PYTHON PROGRAMMING

Lists, Dictionaries and Tuples

#6

Serdar ARITAN

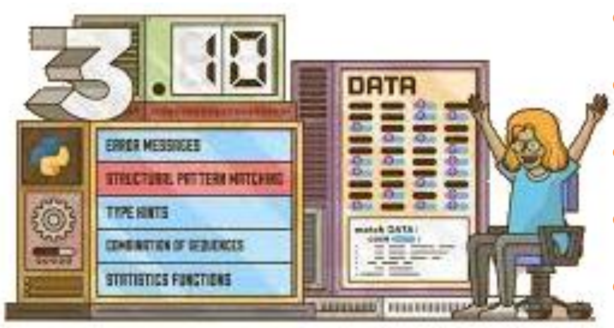
Biomechanics Research Group,
Faculty of Sports Sciences, and
Department of Computer Graphics
Hacettepe University, Ankara, Turkey

PEP 636 -- Structural Pattern Matching

Python Enhancement Proposal

A `match` statement takes an expression and compares it to successive patterns given as one or more `case` blocks. This is superficially similar to a switch statement in C, Java or JavaScript (and many other languages), but much more powerful. Python 3.10 introduced structural pattern-matching syntax in python.

```python
def http_error(status):
    match status:
        case 400:
            return "Bad request"
        case 401:
            return "Unauthorized"
        case 403:
            return "Forbidden"
        case 404:
            return "Not found"
        case 418:
            return "I'm a teapot"
        case _:
            return "Something else"
```



PEP 636 -- Structural Pattern Matching

You can combine several literals in a single pattern using | ("or"):

```
case 401 | 403 | 404:
        return "Not allowed"
```

Patterns can look like unpacking assignments, and can be used to bind variables:

```
# point is an (x, y) tuple
match point:
    case (0, 0):
        print("Origin")
    case (0, y):
        print(f"Y={y}")
    case (x, 0):
        print(f"X={x}")
    case (x, y):
        print(f"X={x}, Y={y}")
    case _:
        raise ValueError("Not a point")
```

PEP 636 -- Structural Pattern Matching

```
def switch(day):
    if day == 1:
        return "Sunday"
    elif day ==2:
        return "Monday"
    elif day == 3:
        return "Tuesday"
    elif day ==4:
        return "Wednesday"
    elif day == 5:
        return "Thursday"
    elif day ==6:
        return "Friday"
    elif day == 7:
        return "Saturday"
    else:
        return "Enter a Valid Day Number"

print(switch(1))   #Sunday
print(switch(4))   #Wednesday
print(switch(7))   #Saturday
print(switch(10))  #Enter a Valid Day Number
```

```
def weekday (day):
    match day:
        case 1:
            return "Sunday"
        case 2:
            return "Monday"
        case 3:
            return "Tuesday"
        case 4:
            return "Wednesday"
        case 5:
            return "Thursday"
        case 6:
            return "Friday"
        case 7:
            return "Saturday"
        case _:
            return "Enter a Valid Day Number"

print(weekday(1))   #Sunday
print(weekday(4))   #Wednesday
print(weekday(7))   #Saturday
print(weekday(10))  #Enter a Valid Day Number
```

PEP 636 -- Structural Pattern Matching

```
total = 200

extra_toppings_1 = 'pepperoni'
extra_toppings_2 = 'onions'

match [extra_toppings_1, extra_toppings_2]:
    case ['pepperoni', 'mushrooms']:
        extra = 79
    case ['pepperoni', 'onions']:
        extra = 49
    case ['pepperoni', 'bacon']:
        extra = 99
    case ['pepperoni', 'extra cheese', 'black olives']:
        extra = 149

print("Your total bill is:", total + extra)

Your total bill is: 249
```

```
key = 'A'

match key:
    case 'a' | 'A':
        print("Move Left")
    case 'w' | 'W':
        print('Move Forward')
    case 'd' | 'D':
        print("Move right")
    case 's' | 'S':
        print("Move Backward")
```

https://cscircles.cemc.uwaterloo.ca/visualize

https://cscircles.cemc.uwaterloo.ca/visualize

Not syncing

Computer Science Circles Homepage | Contact Us

Write your Python 3 code here:

```python
my_list = ['bco', '601','Python','Programlama']
your_list1 = my_list
your_list2 = my_list[:]

```

Visualize Execution

▸ Enter optional text input for the program to read with `input()`:

▸ Examples

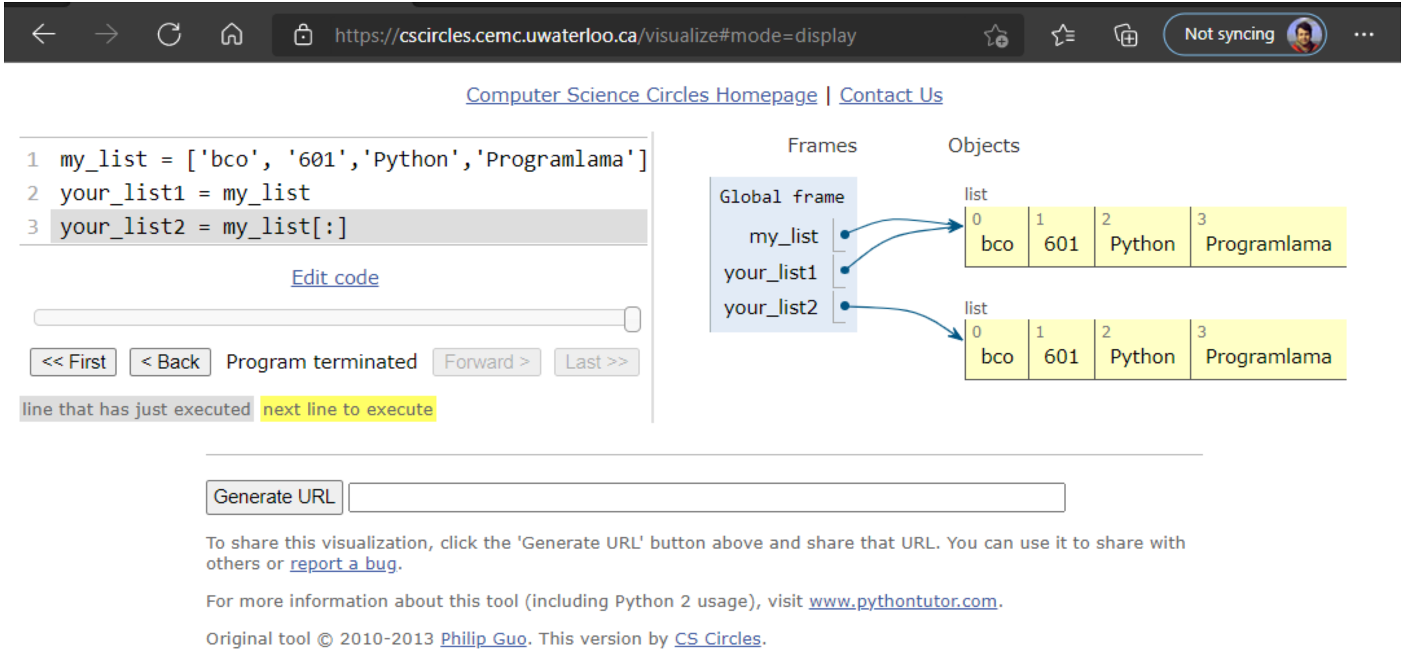
Generate URL

To share this visualization, click the 'Generate URL' button above and share that URL. You can use it to share with others or report a bug.

For more information about this tool (including Python 2 usage), visit www.pythontutor.com.

Original tool © 2010-2013 Philip Guo. This version by CS Circles.

https://cscircles.cemc.uwaterloo.ca/visualize



Lists

Like a string, a list is a sequence of values. In a string, the values are characters; in a list, they can be any type. The values in a list are called elements or sometimes items.

Basic properties:

Lists are contained in square brackets [ ]

Lists can contain numbers, strings, nested sublists, or nothing

Examples:

```
L1 = [0,1,2,3]
L2 = ['zero', 'one']
L3 = [0,1,[2,3],'three',[‘four’, ‘one’]]
L4 = []
```

Lists

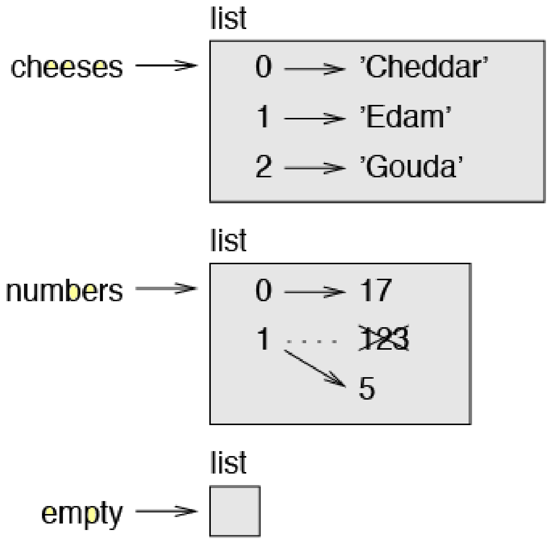
List indexing works just like string indexing
Lists are mutable: individual elements can be reassigned in place. Moreover, they can grow and shrink in place

```
Example:
>>> L1 = [0, 1, 2, 3]
>>> L1[0] = 4
>>> L1[0]
4
```

Lists

Lists Are <u>Mutable</u> : Unlike strings, lists are <u>mutable</u>. When the bracket operator appears on the left side of an assignment, it identifies the element of the list that will be assigned.

```
>>> cheeses = ['Cheddar', 'Edam', 'Gouda']
>>> numbers = [17, 123]
>>> empty = []
>>> print (cheeses, numbers, empty)
['Cheddar', 'Edam', 'Gouda'] [17, 123] []
>>> numbers[1] = 5
>>> print (numbers)
[17, 5]
```



Lists

```
hangi_ay = int(input("Hangi Ay (1-12)? "))

aylar = ['Ocak','Subat','Mart','Nisan','Mayis','Haziran',\
        'Temmuz','Agustos','Eylul','Ekim','Kasim','Aralik']

if 1 <= hangi_ay <= 12:
    print("Sectiğiniz Ay ", aylar[hangi_ay - 1])
```

Lists

```
>>> L = [1, 2, 3]
>>> for element in L:
...     print(element)
...
1
2
3
>>> L.append("I am a string.")
>>> for element in L:
...      print(element)
...
1
2
3
I am a string.
```

Lists

Some basic operations on lists:

Indexing: `L1[i], L2[i][j]` **Slicing**: `L3[i:j]`

Concatenation:

```
>>> L1 = [0,1,2]; L2 = [3,4,5]
>>> L1+L2
[0,1,2,3,4,5]
```

Repetition:

```
>>> L1*3
[0,1,2,0,1,2,0,1,2]
```

Appending:

```
>>> L1.append(3)
[0,1,2,3]
```

Sorting:

```
>>> L3 = [2, 1, 4, 3]
>>> L3.sort()
[1,2,3,4]
```

Lists

Reversal:

```
>>> L4 = [4,3,2,1]
>>> L4.reverse()
>>> L4
[1,2,3,4]
```

Append, Pop and Insert:

```
>>> L4.append(5)          # [1,2,3,4,5]
>>> L4.pop()              # [1,2,3,4]->5
>>> L4.insert(0, 42)      # [42,0,1,2,3,4]
>>> L4.pop(0)             # [1,2,3,4]
>>> L4.pop(2)             # [1,2,4]->3
```

Lists

```
# remove and del
names = ["Tommy", "Bill", "Janet",
"Bill", "Stacy"]
# Remove this value
names.remove("Bill")
print(names)
# Delete all except first two elements
del names[2:]
print(names)
# Delete all except last element
del names[:1]
print(names)
```

```
list = ["dot", "4.5"]
# Insert at index 1.
list.insert(1, "net")
print(list)
['dot', 'net', '4.5']
```

```
list = []
list.append(1)
list.append(2)
list.append(6)
list.append(3)
print(list)
[1, 2, 6, 3]
```

Lists

```
names = ['a', 'a', 'b', 'c', 'a']
# Count the letter a.
value = names.count('a')
print(value)

# Input list.
values = ["uno", "dos", "tres", "cuatro"]

# Locate string.
n = values.index("dos")
print(n, values[n])

# Locate another string.
n = values.index("tres")
print(n, values[n])
```

Lists, Sets

```
def remove_duplicates(values):
    output = []
    seen = set()
    for value in values:
        # If value has not been encountered yet, add it
        if value not in seen:
            output.append(value)
            seen.add(value)
    return output

# Remove duplicates from this list.
values = [5, 5, 1, 1, 2, 3, 4, 4, 5]
result = remove_duplicates(values)
print(result)
```

Lists, Sets

```
# Our input list.
values = [5, 5, 1, 1, 2, 3, 4, 4, 5]

# Convert to a set and back into a list.
setvalue = set(values)
result = list(setvalue)
print(result)

>>> print(set("my name is Serdar and Serdar is my
name".split()))
{'name', 'my', 'is', 'Serdar', 'and'}
```

Lists, Sets

Set(): Sets are lists with no duplicate entries

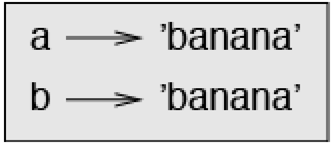
```
>>> seen = set()
>>> seen
set()
>>> seen.add(5)
>>> seen
{5}
>>> type(seen)
<class 'set'>
>>> seen.add(5)
>>> seen
{5}
>>> seen.add(4)
>>> seen
{4, 5}
>>> seenL = list(seen)
>>> seenL
[4, 5]
```

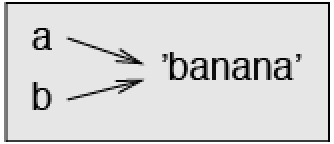
Lists Objects and Values

If we execute these assignment statements:

```
a = 'banana'
b = 'banana'
```

We know that a and b both refer to a string, but we don't know whether they refer to the *same* string. There are two possible states. To check whether two variables refer to the same object, you can use the is operator.

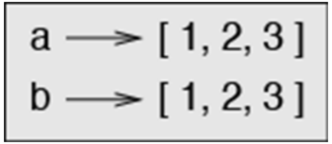





```
>>> a is b
True
```

But when you create two lists, you get two objects:

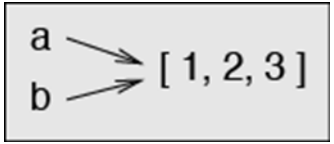
```
>>> a = [1, 2, 3]
>>> b = [1, 2, 3]
>>> a is b
False
```



Lists Objects and Values

If a refers to an object and you assign b = a, then both variables refer to the same object:

```
>>> a = [1, 2, 3]
>>> b = a
>>> b is a
True
```



The association of a variable with an object is called a reference. In this example, there are two references to the same object. An object with more than one reference has more than one name, so we say that the object is aliased. If the aliased object is mutable, changes made with one alias affect the other:

```
>>> b[0] = 17
>>> print (a)
[17, 2, 3]
```

!

Although this behavior can be useful, it is error-prone. In general, it is safer to avoid aliasing when you are working with mutable objects.

Lists Objects and Values

When you pass a list to a function, the function gets a reference to the list. If the function modifies a list parameter, the caller sees the change. For example, `delete_head` removes the first element from a list:

```
def delete_head(t):
    del t[0]
Here's how it is used:
>>> letters = ['a', 'b', 'c']
>>> delete_head(letters)
>>> print(letters)
['b', 'c']
>>> name = list('Serdar')
>>> delete_head(name)
>>> print(name)
['e', 'r', 'd', 'a', 'r']
```

The parameter t and the variable letters are aliases for the same object

Lists Objects and Values

```
def no_side_effects(cities):
    print(cities)
    cities = cities + ["Istanbul", "Ankara"]
    print(cities)

locations = ["London", "New York", "Paris"]

no_side_effects(locations)  # passing a list to a function
?????
print(locations)
?????
```

Lists Objects and Values

```
def side_effects(cities):
    print(cities)
    cities += ["Istanbul", "Ankara"]
    print(cities)

locations = ["London", "New York", "Paris"]

side_effects(locations) # passing a list to a function
?????
print(locations)
?????
```

Lists Objects and Values

```
def side_effects(cities):
    print(cities)
    cities += ["Istanbul", "Ankara"]
    print(cities)

locations = ["London", "New York", "Paris"]

side_effects(locations[:])# shallow copy of the list
?????
print(locations)
?????
```

Classworks

`ord(c)`

Given a string representing one Unicode character, return an integer representing the Unicode code point of that character. For example, **`ord('a')`** returns the integer 97 and **`ord('€')`** (Euro sign) returns 8364. This is the inverse of **`chr()`**.

`chr(i)`

Return the string representing a character whose Unicode code point is the integer i. For example, **`chr(97)`** returns the string **`'a'`**, while **`chr(957)`** returns the string **`'ν'`**. This is the inverse of **`ord()`**. The valid range for the argument is from 0 through 1,114,111 (0x10FFFF in base 16). **`ValueError`** will be raised if i is outside that range.

Classworks

```
# Create a List of number between 0 .. 255
>>> L = random.sample(range(65, 127), 30)
```

1. Convert the integer values of the List into the character
2. Sort the letters
3. Create a new List with unique letters
4. Count how many times each letter appears!!

```
number = 100

# apply chr() function on integer value
result = chr(number)
print("Integer - {} converted to character -".format(number), result)

Integer - 100 converted to character - d
```

Dictionaries

A dictionary is like a list, but more general. In a list, the indices have to be integers; in a dictionary they can be (almost) any type. You can think of a dictionary as a mapping between a set of indices (which are called keys) and a set of values. Each key maps to a value. The association of a key and a value is called a key-value pair or sometimes an item.

The squiggly-brackets, { }, represent an empty dictionary

Dictionary vs List

Values in lists are accessed by means of integers called indices, which indicate where in the list a given value is found.

Dictionaries access values by means of integers, strings, or other Python objects called keys, which indicate where in the dictionary a given value is found.

Both lists and dictionaries can store objects of any type.

```
>>> x = [ ]

>>> y = { }
```

Dictionaries

As an example, we'll build a dictionary that maps from English to Spanish words, so the keys and the values are all strings. The function dict creates a new dictionary with no items. Because dict is the name of a built-in function, you should avoid using it as a variable name.

```
>>> eng2sp = dict()
>>> print(eng2sp)
{}
>>> eng2sp['one'] = 'uno’
```

This line creates an item that maps from the key ’one’ to the value 'uno'.

Dictionaries

```
>>> print(eng2sp)
{'one': 'uno'}
```

This output format is also an input format. For example, you can create a new dictionary with three items:

```
>>> eng2sp = {'one': 'uno', 'two': 'dos', 'three': 'tres'}

>>> print(eng2sp)
{'two': 'dos', 'three': 'tres', 'one': 'uno'}
```

The order of the key-value pairs is not the same. In fact, if you type the same example on your computer, you might get a different result. In general, the order of items in a dictionary is unpredictable.

Dictionaries

But that's not a problem because the elements of a dictionary are never indexed with integer indices. Instead, you use the keys to look up the corresponding values:

```
>>> print(eng2sp['two'])
dos
```

If the key isn't in the dictionary, you get an exception:

```
>>> print(eng2sp['four'])
Traceback (most recent call last):
  File "<pyshell#24>", line 1, in <module>
    print (eng2sp['four'])
KeyError: 'four'
```

Dictionaries

The len function works on dictionaries; it returns the number of key-value pairs:

```
>>> len(eng2sp)
3
```

The in operator works on dictionaries; it tells you whether something appears as a *key* in the dictionary.

```
>>> 'one' in eng2sp
True
>>> 'uno' in eng2sp
False
```

Dictionaries

To see whether something appears as a value in a dictionary, you can use the method values, which returns the values as a list, and then use the in operator:

```
>>> vals = eng2sp.values()
>>> 'uno' in vals
True
```

The in operator uses different algorithms for lists and dictionaries.

Dictionaries

You can obtain all the keys in the dictionary with the keys method.

```
>>> list(eng2sp.keys())
['two', 'three', 'one']
```

It’s also possible to obtain all the values stored in a dictionary, using values:

```
>>> list(eng2sp.values())
['dos', 'tres', 'uno']
```

You can use the <u>items</u> method to return all keys and their associated values as a sequence of tuples:

```
>>> list(eng2sp.items())
[('two', 'dos'), ('three', 'tres'), ('one', 'uno')]
```

Dictionaries

If you want to safely get a key's value in case of the key is not already in dict, you can use the setdefault method:

```
>>> eng2sp.setdefault('four','No Translation')
'No Translation`
>>> print (eng2sp['four'])
No Translation
```

`defaultdict` is useful for settings defaults before filling the `dict` and `setdefault` is useful for setting defaults while or after filling the `dict`.

```
new = {}
for (key, value) in data:
    group = new.setdefault(key, []) # key might exist already
    group.append( value )
```

Dictionaries

You can obtain a copy of a dictionary using the copy method:

```
>>> x = {0: 'zero', 1: 'one'}
>>> y = x.copy()
>>> y
{0: 'zero', 1: 'one'}
```

This makes a shallow copy of the dictionary. This will likely be all you need in most situations. The update method updates a first dictionary with all the key/value pairs of a second dictionary. For keys that are common to both, the values from the second dictionary override those of the first:

```
>>> z = {1: 'One', 2: 'Two'}
>>> x = {0: 'zero', 1: 'one'}
>>> x.update(z)
>>> x
{0: 'zero', 1: 'One', 2: 'Two'}
```

Dictionaries and Lists

You want to count how many times each letter appears. There are several ways you could do it:

1. You could create 26 variables, one for each letter of the alphabet. Then you could traverse the string and, for each character, increment the corresponding counter, probably using a chained conditional.

2. You could create a list with 26 elements. Then you could convert each character to a number (using the built-in function ord), use the number as an index into the list, and increment the appropriate counter.

3. You could create a dictionary with characters as keys and counters as the corresponding values. The first time you see a character, you would add an item to the dictionary. After that you would increment the value of an existing item.

Dictionaries and Lists

An implementation is a way of performing a computation; some implementations are better than others.

```
def histogram(s):
        d = dict()
        for c in s:
                if c not in d:
                        d[c] = 1
                else:
                        d[c] += 1
        return d
>>> h = histogram('brontosaurus')
>>> print(h)
{'t': 1, 'u': 2, 'r': 2, 's': 2, 'o': 2, 'n': 1, 'b': 1, 'a': 1}
```

Dictionaries and Lists

```
def print_hist(h):
    for c in h:
        print(c, h[c])

>>> print_hist(h)
t 1
u 2
r 2
s 2
o 2
n 1
b 1
a 1
```

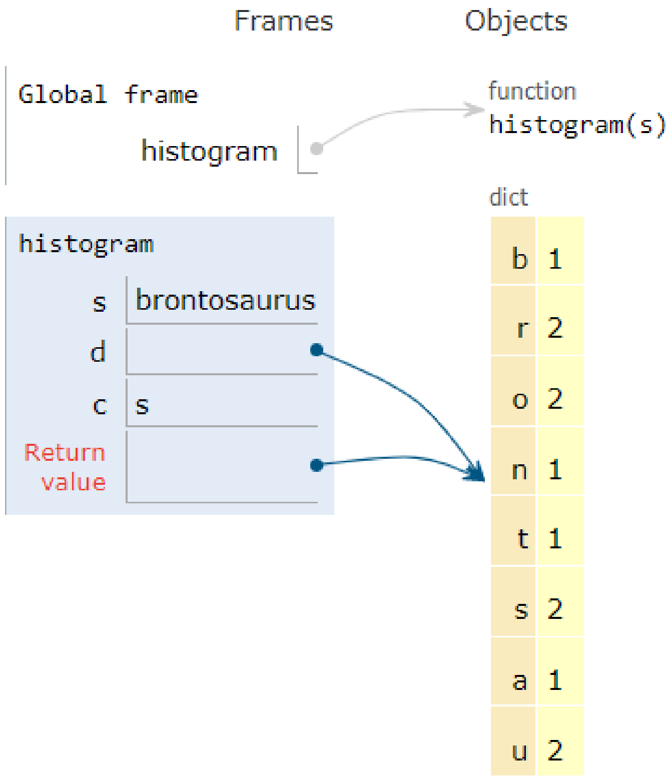
Dictionaries and Lists

```python
def histogram(s):
    d = dict()
    for c in s:
        if c not in d:
            d[c] = 1
        else:
            d[c] += 1
    return d
h = histogram('brontosaurus')
```

Edit code

<< First | < Back | Step 43 of 43 | Forward > | Last >>

line that has just executed | next line to execute



Dictionaries and Lists

Given a dictionary d and a key k, it is easy to find the corresponding value v = d[k]. This operation is called a lookup.

But what if you have v and you want to find k? You have two problems: first, there might be more than one key that maps to the value v. Depending on the application, you might be able to pick one, or you might have to make a list that contains all of them. Second, there is no simple syntax to do a reverse lookup; you have to search.

```
def reverse_lookup(d, v):
        for k in d:
                if d[k] == v:
                        return k
        raise ValueError
```

Dictionaries and Lists

```
>>> print(reverse_lookup(h, 2))
r
>>> print(reverse_lookup(h, 3))
Traceback (most recent call last):
  File "<pyshell#46>", line 1, in <module>
    print(reverse_lookup(h, 3))
  File "D:/Lectures/BCO 601 Python
Programming/dictExample1.py", line 18, in reverse_lookup
    raise ValueError
ValueError
>>>
```

Dictionaries and Lists

```
>>> d = {‘deniz':‘mavi', ‘ağaç':‘yeşil',‘ateş':‘kırmızı'}

>>> for k in d:
        print(k)

deniz
ağaç
ateş

>>> for k in d:
    print(k, '-->', d[k])

ağaç --> yeşil
deniz --> mavi
ateş --> kırmızı
```

Dictionaries and Lists

```
>>> for k, v in d.items():
        print(k, '-->', v)

ağaç --> yeşil
deniz --> mavi
ateş --> kırmızı

>>> names = [‘deniz', ‘ağaç',‘ateş']
>>> colors = [‘mavi', ‘yeşil',‘kırmızı']

>>> d = dict(zip(names, colors))

>>> print(d)
{‘ateş': ‘kıtmızı', ‘ağaç': ‘yeşil', ‘deniz': ‘mavi'}
```

Tuples

A tuple is a sequence of values. The values can be any type, and they are indexed by integers, so in that respect tuples are a lot like lists. The important difference is that tuples are immutable. Syntactically, a tuple is a comma-separated list of values:

```
>>> t = 'a', 'b', 'c', 'd', 'e`
>>> type(t)
<class 'tuple'>
```

Although it is not necessary, it is common to enclose tuples in parentheses:

```
>>> t = ('a', 'b', 'c', 'd', 'e')
```

Tuples

To create a tuple with a single element, you have to include a final comma:

```
>>> t1 = 'a',
>>> type(t1)
<type 'tuple'>
```

A value in parentheses is not a tuple:

```
>>> t2 = ('a')
>>> type(t2)
<type 'str'>
```

Tuples

Another way to create a tuple is the built-in function tuple. With no argument, it creates an empty tuple:

```
>>> t = tuple()
>>> print(t)
()
>>> t = tuple('lupins')
>>> print(t)
('l', 'u', 'p', 'i', 'n', 's')
```

Tuples

Because tuple is the name of a built-in function, you should avoid using it as a variable name. Most list operators also work on tuples. The bracket operator indexes an element:

```
>>> t = ('a', 'b', 'c', 'd', 'e')
>>> print(t[0])
'a'
```

And the slice operator selects a range of elements.

```
>>> print(t[1:3])
('b', 'c')
```

But if you try to modify one of the elements of the tuple, you get an error:

```
>>> t[0] = 'A'
TypeError: object doesn't support item assignment
```

Tuples

You can’t modify the elements of a tuple, but you can replace one tuple with another:

```
>>> t = ('A',) + t[1:]
>>> print(t)
('A', 'b', 'c', 'd', 'e')
```

It is often useful to swap the values of two variables. With conventional assignments, you have to use a temporary variable.

```
>>> temp = a
>>> a = b
>>> b = temp
```

This solution is cumbersome; tuple assignment is more elegant:

```
>>> a, b = b, a
```

The left side is a tuple of variables; the right side is a tuple of expressions. Each value is assigned to its respective variable.

Tuples

```
>>> addr = 'monty@python.org'
>>> uname, domain = addr.split('@')
```

The return value from split is a list with two elements; the first element is assigned to `uname`, the second to domain.

```
>>> print(uname)
monty
>>> print(domain)
python.org
```

Tuples

A function can only return one value, but if the value is a tuple, the effect is the same as returning multiple values. For example, if you want to divide two integers and compute the quotient and remainder, it is inefficient to compute x//y and then x%y. The built-in function divmod takes two arguments and returns a tuple of two values, the quotient and remainder. You can store the result as a tuple:

```
>>> t = divmod(7, 3)
>>> print(t)
(2, 1)
```

Tuples

Or use `tuple` assignment to store the elements separately:

```
>>> quot, rem = divmod(7, 3)
>>> print (quot, rem)
2 1
```

Here is an example of a function that returns a `tuple`:

```
def min_max(t):
        return min(t), max(t)
```

Namedtuple

import the collections module and then specify the Type name. And pass a string list of all the field names—these can then be directly used as fields:

```
import collections

# Specify the Student namedtuple.
Student = collections.namedtuple("Student",["id","name", "course"])

# Create Student instance.
s = Student(1, "Serdar Aritan", "BCO601")
>>> s
Student(id=1, name='Serdar Aritan', course='BCO601')
# Display Student.
print(s)
print("Student", s.name, "has taken the course", s.course)
```

Namedtuple

With `_make()` we create a `namedtuple` from an iterable (like a list). The list elements are turned into tuple fields.

```
import collections

# A namedtuple type.
Style = collections.namedtuple("Style", ["color", "size", "width"])

# A list containing three values.
values = ["red", 10, 15]

# Make a namedtuple from the list.
ntuple = Style._make(values)
print(ntuple)
```

Namedtuple

Benchmark tests how long it takes to create a `namedtuple` versus a regular tuple.

```
import collections
import time
# The namedtuple instance.
Animal = collections.namedtuple("Animal", ["size", "color"])
start = time.time()
# Version 1: create namedtuple.
i = 0
while i < 10000000:
    a = Animal(100, "blue")
    if a[0] != 100:
        raise Exception()
    i += 1

print("namedtuple : ", time.time() - start)
```

Namedtuple

Benchmark tests how long it takes to create a `namedtuple` versus a regular tuple..

```
start = time.time()
# Version 2: create tuple
i = 0
while i < 10000000:
    a = (100, "blue")
    if a[0] != 100:
        raise Exception()
    i += 1

print("tuple : ", time.time() - start)

namedtuple :  6.765365839004517
tuple :  2.1561741828918457
```

Turtle Graphics

Remember the random shapes.

```
import turtle
import random

turtle.title("Polygon Random Pattern")
turtle.setup(500, 500, 0, 0)

def polygon(sides, length):
    for x in range(sides):
        turtle.forward(length)
        turtle.right(360/sides)
turtle.setup(600, 600)
# hide the turtle
turtle.hideturtle()
# tell python to no longer update the graphics
turtle.tracer(0)
```

Polygon Function

Turtle Graphics

```
side_len = 50

# function names are references to functions
# and that we can assign multiple names to the same function

def besgen(): return polygon(5, side_len)
def kare(): return polygon(4, side_len)
def ucgen(): return polygon(3, side_len)
```

```
besgen = lambda : polygon(5, side_len)
kare = lambda : polygon(4, side_len)
ucgen = lambda : polygon(3, side_len)

shape_list = [besgen, kare, ucgen]
```

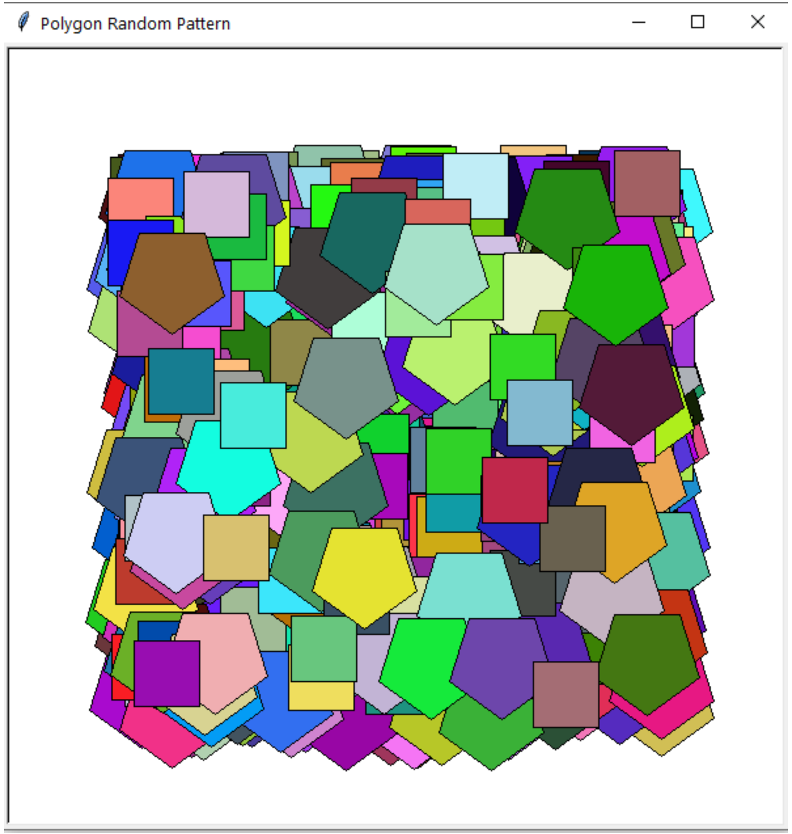
Turtle Graphics

```
for _ in range(1000):

    # choose a random spot
    xpos = random.randint(-200,200)-(side_len/2)
    ypos = random.randint(-200,200)+(side_len/2)

    # goto this spot
    turtle.penup()
    turtle.goto(xpos, ypos)
    turtle.pendown()

    # generate a random color
    red = random.random() # returns a number between 0 and 1
    green = random.random()
    blue = random.random()
```

Turtle Graphics

```
    # fill in our shape
    turtle.fillcolor(red, green, blue)

    # draw the shape
    turtle.begin_fill()
    shape_list[random.randint(0,1)]()
    turtle.end_fill()

    # update the screen with our drawing
    turtle.update()

turtle.exitonclick()
```

Turtle Graphics

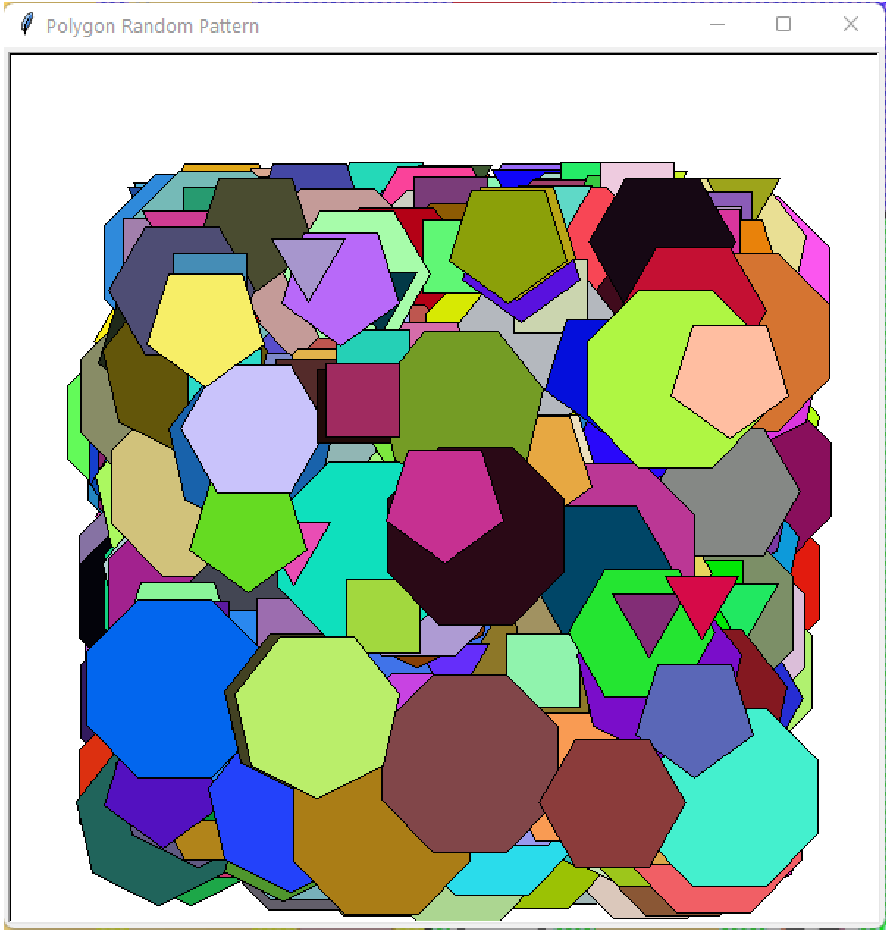


Assign functions to the dictionary

```
import turtle
import random

def polygon(sides, length):

    for x in range(sides):
        turtle.forward(length)
        turtle.right(360/sides)

side_len = 50
# Assign functions to the dictionary

gens = {3 : lambda : polygon(3, side_len),
        4 : lambda : polygon(4, side_len),
        5 : lambda : polygon(5, side_len),
        }

gens[random.randint(3,5)]()
```

Assign functions to the dictionary

```
import turtle
import random
import functools
# functools.partial lets you attach arguments to an existing function.

def polygon(sides, length):

    for x in range(sides):
        turtle.forward(length)
        turtle.right(360/sides)

side_len = 50

values = [functools.partial(polygon,x, side_len) for x in range(3, 9)]

gens =  dict(list(enumerate(values,3))) # Assign functions to the dictionary

gens[random.randint(3,8)]()
```


Polygon Random Pattern

List Example

```
menu_item = 0
namelist = []
while menu_item != 9:
    print("--------------------")
    print("1. Liste Ciktisi")
    print("2. Listeye Isim Ekle")
    print("3. Listeden Isim Sil")
    print("4. Isim Guncelle")
    print("9. Cikis")
    menu_item = int(input("Islem Seciniz : "))
    print(menu_item)
    if menu_item == 1:
        current = 0
        if len(namelist) > 0:
            while current < len(namelist):
                print(current, ".", namelist[current])
                current = current + 1
        else:
            print("Bos Liste")
    elif menu_item == 2:
        name = input("Eklemek icin Isim Giriniz : ")
        namelist.append(name)
```

List Example

```
 elif menu_item == 3:
        del_name = input("Hangi Ismi Silmek Istersiniz: ")
        if del_name in namelist:
            namelist.remove(del_name)
        else:
            print(del_name, "bulunamadi")
  elif menu_item == 4:
        old_name = input("Hangi Ismi Guncellemek Istersiniz : ")
        if old_name in namelist:
            item_number = namelist.index(old_name)
            new_name = input("Yeni Ismi Giriniz : ")
            namelist[item_number] = new_name
        else:
            print(old_name, "bulunamdi")

print("Hoscakalin")
```

Oops…… There a is slight problem!!

List Example

```
elif menu_item == 3:
    del_name = input("Hangi Ismi Silmek Istersiniz: ")
    if del_name in namelist:
        item_number = namelist.index(del_name)
        while del_name in namelist:
            item_number = namelist.index(del_name)
            del namelist[item_number]
    else:
        print(del_name, "bulunamadi")
```