze çarpan şey nedir? Ben bu yazıyı bilgisayar ekraının önünde klavye tuşlarına basarak yazıyorum, bu bilgisayarın bir ağırlığı, rengi, markası var, bunlar bu bigisayarın özellikleri. Aynı zamanda bu bilgisayar üzerinden internete bağlanıyorum, mesajlar okuyorum; müzik dinilyorum bunlar ise davranışı. Nesneler iki temel bileşenden oluşur. Bunlar özellik (property) ve davranış (behavior)'dır. Nesneleri yazmak için önce bir şablon(kalıp) oluşturulur, daha sonra bu şablondan istediğimiz kad ar nesneyi çıkartabiliriz. Bu şablonlara class, bu şablonlardan oluşturulan nesnelere'de object adı verilir. Class ile object arasındaki ilişki bu şekildedir.

Sınıf nesneyi tanımlayan bir veri türü, nesne ise sınıftan türetilebilen bir yapıdır. Nesneleri yazmak için önce bir şablon oluşturur, sonra bu şablondan istediğimiz kadar nesneyi çıkartabiliriz.

4.1. Access Modifiers (Erişim Belirleyiciler)

Sınısal yazılım ile veri depolama alanı arasındaki bağ çok önemlidir. Ram kısaca, stack (depo) ve heap (yığın) olan iki alandan oluşur. Stack, local variable'ların olduğu, heap ise class instance'larının olduğu alandır.

Yazdığımız program içerisindeki classlara, bu classlar içindeki metodlara, değişkenlere nasıl ve ne zaman erişeceğimizi belirten kurallar var, bunlara Acces Modifiers (Erişim Belirleyiciler) denir. Bu kurallar nesne içerisindeki üyelerin erişim durumlarını belirler. Bunlar: public, private, protected, internal, protected internal 'dır. Şimdi bunları inceleyelim.

public: Public olarak tanımlanan üye tüm erişimlere açıktır.
protected: Kendi üyesi olduğu (tanımlandığı) sınıflardan ve bu sınıftan türetilmiş sınıflardan erişilir.
private: Yalnızca üyesi olduğu sınıftan erişilir.
internal: Aynı program(assembly) içerisinden erişilir.
protected internal: Class'ın içinden ve ondan türetilen sınıfların içinden erişilir.

Why python does not have access modifier?And what are there alternatives in python? Because Python is not C# or Java. Why do those languages not have dynamic types and significant whitespace? Because they are not Python.

4.2. Constructor (Yapıcı) - Destructor (Yıkıcı) Metot

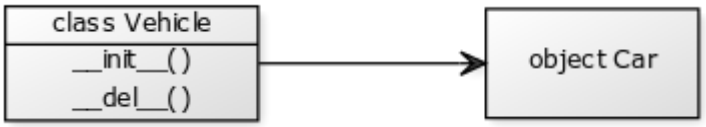
Sınıflardan bir nesne oluşturulduğu zaman, nesneyi ram'de inşa ederken başlangıç olarak o nesne ile ilgili yapısını değiştirecek bazı farklılıklar olsun isteyebiliriz, bunun için constructor adı verilen özel bir metot var. Aslında bir nesneyi türetirken **new** anahtar sözcüğünü kullandığımızda default olarak bir constructor oluşturmuş oluruz, eğer kendi constructor'ımızı oluşturmazsak default olarak bu temel alınacaktır. (variable initilializer) Constructor sınıf adı ile aynı adı taşır, dışarıdan parametre alabilirler, geriye değer döndürmezler, overload yapılabilirler.

Bir constructor oluşturuyoruz, constructor sınıf adı ile aynı adı taşır, birden fazla constroctor olabilir. Nesneyi her ürettiğimiz yerde tekrar bu marka değişkenine değer atamak zorunda kalmayız. İstisnai durumlarda ise overload yapılabilsin diye, ikinci construcor parametre alarak marka değişkenine değer atanması sağlanır.

int, byte, char gibi değer tiplerinin de yapıcı yordamları bulunmaktadır, kod yazarken bu değerlere tiplerine ilk değerlerini vermeden kullanmak istediğimizde hata alınır.

Oluşturduğumuz nesnenin, bellekten silinmeden önceki gerçekleştireceği son işlemi gerçekleştiren metottur.

Aşağıdaki örnek sınıf (Class) bir yapıcı (__init__) ve yıkıcı (__del__) içerir.
Sınıftan bir örnek oluşturuyor ve hemen sonra silmesini sağlıyoruz.




```
class Vehicle:
   def __init__(self):
      print('Vehicle created.')
    def __del__(self):
      print('Destructor called, vehicle deleted.')

car = Vehicle()
del car
```

Bu çıktıyı görmek için programı çalıştırıldığında:
Vehicle created.
Destructor called, vehicle deleted.

4.3. Encapsulation (Sarmalama)

Encapsulation adı verilen yapı, bir sınıf içerisindeki değişkenlere “kontrollü bir şekilde erişimi sağlamak /denetlemek” için kullanılan bir yapıdır. Class içerisindeki değişken privete yapılırak dışarıdan direkt erişilmesi engellenir, bu değişken içerisine değer atıp değer okumak için get ve set adı verilen metodlar kullanılır. Yani direkt değişkene değil, bu değişkene erişmek (işlem yapmak) için bu metodlar ile çalışılır. set değeri alıp değişkene atar, get’de değeri geri döndürür, tabii bu işlem sizin belirlediğiniz, olması gereken kurallar çerçevesi içinde gerçekleşir.

Python'un özel anahtar sözcüğü yoktur (private keyword), diğer nesne tabanlı dillerin aksine, kapsülleme yapılabilir. Bunun yerine, sözleşmeye dayanır: doğrudan erişilmesi gereken bir sınıf değişkenine altçizgi başlığı eklenmelidir.

Kapsülleme, verilerin nitelik olarak bir yerde güvenli bir şekilde depolanmasını sağlamakla ilgilidir. Kapsülleme şunları bildirir:

- Verilere yalnızca örnek yöntemlerle erişilmelidir.
- Veriler, sınıf yöntemlerinde belirlenen doğrulama gereksinimine dayalı olarak her zaman doğru olmalıdır.
- Veriler, dış işlemlerdeki değişikliklerden korunmalıdır.

Nesne özellik değerlerine erişmek için 'setter' ve 'getter' yöntemlerinin kullanılması gerekir. Değişken / öznitelik değeri ayarlamak ve almak için bir ayarlayıcı ve alma yöntemi (dikkat edin, ayarlayıcı ve alıcı gerekli değildir) kullanan bir sınıf örneği.

Örnek:

```
class Tuxlabs(object):
   def setter(self, website):
      self.website = website

   def getter(self):
      return self.website

tuxlabs = Tuxlabs()

tuxlabs.setter('http://www.tuxlabs.com')
print (tuxlabs.getter())

tuxlabs.setter('http://www.google.com')
print (tuxlabs.getter())
```

Kapsülleme işleminin Python tarafından zorlanmadığını bilmek önemlidir. Bu nedenle, bir sınıfı kullanan bir programcı, getter ve setter yöntemleri yoluyla verilere erişmek zorunda değildir. Örneğin, yukarıdaki örnek sınıftaki birisi web sitesini, belirleyici kurucu ve alıcı sınıflarıyla hiç etkileşim içine girmeden ayarlayabilir. Buna, kapsüllemeyi kırmak denir ve sınıf yazarının ve yöneticisinin, erişilen verilerin artık geçerli hale gelememesi ve bunun sınıfa bağımlı bir programda öngörülemeyen sorunlara neden olabileceği için bir programcı için kötü bir form / pratiktir. Burada, kapsüllemenin kırıldığı değiştirilmiş bir örnek.

```
class Tuxlabs(object):
   def setter(self, website):
      if 'http' in website:
         self.website = website
      else:
         print ('ERROR: Unable to set website to %s.\nRequired format is:
http://www.example.com' % (website))

   def getter(self):
      output = 'Website is set to: ' + str(self.website) + '\n'
      return output

tuxlabs = Tuxlabs()

website = 'http://www.tuxlabs.com'
print ("Attempting to set website to: \"%s\"" % (website))
tuxlabs.setter(website)
print (tuxlabs.getter())

website = 'http://www.google.com'
print ("Attempting to set website to: \"%s\"" % (website))
tuxlabs.setter(website)
print (tuxlabs.getter())

website = 'I should not be accessing website in the class directly!, but since I am breaking
encapsulation it does still work, it\'s just naughty!'
print ("Attempting to set website to: \"%s\"" % (website))
tuxlabs.website = website
print (tuxlabs.website) ## Printing without the getter doh !

website = 'This is not a website, so an error is thrown!'
print ("Attempting to set website to: \"%s\"" % (website))
tuxlabs.setter(website)
```

İşte yukarıdaki örneğin çıktısı:

```
Attempting to set website to: "http://www.tuxlabs.com"
Website is set to: http://www.tuxlabs.com

Attempting to set website to: "http://www.google.com"
Website is set to: http://www.google.com

Attempting to set website to: "I should not be accessing website in the class directly!, but since I am breaking encapsulation it does still work, it's just naughty!"
I should not be accessing website in the class directly!, but since I am breaking encapsulation it does still work, it's just naughty!
Attempting to set website to: "This is not a website, so an error is thrown!"
ERROR: Unable to set website to This is not a website, so an error is thrown!.
Required format is: http://www.example.com
```

4.4. Inheritance (Miras Alma, Kalıtım)

Miras (Inheritance) nesne tabanlı programlamada güçlü bir özelliktir. Varolan bir sınıfı çok az veya hiç değiştirmeden yeni bir sınıf tanımlamaya atıfta bulunur. Yeni sınıfa türetilmiş (veya çocuk) sınıf denir ve onu devraldığı sınıfa taban (veya ana sınıf) denir.

Miras kelimesini tanımı olarak, en basit haliyle birinden başka birine kalan varlık anlamına gelir. Programlama tarafındaki anlamı ise; bir class içersindeki elemanların (property, metod) farklı nesneler tarafından kullanılabilmesini sağlayan yapıdır.

Aşağıdaki Poligon olarak adlandırılan bir sınıf (Class) var.

```
class Polygon:
  def __init__(self, no_of_sides):
    self.n = no_of_sides
    self.sides = [0 for i in range(no_of_sides)]

  def inputSides(self):
    self.sides = [float(input("Enter side "+str(i+1)+" : ")) for i in range(self.n)]

  def dispSides(self):
    for i in range(self.n):
      print("Side",i+1,"is",self.sides[i])
```

Bu sınıf (Class), kenarların sayısı, “n” ve herbir kenarın büyüklüğünü bir liste olarak kenarları depolamak için veri özniteliklerine sahiptir.

InputSides () fonksiyonu her iki tarafın büyüklüğünü alır ve benzer şekilde, dispSides () bunları düzgün şekilde görüntüler.

Üçgen üç kenarlı bir çokgentir. Böylece, Poligon'dan miras alan Triangle adlı bir sınıf oluşturabiliriz. Bu, Poligon sınıfında mevcut olan tüm nitelikleri Üçgen'de hazır hale getirilir. Onları tekrar tanımlamamıza gerek yok (kodu yeniden kullanılabilirlik). Üçgen aşağıdaki gibi tanımlanır.

```
class Polygon:
  def __init__(self, no_of_sides):
    self.n = no_of_sides
    self.sides = [0 for i in range(no_of_sides)]

  def inputSides(self):
```

```
        self.sides = [float(input("Enter side "+str(i+1)+" : ")) for i in range(self.n)]

    def dispSides(self):
        for i in range(self.n):
            print("Side",i+1,"is",self.sides[i])

class Triangle(Polygon):
    def __init__(self):
        Polygon.__init__(self,3)

    def findArea(self):
        a, b, c = self.sides
        # calculate the semi-perimeter
        s = (a + b + c) / 2
        area = (s*(s-a)*(s-b)*(s-c)) ** 0.5
        print('The area of the triangle is %0.2f' %area)
```

Bununla birlikte, Class Triangle, üçgen alanını bulmak ve yazdırmak için yeni bir findArea () fonksiyonu içerir. İşte bir örnek çalışma.

```
t = Triangle()
t.inputSides()
t.dispSides()
t.findArea()
```

```
>>>
================== RESTART: C:/Users/ckk/Desktop/Polygon.py ==================
Enter side 1 : 4
Enter side 2 : 4
Enter side 3 : 4
Side 1 is 4.0
Side 2 is 4.0
Side 3 is 4.0
The area of the triangle is 6.93
```

C ++ da olduğu gibi Python'da da bir Class'dan birden fazla taban sınıf türetilebilir. Buna çoklu miras denir.

Birden çok devralmada ya da çoklu mirasda, tüm temel sınıfların özellikleri türetilen sınıfa devredilir. Çoklu kalıtım, tekli kalıtıma benzer.

Sınıfının doğrusallaştırılmasında sırayı bulmak için kullanılan kurallar dizisine Method Resolution Order (MRO) adı verilir. MRO, yerel öncelik sıralamasını önlemeli ve ayrıca monotonluk sağlamalıdır. Bu, bir sınıfın daima ana-babanın önünde çıkmasını ve birden çok ebeveyn olması durumunda sıranın temel sınıfların tuple'si ile aynı olmasını sağlar. Bir sınıfın MRO'su __mro__ nitelik veya mro () yöntemi olarak görülebilir. Birincisi bir liste döndürürken, birincisi bir tuple döndürür.

4.5. Abstract (Soyutlama)

Nesnelerin oluş tarzını ifade eden kavramlara soyut kavram denir. Bir restorana gittiğimiz zaman, bizimle ilgilenen garsona “bize biraz besin getirebilir misin?” diyebilir miyiz? Aklıselim bir insan isek cevabımız hayır olacaktır. Biraz besin yerine “zeytinyağlı enginar” dersek daha doğru olacaktır, çünkü besin soyut bir kavramdır, zeytinyağlı enginar ise besin sınıfına ait bir nesnedir.

Bu örneğe kod tarafından yaklaşalım; kendisinden nesne üretmeye izin vermeyen, sadece miras alınabilen sınıfları abstract class olarak tanımlayabiliriz. Abstract classın içerisine, gövdesi olan abstract bir metod yazdığınızda ' cannot declare a body because it is marked abstract hatası alırsınız, çünkü abstract metodların gövdesi olmaz. Yine bu metodu private olarak tanımladığınızda virtual or abstract members cannot be private hatası alırsınız, yani hata metninden de anlaşılacağı üzere abstract metodlar private olamaz. Abstract base class mantığındadır, sadece temel oluşturmak için kullanılır, yani bu sınıf inheritance alındıktan sonra kullanılabilir, tek başına işlevi yoktur, bu nedenle en karakteristik özelliği kendisinden nesne üretmesine izin vermez, abstract olmayan bir sınıfın içerisinde abstract bir elaman olamaz. Abstract bir sınıfı, miras alan bir sınıf, bu miras aldığı abstract sınıfı içerisindeki metodu override eder.

Örnek: abc modülü, türetilmiş bir sınıfın, o yöntem üzerinde özel bir @abstractmethod dekoratörü kullanarak belirli bir yöntemi uygulamasına izin verir. Soyut (Abstract) bir yöntem uygulanabilir ancak yalnızca türetilmiş bir sınıftan “super” ile çağrılabilir.

```
from abc import ABCMeta, abstractmethod
class Animal:
    __metaclass__ = ABCMeta

    @abstractmethod
    def say_something(self):
        return "I'm an animal!"

class Cat(Animal):
    def say_something(self):
        s = super(Cat, self).say_something()
        return "%s - %s" % (s, "Miauuu")

>>> c = Cat()
>>> c.say_something()
"I'm an animal! - Miauuu"
```

4.6. Polymorphism (Çok Biçimlilik)

Polymorphism; bir sınıftan miras alınan metodun, miras alındığı sınıftaki işlevi ile, miras alan sınıfta farklı işlevlerle çalıştırılabilme özelliğidir.

Bu işlemin nasıl olduğunu görmeden önce bilmemiz gereken bazı kavramlar var:
virtual: Metodun türetilmiş bir sınıfta geçersiz kılınmasını sağlamak için kullanılır.

override: Inheritance alınan bir sınıftaki metodu tekrar yazarak, base class'takini ezmiş, yoksaymış yani override etmiş oluruz. Bunun gerçekleşebilmesi için base class'taki metodun virtual olarak tanımlanması gerekir, yani buna izin vermesi gerekir ki virtual'un tanımı bu şekilde yapılsın.

overload: Türkçe karşılığı ile bir metodun aşırı yüklenmesidir, yani aynı isimde birden fazla metod farklı parametreler alabilir, ya da almayabilir.

base: türetilmiş sınıfın içerisinden, türetildiği sınıftaki base class'taki (public, internal, protected olarak tanımlanmış) elemanlara erişmek için kullanırız. Özetle Base class'ı referans alır. (abstract olarak tanımlanmış üyeler için kullanılamaz) base sınıfı bir üst sınıfı temsil eder. grandparent-parent-child gibi bir yapıyı düşünelim, grandparent'ta tanımlanış bir metoda child üzerinden this ile erişemeyiz. Sadece bir üst sınıfa, yani türetildiği sınıfa erişilebilir.

this: bu sözcüğün base'den farkı içinde bulunduğu nesneyi referans alır.

Gerçek hayatta, polimorfizmin pratik bir örneği verilebilir mi? "+" operatörü ile a + b = c yazılır. 2 + 2 = 4 olduğundan, bu polimorfizmdir.

```
class Animal:
    def __init__(self, name):    # Constructor of the class
        self.name = name
    def talk(self):              # Abstract method, defined by convention only
        raise NotImplementedError("Subclass must implement abstract method")

class Cat(Animal):
    def talk(self):
        return 'Meow!'

class Dog(Animal):
    def talk(self):
        return 'Woof! Woof!'
```

```
animals = [Cat('Missy'),
        Cat('Mr. Mistoffelees'),
        Dog('Lassie')]

for animal in animals:
   print (animal.name + ': ' + animal.talk())
```

Program çalıştırılırsa,
Missy: Meow!
Mr. Mistoffelees: Meow!
Lassie: Woof! Woof!

Dikkat edin: tüm hayvanlar "konuşmak", ancak farklı konuşuyorlar. Bu nedenle "konuşma" davranışı, hayvana bağlı olarak farklı şekilde gerçekleştiği anlamında polimorfiktir. Dolayısıyla soyut "hayvan" kavramı aslında "konuşmak" değil, belirli hayvanlara (köpekler ve kediler gibi) "konuşma" eyleminin somut bir şekilde uygulanmasına sahiptir.

Benzer şekilde, "add" işlemi birçok matematiksel varlıkta tanımlanır, ancak özel kurallara göre "eklediğiniz" belirli durumlarda: 1 + 1 = 2, (1 + 2i) + (2-9i) = (3-7i).

Polimorfik davranış, yaygın yöntemleri "soyut" bir seviyede belirlemenize ve bunları belirli durumlarda uygulamaya izin verir.

Örnek: Araba özellikleri Sportscar veya Truck ise daha fazla hesaba katmadan işlevselliği arayabiliriz.

```
class Car:
   def __init__(self, name):
      self.name = name

   def drive(self):
      raise NotImplementedError("Subclass must implement abstract method")

   def stop(self):
      raise NotImplementedError("Subclass must implement abstract method")

class Sportscar(Car):
   def drive(self):
      return 'Sportscar driving!'
```

```
    def stop(self):
        return 'Sportscar braking!'

class Truck(Car):
    def drive(self):
        return 'Truck driving slowly because heavily loaded.'

    def stop(self):
        return 'Truck braking!'


cars = [Truck('Bananatruck'),
        Truck('Orangetruck'),
        Sportscar('Z3')]

for car in cars:
    print (car.name + ': ' + car.drive())
```

Program çalıştırılırsa,
Bananatruck: Truck driving slowly because heavily loaded.
Orangetruck: Truck driving slowly because heavily loaded.
Z3: Sportscar driving!

4.7. Interface (Arayüz)

Interface (arayüz ya da arabirim olarak Türkçeye çevirilmiş) bir sınıfın temelde hangi elemanlardan oluşmak zorunda olduğunu belirleyen şablondur, inheritance kavramında temel bir sınıfımız vardı ve bu sınıfı miras alan derived class, base classtaki metodları, özellikleri vb. kullanabilirdi fakat bu zorunlu değildi. Interface ise bunu mecburi kılan bir yapıdır, yani içerisindeki tüm elemanlar implement edilmek zorundadır, aksi takdirde "class does not implement interface member InterfaceName" şeklinde bir hata ile karşılaşırsınız.

Bir class birden fazla sınıftan miras alamaz, fakat interface'lerin multi inhertitance desteği vardır, yani bir class birden fazla interface'den miras alabilir. interface içerisindeki metodlar default olarak abstract seviyesindedir ve içerisindeki tüm elemanlar public olarak tanımlıdır, interface'i de soyut bir sınıf olarak düşünebiliriz.

```
class Team:
    def __init__(self, members):
        self.__members = members

    def __len__(self):
        return len(self.__members)

    def __contains__(self, member):
        return member in self.__members

justice_league_fav = Team(["batman", "wonder woman", "flash"])

# Sized protocol
print(len(justice_league_fav))

# Container protocol
print("batman" in justice_league_fav)
print("superman" in justice_league_fav)
print("cyborg" not in justice_league_fav)
```

Program çalıştırılırsa,

```
3
True
False
True
```

5. Açıklamalı örnekler

```
>>> import math
>>> import cmath
```

```
# File: cmath-example-1.py
import cmath
print ("pi  =>", cmath.pi)
print ("sqrt(-1) =>", cmath.sqrt(-1))
```

```
# Python program to find the factorial of a number provided by the user.
# change the value for a different result
# num = 3
# uncomment to take input from the user
num = int(input("Enter a number: "))
factorial = 1
# check if the number is negative, positive or zero

if num < 0:
   print("Sorry, factorial does not exist for negative numbers")
elif num == 0:
   print("The factorial of 0 is 1")
else:
   for i in range(1,num + 1):
       factorial = factorial*i

   print("The factorial of",num,"is",factorial)
```

```
# Python Program to calculate the square root
num = float(input('Enter a number: '))
num_sqrt = num ** 0.5
print('The square root of %0.3f is %0.3f'%(num ,num_sqrt))
```

 # Solve the quadratic equation ax2 + bx + c = 0**

```
# import complex math module
import cmath
# To take coefficient input from the users
a = float(input('Enter a: '))
b = float(input('Enter b: '))
c = float(input('Enter c: '))

# calculate the discriminant
d = (b**2) - (4*a*c)

# find two solutions
x1 = (-b-cmath.sqrt(d))/(2*a)
x2 = (-b+cmath.sqrt(d))/(2*a)

print('The solution are {0} and {1}'.format(x1,x2))
```

 # Program to generate a random number between 0 and 49

```
# import the random module
import random
print(random.randint(0,49))
```

 # Python Program to Convert Decimal to Binary Using Recursion

```
def convertToBinary(n):
   """Function to print binary number
   for the input decimal using recursion"""
   if n > 1:
      convertToBinary(n//2)
  print(n % 2,end = '')

# decimal number
dec = int(input('Enter dec: '))
convertToBinary(dec)
```

```
# Program to add two matrices using nested loop

X = [[12,7,3],    [4 ,5,6],     [7 ,8,9]]

Y = [[5,8,1],    [6,7,3],     [4,5,9]]

result = [[0,0,0],         [0,0,0],         [0,0,0]]

# iterate through rows
for i in range(len(X)):
  # iterate through columns
  for j in range(len(X[0])):
     result[i][j] = X[i][j] + Y[i][j]

for r in result:
  print(r)
```

Örnek: 10 Elemanlı rastgele oluşturulan bir dizideki en büyük ve en küçük sayıyı bulan program.

```
from random import randint
sayilar=[]
for i in range(0,10):
   rand=randint(0, 9)
   sayilar.append(rand)
   print(rand)

minNumber = sayilar[0]
maxNumber = sayilar[0]

for i in range(0,10):
   if minNumber > sayilar[i]:
      minNumber = sayilar[i]
   if maxNumber < sayilar[i]:
      maxNumber = sayilar[i]

print("Dizideki En Büyük Değer    : > > >{0} ".format(maxNumber))
print("Dizideki En Küçük Değer    : > > >{0} ".format(minNumber))
```

Örnek. Verilen bir tarihin yılın kaçıncı günü olduğunu bulan program

```
def ArtıkYıl(yıl):
   artık=False
   if yıl%400==0 or (yıl%4==0 and yıl%100!=0): artık=True
   return artık
 def YılınGünü(Ay,Gün,Yıl):
   günler=[31,28,31,30,31,30,31,31,30,31,30,31]
   if ArtıkYıl(Yıl):
      günler[1]=29
   sıra=0
   for a in range(Ay-1):
      sıra+=günler[a]
   sıra+=Gün
   return sıra

print(YılınGünü(3,12,2017))
```

Örnek. İki matrisin çarpılması

```
def MatCarp(a,b):
   # a matrisinin boyutları m x n dir.
   # b matrisinin boyutları n x p dir.
   m=len(a)
   n=len(a[0])
   p=len(b[0])
   # Çarpımın sonucu olacak c matrisinin boyutları m x p dir.
   # c matrisini oluşturalım.
   c=[[0 for i in range(p)] for j in range(m)]
   # i: a ve c nin satır indisidir.
   # j: a nın sütun indisidir.
   # j: b nin satır indisidir.
   # k: b ve c nin sütun indisidir.
   for i in range(m):
      for k in range(p):
         for j in range(n):
            c[i][k]+=a[i][j]*b[j][k]
   return c
```

```
# Örnek olarak x ve y matrisleri tanımlayalım ve bu matrisleri çarpalım.
x=[[2,3],[5,7]]
y=[[1,5],[6,4]]
print(MatCarp(x,y))
```

Örnek. Saniye cinsinden verilen zamanı, saat, dakika ve saniyeye ayrıştıran program.

```
def ZamanAyrıştır(sec):
   saatler=sec//3600
   dakikalar=sec%3600//60
   saniyeler=sec%60
   return saatler,dakikalar,saniyeler
saniyezaman=436400  # Saniye olarak zaman
st, dk, sn=ZamanAyrıştır(saniyezaman)
print(saniyezaman, 'saniye=', st,'saat,', dk,'dakika ve',sn, 'saniye')
```

Örnek. İkinci dereceden bir denklemin kökleri

```
# a*x^2+b*x+c şeklindeki bir denklemin kökleri
def kökler(a,b,c):
   delta=b**2-4*a*c
   if delta>;0:
      x1=(-b+delta**0.5)/(2*a)
      x2=(-b-delta**0.5)/(2*a)
      return x1, x2
   return False
print(kökler(1,2,-15))
print(kökler(3,2,1))
```

Örnek. Verilen bir listenin maksimum, minimum ve ortalama değerlerini bulan program yazınız. Kütüphanede mevcut fonksiyonları kullanmayınız.

```
def maksd(birlist):
   m=birlist[0]
   for x in birlist:
      if m<x: m=x
   return m
def mind(birlist):
   m=birlist[0]
   for x in birlist:
      if m>x: m=x
   return m
def ortd(birlist):
   t=0
   for x in birlist:
      t=t+x
   return t/len(birlist)

listem=[-5,70,12,1]
print("Maksimum:",maksd(listem), "Minimum:",mind(listem), \
       "Ortalama:",ortd(listem))
```

Örnek. Uygulamalar

```
# Sözlük uygulamaları
# Bir sözlük oluşturalım
sözlük={'Adı':'Ali','Yaşı':20,'Sınıfı':3}

# Sözlük elemanlarını yazdıralım
for i in sözlük:
   print(i,':',sözlük[i])

# Sözlük elemanlarını items() fonksiyonunu kullanarak yazdıralım
for i,j in sözlük.items():
   print(i,':',j)

# Bir string değişkenin karakterlerinin tek tek yazdırılması
a='Merhaba Arkadaşlar'
for i in a:
   print(i)
```

```
# Değişkendeki karakterleri sıra numarası ile birlikte yazdıralım
for i,j in enumerate(a):
   print(i,':',j)

# Değişkendeki karakterleri bir listeye aktarıp, bu değişkeni kullanarak
# karakterleri sıra numaraları ile birlikte yazdıralım
b=[]
for i in a: b.append(i)
for i,j in enumerate(b):
   print(i,':',j)

# Bir string değişkenindeki bir karakteri veya karakter dizisini
# başka bir şeyle değiştirelim. Örneğin aşağıda boşluk karakterleri
# siliniyor.
a= “A B C D”
a.replace(' ','')

# Lambda fonksiyonu kullanarak bir sayının karesini hesaplayan
# bir fonksiyon yazalım
karesi=lambda x:x**2
print (karesi(2))

# bir x listesi icin, y=f(x) listesi oluşturalım. f(x)=sin(x) olsun
import math
x=[1,3,8,15]
f=lambda x: math.sin(x)
y=[]
for i in x:
   y.append(f(i))

# Bu işlemi map fonksiyonu aracılığıyla yapalım.
x=[1,3,8,15]
f=lambda x: math.sin(x)
# Burada f  fonsiyonunu
# def f(x): return math.sin(x)
# şeklinde de tanımlayabilirdik.
y=list(map(f,x))
```

Örnek. Sayı olarak verilen sınav notunu harf notuna dönüştüren fonksiyon

```
# Eğer puan tamsayı değilse yuvarlanmalıdır.
# Ancak kütüphanedeki round() fonksiyonu ile ilgili bir
# sorun var. round(50.5)=50 verir, halbuki biz bunu 51'e
# yuvarlarız. Bu 50 nin çift sayı olmasındandır.
# Bu sorunu yeni bir yuvarlama fonksiyonu ile çözelim.

# Aşağıdaki fonksiyon önce s sayısının tamsayı olup olmadığını kontrol
# ediyor. Daha sonra, eğer s'nin tamsayı bölümü çift ise ve ondalık
# rakam 5 ise round fonksiyonu ile aşağı yuvarlanan sayıyı bir artırıyoruz.

def Yuvarla(s):
    if int(s)!=s:
        p=str(s).split('.')
        s=round(s)
        if eval(p[0])%2==0 and eval(p[1])==5: s+=1
    return s

# Harf notlarının şöyle dağıldığını varsayalım.
# 0-30:FF 31-40:FD 41-50:DD 51-55:DC 56-60:CC
# 61-70:CB 71-80:BB 81-90:BA 91-100:AA

def HarfNot(puan):
    puan=Yuvarla(puan)
    Harfler= “FF FD DD DC CC CB BB BA AA”.split()
    Aralıklar=[0,31,41,51,56,61,71,81,91,101]
    harf='Geçersiz Not'
    for i in range(len(Harfler)):
        if puan>=Aralıklar[i] and puan<Aralıklar[i+1]:
            harf=Harfler[i]
    return harf

# Deneyelim
print(HarfNot(61))
```

Örnek. Karmaşık sayı için class tanımı

```
# Karmaşık sayıların ekranda uygun formatta yazılması için __repr__
# fonksiyonunu ve toplam ve çarpım işlemlerini yapan __add__
# ve __mul__ fonksiyonlarını yazalım.

class karmaşık:
   def __init__(self,reel,sanal):
      self.reel=reel
      self.sanal=sanal

   def __add__(self,sağ):
      sol=self
      reel=sol.reel+sağ.reel
      sanal=sol.sanal+sağ.sanal
      return karmaşık(reel,sanal)

   def __mul__(self,sağ):
      sol=self
      reel=sol.reel*sağ.reel-sol.sanal*sağ.sanal
      sanal=sol.reel*sağ.sanal+sol.sanal*sağ.reel
      return karmaşık(reel,sanal)

   def __repr__(self):
      s=self
      reel=str(s.reel)
      sanal=str(s.sanal)
      işaret='+'
      if s.sanal<0: işaret=''
      text='('+reel+işaret+sanal+'j)'
      return text
 # Deneyelim
x=karmaşık(3,2)
y=karmaşık(5,-4)
print(' x=',x,'\n y=',y,'\n x+y=',x+y,'\n x*y=',x*y,'\n')

# Python içinde mevcut complex class'ını kullanarak,
x=complex(3,2)
y=complex(5,-4)
print(' x=',x,'\n y=',y,'\n x+y=',x+y,'\n x*y=',x*y)
```

Örnek. Şekil ve kesit class örneği

```
from math import pi
 class Section:
       SectionList=[]

   def __repr__(self):
      s=self
      A=str(round(s.A,2))
      I=str(round(s.I,2))
      return s.shape+' A:'+A+' I:'+I

   def printlist(self):
      for each in self.SectionList:
         print(each)

class Circle(Section):  #Circle class inherits from Section class
   # A: Area, I: Moment of inertia
   def __init__(self,r):
      s=self
      s.A=pi*r*r
      s.I=pi*r**4/2
      s.shape="Circle"
      s.SectionList.append(s)

class Rectangle(Section): #Rectangle class inherits from Section class
   # A: Area, I: Moment of inertia
   def __init__(self,b,h):
      s=self
      s.A=b*h
      s.I=b*h**3/12
      s.shape="Rectangle"
      s.SectionList.append(s)

sec1=Circle(2)
sec2=Rectangle(2,4)
sec1.printlist()
```

HOW TO WRITE AN EXCEL FILE IN PYTHON

```
>>> import openpyxl
>>> wb = openpyxl.load_workbook('example.xlsx')
>>> sheet = wb.active
>>> sheet.title = 'Spam Spam Spam'
>>> wb.save('example_copy.xlsx')
```

```
import xlrd
#--------------------------------------------------------------------
def open_file(path):
    """
    Open and read an Excel file
    """
    book = xlrd.open_workbook(path)
    # print number of sheets
    print book.nsheets
    # print sheet names
    print book.sheet_names()
    # get the first worksheet
    first_sheet = book.sheet_by_index(0)
    # read a row
    print first_sheet.row_values(0)
    # read a cell
    cell = first_sheet.cell(0,0)
    print cell
    print cell.value
    # read a row slice
    print first_sheet.row_slice(rowx=0,
                                start_colx=0,
                                end_colx=2)
#--------------------------------------------------------------------
if __name__ == "__main__":
    path = "test.xls"
    open_file(path)
```

Example: High quality plotting library.

```
#!/usr/bin/env python
import numpy as np
import matplotlib.mlab as mlab
import matplotlib.pyplot as plt
mu, sigma = 100, 15
x = mu + sigma*np.random.randn(10000)
# the histogram of the data
n, bins, patches = plt.hist(x, 50, normed=1, facecolor='green',
alpha=0.75)
# add a 'best fit' line
y = mlab.normpdf( bins, mu, sigma)
l = plt.plot(bins, y, 'r--', linewidth=1)
plt.xlabel('Smarts')
plt.ylabel('Probability')
plt.title(r'$\mathrm{Histogram\ of\ IQ:}\ \mu=100,\ \sigma=15$')
plt.axis([40, 160, 0, 0.03])
plt.grid(True)
plt.show()
```

6. Çizim örnekleri

Örnek:

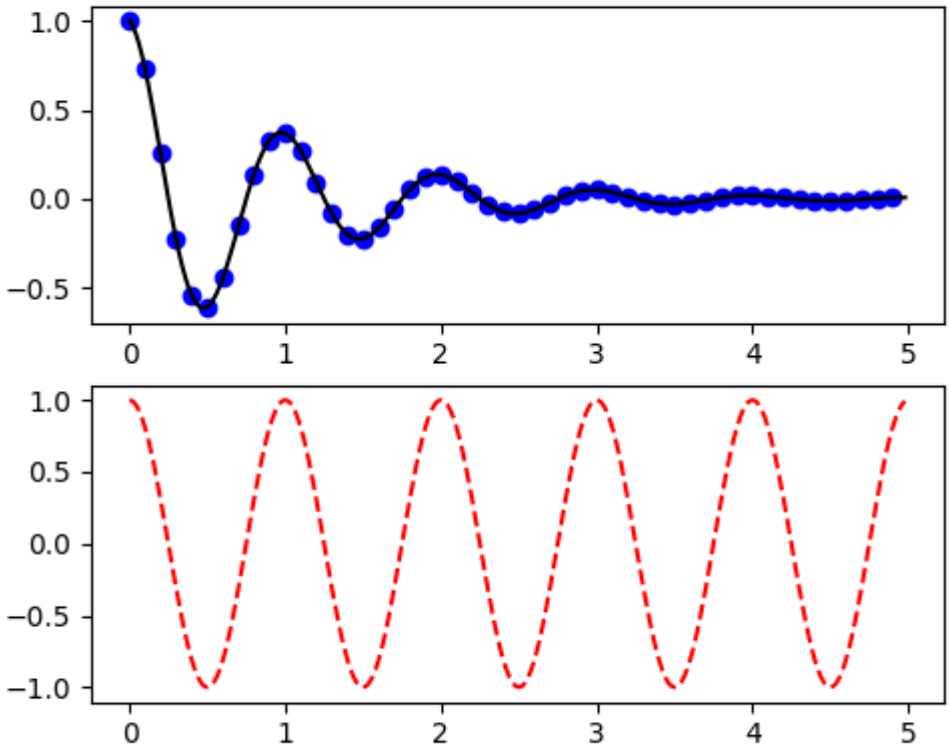
```
import matplotlib.pyplot as plt
plt.plot([1,2,3,4], [1,4,9,16], 'ro')
plt.axis([0, 6, 0, 20])
plt.show()
```

Örnek:

```
import numpy as np
import matplotlib.pyplot as plt
# evenly sampled time at 200ms intervals
t = np.arange(0., 5., 0.2)
# red dashes, blue squares and green triangles
plt.plot(t, t, 'r--', t, t**2, 'bs', t, t**3, 'g^')
plt.show()
```

Örnek:

```
import numpy as np
import matplotlib.pyplot as plt
def f(t):
    return np.exp(-t) * np.cos(2*np.pi*t)

t1 = np.arange(0.0, 5.0, 0.1)
t2 = np.arange(0.0, 5.0, 0.02)

plt.figure(1)
plt.subplot(211)
plt.plot(t1, f(t1), 'bo', t2, f(t2), 'k')

plt.subplot(212)
plt.plot(t2, np.cos(2*np.pi*t2), 'r--')
plt.show()
```



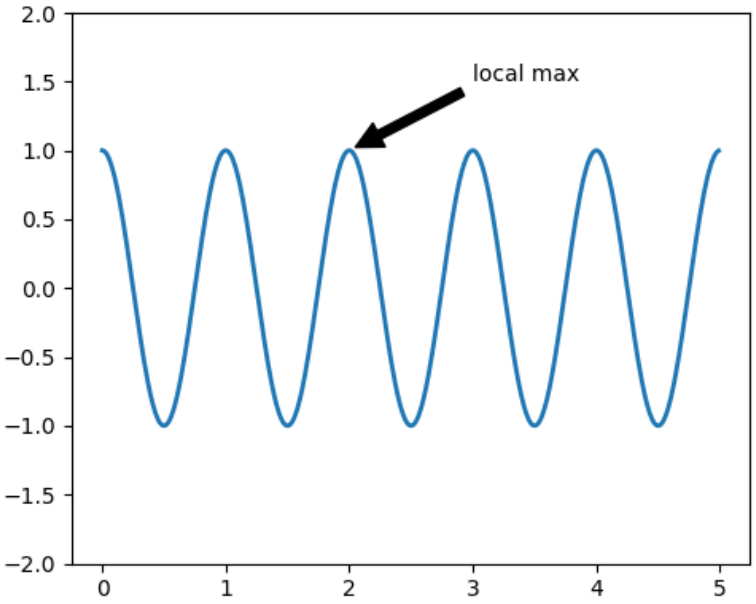
Örnek: Working with text

```
import numpy as np
import matplotlib.pyplot as plt

# Fixing random state for reproducibility
np.random.seed(19680801)

mu, sigma = 100, 15
x = mu + sigma * np.random.randn(10000)

# the histogram of the data
n, bins, patches = plt.hist(x, 50, normed=1, facecolor='g', alpha=0.75)

plt.xlabel('Smarts')
plt.ylabel('Probability')
plt.title('Histogram of IQ')
plt.text(60, .025, r'$\mu=100,\ \sigma=15$')
plt.axis([40, 160, 0, 0.03])
```

```
plt.grid(True)
plt.show()

t = plt.xlabel('my data', fontsize=14, color='red')
```

Örnek: Annotating text, Text rendering With LaTeX.

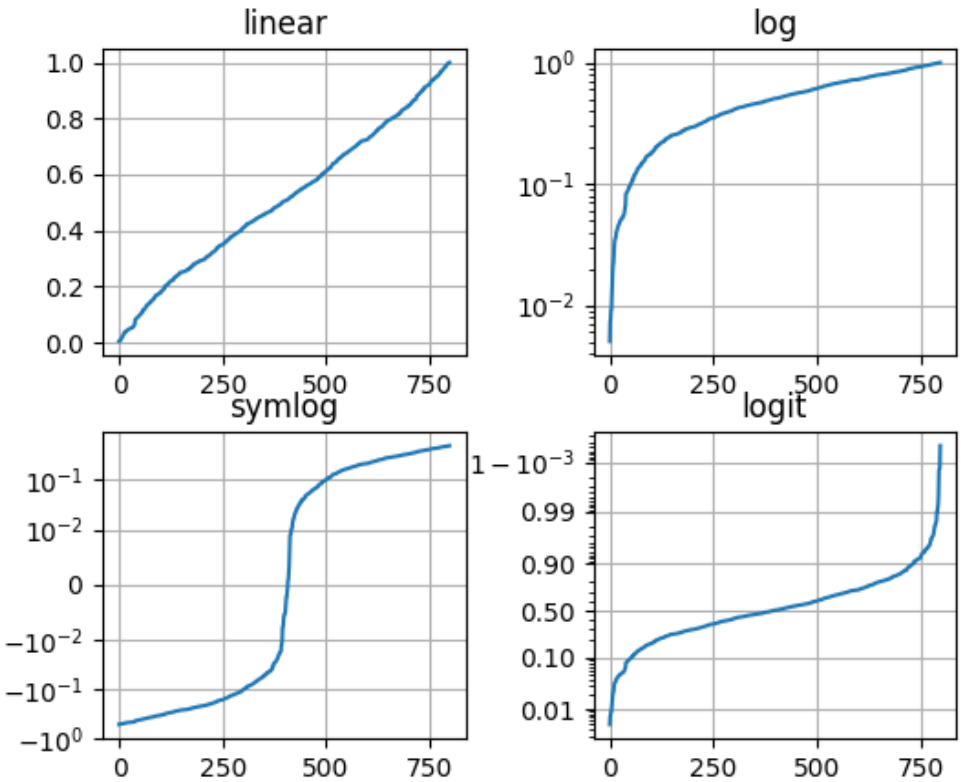
```
import numpy as np
import matplotlib.pyplot as plt

ax = plt.subplot(111)

t = np.arange(0.0, 5.0, 0.01)
s = np.cos(2*np.pi*t)
line, = plt.plot(t, s, lw=2)

plt.annotate('local max', xy=(2, 1), xytext=(3, 1.5),
        arrowprops=dict(facecolor='black', shrink=0.05),
        )

plt.ylim(-2,2)
plt.show()
```



Example: Logarithmic and other nonlinear axes

```
import numpy as np
import matplotlib.pyplot as plt

from matplotlib.ticker import NullFormatter  # useful for `logit` scale

# Fixing random state for reproducibility
np.random.seed(19680801)

# make up some data in the interval ]0, 1[
y = np.random.normal(loc=0.5, scale=0.4, size=1000)
y = y[(y > 0) & (y < 1)]
y.sort()
x = np.arange(len(y))

# plot with various axes scales
plt.figure(1)

# linear
plt.subplot(221)
plt.plot(x, y)
plt.yscale('linear')
plt.title('linear')
plt.grid(True)

# log
plt.subplot(222)
plt.plot(x, y)
plt.yscale('log')
plt.title('log')
plt.grid(True)

# symmetric log
plt.subplot(223)
plt.plot(x, y - y.mean())
plt.yscale('symlog', linthreshy=0.01)
plt.title('symlog')
plt.grid(True)
```

```
# logit
plt.subplot(224)
plt.plot(x, y)
plt.yscale('logit')
plt.title('logit')
plt.grid(True)
# Format the minor tick labels of the y-axis into empty strings with
# `NullFormatter`, to avoid cumbering the axis with too many labels.
plt.gca().yaxis.set_minor_formatter(NullFormatter())
# Adjust the subplot layout, because the logit one may take more space
# than usual, due to y-tick labels like "1 - 10^{-3}"
plt.subplots_adjust(top=0.92, bottom=0.08, left=0.10, right=0.95, hspace=0.25,
                    wspace=0.35)

plt.show()
```



Örnek: Working with multiple figure windows and subplots

```
import matplotlib.pyplot as plt
import numpy as np

t = np.arange(0.0, 2.0, 0.01)
s1 = np.sin(2*np.pi*t)
s2 = np.sin(4*np.pi*t)

plt.figure(1)
plt.subplot(211)
plt.plot(t, s1)
plt.subplot(212)
plt.plot(t, 2*s1)

plt.figure(2)
plt.plot(t, s2)

# now switch back to figure 1 and make some changes
plt.figure(1)
plt.subplot(211)
plt.plot(t, s2, 's')
ax = plt.gca()
ax.set_xticklabels([])

plt.show()
```

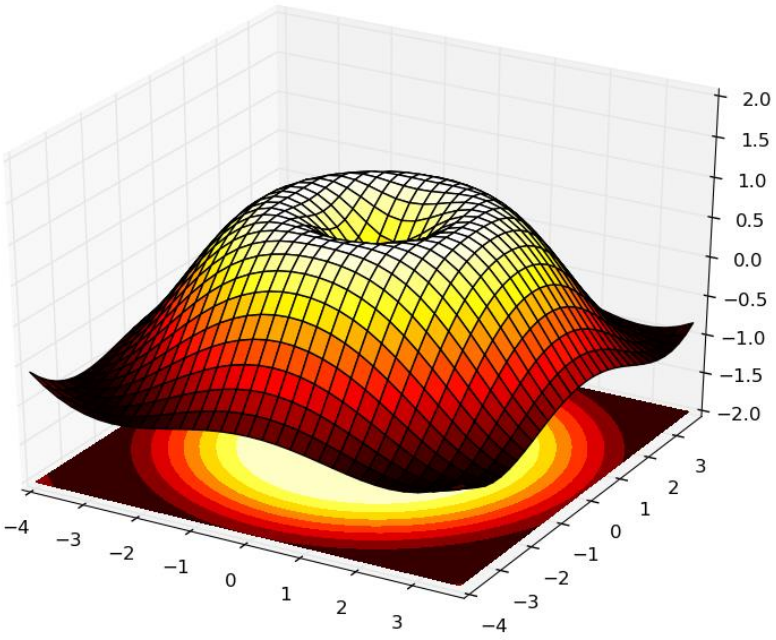
Örnek: Plotting with default settings

```
import numpy as np
import matplotlib.pyplot as plt

X = np.linspace(-np.pi, np.pi, 256, endpoint=True)
C, S = np.cos(X), np.sin(X)

plt.plot(X, C)
plt.plot(X, S)

plt.show()
```

Example: Instantiating defaults

```
import numpy as np
import matplotlib.pyplot as plt

# Create a figure of size 8x6 inches, 80 dots per inch
plt.figure(figsize=(8, 6), dpi=80)

# Create a new subplot from a grid of 1x1
plt.subplot(1, 1, 1)

X = np.linspace(-np.pi, np.pi, 256, endpoint=True)
C, S = np.cos(X), np.sin(X)

# Plot cosine with a blue continuous line of width 1 (pixels)
plt.plot(X, C, color="blue", linewidth=1.0, linestyle="-")

# Plot sine with a green continuous line of width 1 (pixels)
plt.plot(X, S, color="green", linewidth=1.0, linestyle="-")

# Set x limits
plt.xlim(-4.0, 4.0)

# Set x ticks
plt.xticks(np.linspace(-4, 4, 9, endpoint=True))
```

```
# Set y limits
plt.ylim(-1.0, 1.0)

# Set y ticks
plt.yticks(np.linspace(-1, 1, 5, endpoint=True))

# Save figure using 72 dots per inch
# plt.savefig("exercice_2.png", dpi=72)

# Show result on screen
plt.show()

# Changing colors and line widths
...
plt.figure(figsize=(10, 6), dpi=80)
plt.plot(X, C, color="blue", linewidth=2.5, linestyle="-")
plt.plot(X, S, color="red",  linewidth=2.5, linestyle="-")
...

Örnek: 3D Plots
from mpl_toolkits.mplot3d import Axes3D

fig = plt.figure()
ax = Axes3D(fig)
X = np.arange(-4, 4, 0.25)
Y = np.arange(-4, 4, 0.25)
X, Y = np.meshgrid(X, Y)
R = np.sqrt(X**2 + Y**2)
Z = np.sin(R)

ax.plot_surface(X, Y, Z, rstride=1, cstride=1, cmap='hot')
```


2.0
1.5
1.0
0.5
0.0
−0.5
−1.0
−1.5
−2.0
−4
−3
−2
−1
0
1
2
3
−4
−3
−2
−1
0
1
2
3

Kaynaklar

1- "A Practical Introduction to Python Programming", Brian Heinold, Department of Mathematics and Computer Science Mount St. Mary's University.
2- "Learning to Program with Python", Richard L. Halterman
3- "Introduction to Python", Heavily based on presentations by Matt Huenerfauth (Penn State), Guido van Rossum (Google), Richard P. Muller (Caltech)
4- "Python ile Programlamaya Giriş", Cihangir Beşiktaş
5- "Python Programming Language", www.tutorialspoint.com
6- https://en.wikibooks.org/wiki/A_Beginner%27s_Python_Tutorial/Classes
7- http://introtopython.org/classes.html
8- https://matplotlib.org/2.0.2/users/pyplot_tutorial.html
9- Downloads: http://www.python.org
10- Documentation: http://www.python.org/doc/
11- Free book: http://www.diveintopython.org
12- http://docs.python.org/modindex.html
13- Downloads: http://numpy.scipy.org/
14- Tutorial: http://www.scipy.org/Tentative_NumPy_Tutorial
15- http://matplotlib.sourceforge.net/
16- http://www.stsci.edu/resources/software_hardware/pyfits
17- Sage: http://www.sagemath.org/
18- http://code.google.com/p/python-sao/
19- staff.ltam.lu/feljc/software/python/python_basics.pdf
20- legacy.python.org/doc/essays/ppt/lwnyc2002/intro22.ppt
21- www.fh.huji.ac.il/~goldmosh/PythonTutorialFeb152012.ppt
22- www.cs.brandeis.edu/~cs134/Python_tutorial.ppt
23- www.enderunix.org/docs/python_giris.pdf
24- http://tdc-www.harvard.edu/Python.pdf
25- ocw.metu.edu.tr/pluginfile.php/231/mod_resource/content/0/konular.pdf
26- https://stackoverflow.com/questions/3724110/practical-example-of-polymorphism
27-