Birden fazla atama

→ yeni atama yeni temsil

```
1  bruce = 5
2  print bruce,
3  bruce = 7
4  print bruce
```

→ ekran çıktısı: 5, 7

değişken: kutu/kova benzetmesi

→ şimdiye kadar değişkenlere kova benzetmesinde bulunduk

```
1  bruce = 5
2  bruce = 7
```

→ ilk satırda kovada 5 değeri vardı, ardından 7 konuldu

→ Python diğer dillerden farklıdır

→ değişkenleri kova yerine etiket veya “sticky note”

→ bruce=5: nesneye (burada 5) bruce etiketini ver

→ bruce=7: nesneye (burada 7) bruce etiketini ver

→ Python’da her şey nesnedir: 1, [1, 2], abs(), ...

id(): nesnelerin bellekteki yeri

→ id(): nesnelerin bellekteki adresini söyler

```
>>> help(id)
Help on built-in function id in module __builtin__:

id(...)
    id(object) -> integer

    Return the identity of an object.  This is guaranteed to b
    simultaneously existing objects.  (Hint: it's the object's
```

→ örnek

```
>>> id(5)
159747376
>>> id(7)
159747352
```

değişken/etiket - kimliği

→ değişkenler nerede saklanır?

```
1  >>> bruce=5
2  >>> id(bruce)
3  159747376
4  >>> bruce=7
5  >>> id(bruce)
6  159747352
```

→ sayılar birbirine benzer fakat

→ ilk id(bruce) ile id(5) aynı sayıyı

→ aynı bellek adresini gösteriyor

→ Python'da değişken tanımını hatırlatırsak

→ nesneye (ör. 5) etiket (ör. bruce) ver

değişkenleri güncelleme

→ değer güncellemede olan nedir?

```
>>> x = 1
>>> x, id(x), id(1)
(1, 159747424, 159747424)
>>> x = x + 1
>>> x, id(x), id(1)
(2, 159747412, 159747424)
>>> x, id(x), id(1), id(2)
(2, 159747412, 159747424, 159747412)
```

değişkenleri güncelleme

→ kodları filtrele

```
>>> x = 1
>>> x
1
>>> x = x + 1
>>> x
2
```

→ 1 nesnesine x etiketini ata

→ x etiketli nesneyle 1 nesnesini topla

→ help(x) *ve* help(1) *ne üretiyor?*

→ *her şey nesne!!!*

→ *tamsayı nesneleri üzerinde* + *işleci toplama yapar*

→ 2 nesnesine x etiketini ata

ilklendirme

→ değişkenin ilkin değeri (eski değeri)?

```
>>> toplam = toplam + 10
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'toplam' is not defined
```

→ siz toplayın

→ soracaksınız toplam ne ki?

→ üzerine 10 ekleyeyim

while cümlesi

→ yineleme amacıyla kullanılan ilk yapı

```
1  def countdown(n):
2      while n > 0:
3          print n
4          n = n-1
5      print "Yokol!"
```

→ koşul: True veya False

→ koşul doğru olduğu müddetçe while cümlelerini söyle

→ yanlış olunca çık

sonsuz döngü

→ döngü, sonsuz döngü

```
1  def sequence(n):
2      while n != 1:
3          print n,
4          if n % 2 == 0:          # n çifttir
5              n = n / 2
6          else:                   # n tektir
7              n = n * 3 + 1
```

→ döngü hangi şartlarda kesilir?

basamakları sayma

→ basamakları sayalım

```
1  def num_digits(n):
2      count = 0
3      while n:
4          count = count + 1
5          n = n / 10
6      return count
```

→ doctest'lerimiz

```
1  >>> num_digits(710)
2  3
3  >>> num_digits(12)
4  2
5  >>> num_digits(12345)
6  5
```

→ pastebin: http://bpaste.net/show/5073/, veya PB=5073

→ bu arada bpython'u denediniz mi?

koşul eklentisi

→ sadece basamak değeri 0 ve 5 olanları saymak isteseydik (PB=5074)

```
>>> def num_zero_and_five_digits(n):
...     count = 0
...     while n:
...         digit = n % 10
...         if digit == 0 or digit == 5:
...             count = count + 1
...         n = n / 10
...     return count
...
>>> num_zero_and_five_digits(1055030250)
7
>>> num_zero_and_five_digits(1200)
2
>>> num_zero_and_five_digits(120051)
3
```

sıra sizde

→ basamak değeri en büyük hangisi

→ bu sayının basamaklarıyla elde edilecek en büyük/küçük sayı kaçtır?

kısaltılmış atama

→ değişkeni arttırmayla sık karşılaşırız (PB=5075)

```
>>> count = 0
>>> count += 1
>>> count
1
>>> count = count + 1
>>> count
2
```

→ diğerleri (PB=5076)

```
>>> n += 5
>>> n = 5
>>> n += 3
>>> n
8
>>> n * = 2
>>> n
16
>>> n %= 3
>>> n
1
```

tablolar

→ basit (PB=5077)

```
>>> x = 1
>>> while x < 13:
...     print x, '\t', 2**x
...     x += 1
```

→ çıktı

```
1   1       2
2   2       4
3   3       8
4   4       16
5   5       32
6   6       64
7   7       128
8   8       256
9   9       512
10  10      1024
11  11      2048
12  12      4096
```

→ burada t: kaçış serisi (escape sequences)

iki boyutlu tablolar

→ 2B tablo (PB=5078)

```
>>> i = 1
>>> while i <= 6:
...     print 2*i, '   ',
...     i += 1
...
...
2     4     6     8     10     12
```

→ print cümlesinin sonundaki virgül–, dikkat

sarma (encapsulation) ve genelleştirme

sarma: (kabaca) bir kod parçasını işlev içerisine koymadır

→ PB:5079

```
1  def print_multiples(n):
2      i = 1
3      while i <= 6:
4          print n*i, '\t',
5          i += 1
6      print
```

→ PB:5078'deki 2 değerinin, n ile değiştirilmesi genelleştirmedir

→ çıktılar

```
1  >>> print_multiples(3)
2  3       6       9       12
3  >>> print_multiples(4)
4  4       8       12      16
```

=

→ sütunda 6'ya kadar yazdırılıyor

→ genelleştirin

çarpım tablosu

→ PB:5080

```
>>> i = 1
>>> while i <= 6:
...     print_multiples(i)
...     i += 1
...
1       2       3       4       5       6
2       4       6       8       10      12
3       6       9       12      15      18
4       8       12      16      20      24
5       10      15      20      25      30
6       12      18      24      30      36
```

daha fazla sarma

→ PB:5080'ni sarmayabiliriz (PB:5081)

```
>>> def print_mult_table():
...     i = 1
...     while i <= 6:
...         print_multiples(i)
...         i += 1
...
>>> print_mult_table()
1       2       3       4       5       6
2       4       6       8       10      12
3       6       9       12      15      18
4       8       12      16      20      24
5       10      15      20      25      30
6       12      18      24      30      36
```

yerel değişkenler

- → i değişkeni hem print_multiples hem de print_mult_table işlevinde
- → aynı i değil
- → çünkü işlevde yaratılan değişken ona yereldir
- → dolayısıyla, print_multiples:i ve print_mult_table:i diyebiliriz
- → işleve yerel dışarıdan erişmezsin!

daha fazla genelleştirme

→ satır yüksekliğini değiştirmek istesek (PB:5082)

```
>>> def print_mult_table(high):
...     i = 1
...     while i <= high:
...         print_multiples(i)
...         i += 1
...
```

→ çıktı

```
>>> print_mult_table(3)
1       2       3       4       5
2       4       6       8       10
3       6       9       12      15
>>> print_mult_table(5)
1       2       3       4       5
2       4       6       8       10
3       6       9       12      15
4       8       12      16      20
5       10      15      20      25
```

daha fazla genelleştirme

→ high parametresini print_multiples işlevine de aktaralım (PB:5083)

```
>>> def print_multiples(n, high):
...     i = 1
...     while i <= high:
...         print n*i, '\t',
...         i += 1
...     print
```

→ print_mult_table değişmedi

```
>>> def print_mult_table(high):
...     i = 1
...     while i <= high:
...         print_multiples(i, high)
...         i += 1
```

→ çıktı

```
>>> print_mult_table(3)
1       2       3
2       4       6
3       6       9
>>> print_mult_table(5)
1       2       3       4       5
2       4       6       8       10
3       6       9       12      15
4       8       12      16      20
5       10      15      20      25
```

daha fazla genelleştirme

→ print_mult_table işlevindeki print_multiples çağrısında

```
1  print_multiples(i, high)
```

→ yerine

```
1  print_multiples(i, i)
```

→ dersek, ekran çıktısı

```
1  >>> print_mult_table(3)
2  1
3  2       4
4  3       6       9
5  >>> print_mult_table(5)
6  1
7  2       4
8  3       6       9
9  4       8       12      16
10 5       10      15      20      25
```

neden işlev

1. # Bir cümle dizisine bir isim vermeniz programınızın okunabilirliğini arttıracak, hata ayıklamayı kolaylaştıracaktır.
2. Büyük bir programı fonksiyonlara parçalamanız, programda parçaları birbirinden ayırmanızı sağlayacaktır. Böylece izole bir şekilde hataları ayıklayabilecek, bu farklı parçaların bir bütün olarak davranmasını sağlayabileceksiniz.
3. Fonksiyonlar yinelemenin kullanımını kolaylaştırır.
4. İyi tasarlanmış fonksiyonlar, yazılıp iyi bir şekilde hatalardan arındırıldıktan sonra tekrar kullanılabildiği için, bir çok program için yararlıdır.

newton karekök

→ newton yöntemi(PB:5084)

```
>>> def sqrt(n):
...     approx = n/2.0
...     better = (approx + n/approx)/2.0
...     while better != approx:
...         approx = better
...         better = (approx + n/approx)/2.0
...     return approx
...
>>> sqrt(25)
5.0
>>> sqrt(24)
4.898979485566356
```

fibonacci serisi

→ fibonacci sayıları

```
>>> a, b = 0, 1
>>> while b < 1000:
...     print b,
...     a, b = b, a+b
...
1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987
```

sıra sizde

→ sqrt işlevindeki better değişkenini yineleme numarasıyla ekranda gösterin

→ üçgensel sayılar

```
1  def print_triangular_numbers(n):
2      """\
3          >>> print_triangular_numbers(5)
4          1        1
5          2        3
6          3        6
7          4        10
8          5        15
9      """
```