

# Functions, Turtle Graphics

#4



Serdar ARITAN

Department of Computer Graphics  
Hacettepe University, Ankara, Turkey

Printing in Python is probably simpler than some of its details may imply. To illustrate, let's run some quick examples.

```
>>> print()

>>> x = 'spam'           # String
>>> y = 91                # Integer
>>> z = ['hello']         # List
>>> print(x, y, z)
spam 91 ['hello']
>>> print(x, y, z, sep='') # Suppress separator
spam91['hello']
>>>
>>> print(x, y, z, sep=', ') # Custom separator
spam, 91, ['hello']
```

**Automatic Newline.** Normally, any print command automatically adds a newline ( ' \n ' ) character at the end of what is printed

```
print("one",1)
print("two",2)
print("three",3)
one 1
two 2
three 3
```

**What if you would rather not have this behavior? A special syntax for this in Python3 is the following:**

```
print("one",1, end=' ')
print("two",2, end=' ')
print("three",3)
```

Also by default, print adds an end-of-line character to terminate the output line.

```
>>> print(x, y, z, end='')
spam 91 ['hello']
>>> print(x, y, z, end='...\n') # Custom line end
spam 91 ['hello']...
>>> print(x, y, z, sep='...', end='!\n')
# Multiple keywords
spam...91...['hello']!
>>> print(x, y, z, end='!\n', sep='...')
# Order doesn't matter
spam...91...['hello']!
```

Finally, keep in mind that the separator and end-of-line options provided by print operations are just conveniences.



When you run a Python script by typing a shell command line such as `python dir1\dir2\file.py`, the CWD is the directory you were in when you typed this command, not `dir1\dir2`.

```
>>> import os, sys
```

```
>>> print('my os.getcwd =>', os.getcwd()) #show my cwd execution dir
my os.getcwd => C:\Program Files\Python312
```

```
>>> print('my sys.path =>', sys.path[:6]) #show first 6 import paths.
my sys.path => ['', 'C:\\Program Files\\Python312\\Lib\\idlelib',
'C:\\Program Files\\Python312\\python312.zip', 'C:\\Program
Files\\Python312\\DLLs', 'C:\\Program Files\\Python312\\Lib',
'C:\\Program Files\\Python312']
```

Here is how the **file** keyword argument is used—it directs the printed text to an open output file or other compatible object for the duration of the single print (this is really a form of **stream redirection**, a topic we will revisit later)

```
# Print to a file
```

```
>>> print(x, y, z, sep='...', file=open('data.txt', 'w'))
```

```
>>> print(x, y, z) # Back to stdout
```

```
spam 91 ['hello']
```

```
>>> print(open('data.txt').read()) # Display file text
```

```
spam...91...['hello']
```



```
>>> print('hello world') # Print a string object in 3.X  
hello world
```

Because expression results are echoed on the interactive command line, you often don't even need to use a print statement there

```
>>> 'hello world' # Interactive echoes  
'hello world'
```



if you enjoy **working harder** than you must, you can also code print operations this way:

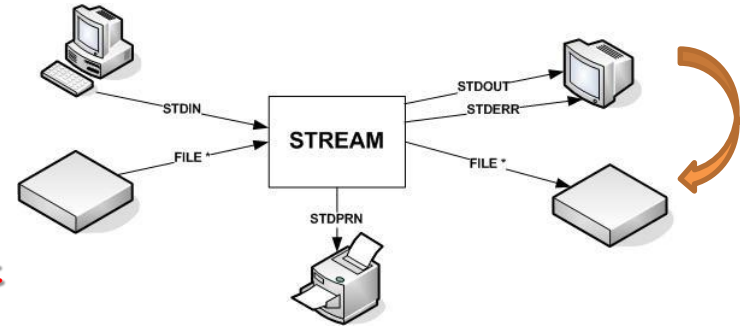
```
>>> import sys # Printing the hard way  
>>> sys.stdout.write('hello world\n')  
hello world  
12
```

This code explicitly calls the write method of `sys.stdout`

The long form isn't all that useful for printing by itself. However, it is useful to know that this is exactly what print operations do because it is possible to reassign `sys.stdout` to something different from the standard output stream.



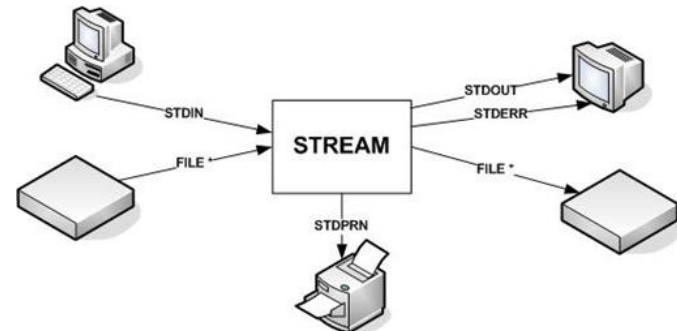
```
import sys
sys.stdout = open('log.txt', 'a')
# Redirects prints to a file
...
print(x, y, x) # Shows up in log.txt
```



we reset `sys.stdout` to a manually opened file named `log.txt`, located in the script's working directory and opened in append mode



```
>>> import sys
>>> temp = sys.stdout # Save for restoring later
>>> sys.stdout = open('log.txt', 'a')
# Redirect prints to a file
>>> print('spam')
>>> print(1, 2, 3)
>>> sys.stdout.close() # Flush output to disk
>>> sys.stdout = temp # Restore original stream
>>> print('back here') # Prints show up here again
back here
>>> print(open('log.txt').read())
# Result of earlier prints
spam
1 2 3
```



Suppose we need to print something in a complicated list, for instance a list that contains tuples, strings, and a dictionary. A typical interactive session might be:

```
>>> a = ("R", False, 78)
>>> b = { "white": (255, 255, 255), "black": (0, 0, 0) }
>>> c = [ a, b, "+++", b ]
>>> print(c)
[('R', False, 78), {'white': (255, 255, 255), 'black': (0, 0, 0)},
'+++', {'white': (255, 255, 255), 'black': (0, 0, 0)}]
>>> from pprint import *
>>> pprint(c)
[('R', False, 78),
{'black': (0, 0, 0), 'white': (255, 255, 255)},
'+++',
{'black': (0, 0, 0), 'white': (255, 255, 255)}]
```



```
>>> "hello" + "world"    "helloworld"    # concatenation
>>> "hello" * 3          "hellohellohello"  # repetition
>>> "hello"[0]           "h"                # indexing
>>> "hello"[-1]          "o"                # (from end)
>>> "hello"[1:4]         "ell"              # slicing
>>> len("hello")         5                  # size
>>> "hello" < "jello"    1                  # comparison
>>> "e" in "hello"       1                  # search
```

```
Python Shell
File Edit Shell Debug Options Windows Help
>>> "hello"+"world"
'helloworld'
>>> "hello"*3
'hellohellohello'
>>> "hello"[0]
'h'
>>> "hello"[-1]
'o'
>>> "hello"[1:4]
'ell'
>>> len("hello")
5
>>> "hello" < "jello"
True
>>> "e" in "hello"
True
Ln: 23 Col: 4
```



- Characters in a string are numbered with indexes starting at 0:

- Example:

```
name = "S.Arıtan"
```

Be Careful: Starts from 0

index	0	1	2	3	4	5	6	7
Character	S	.	A	r	ı	t	a	n

- Accessing an individual character of a string:

```
variableName [ index ]
```

- Example:

```
print (name, "starts with", name[0])
```

Output:

```
S.Arıtan starts with S
```





## String Index Positions

s[-9]	s[-8]	s[-7]	s[-6]	s[-5]	s[-4]	s[-3]	s[-2]	s[-1]
L	i	g	h	t		r	a	y
s[0]	s[1]	s[2]	s[3]	s[4]	s[5]	s[6]	s[7]	s[8]

**The slice operator has three syntaxes:**

`seq[start]`

`seq[start:end]`

`seq[start:end:step]`

**The seq can be any sequence, such as a list, string, or tuple. The start, end, and step values must all be integers (or variables holding integers).**

## String Index Positions

← s[4:11] → ← s[-3:] →

T	h	e		w	a	x	w	o	r	k		m	a	n
---	---	---	--	---	---	---	---	---	---	---	--	---	---	---

← s[:7] → ← s[7:] →

```
>>> s = s[:12] + "wo" + s[12:]
```

```
>>> s
```

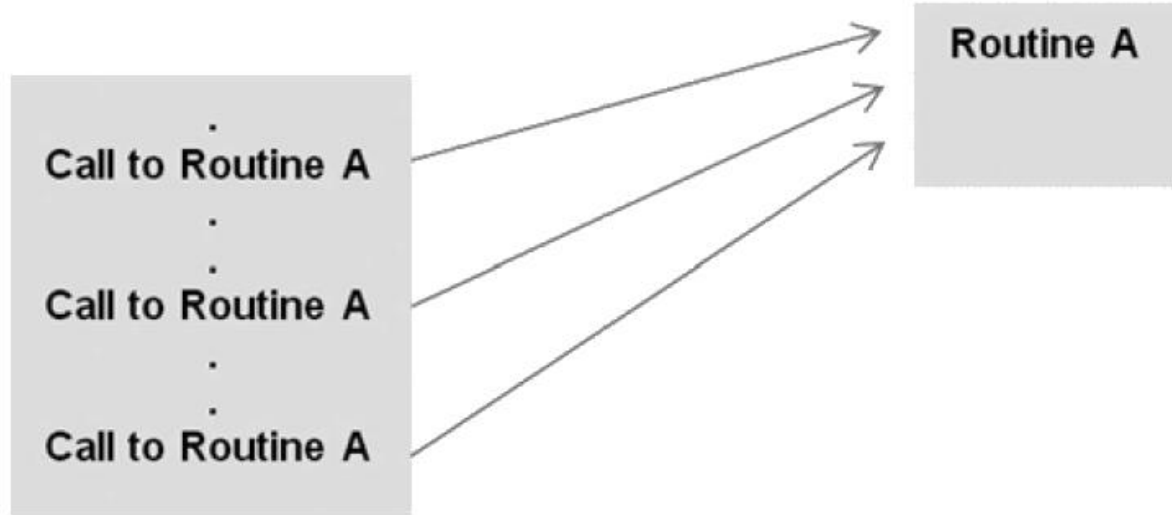
```
'The waxwork woman'
```

In fact, since the text "wo" appears in the original string, we could have achieved the same effect by assigning

```
s[:12] + s[7:9] + s[12:]
```

```
>>> s
```

```
'The waxwork woman'
```



**A program routine is a named group of instructions that accomplishes some task. A routine may be invoked (called) as many times as needed in a given program. A function is Python's version of a program routine.**

The first step to code reuse is the function: **a named piece of code**, separate from all others. A function can take **any number and type of input parameters** and **return any number and type of output results**. You can do two things with a function:

- Define it
- Call it

```
>>> def do_nothing():  
    pass  
  
>>> do_nothing()  
>>> type(do_nothing)  
<class 'function'>  
>>> do_nothing  
<function do_nothing at 0x00000001036AFD90>  
>>>
```

Just like a value can be associated with a name, a piece of logic can also be associated with a name by defining a function.

```
>>> def square(x):  
    return x * x
```



```
>>> square(5)  
25
```

The body of the function is **indented**. Indentation is the Python's way of grouping statements.

The functions can be used in any expressions.

```
>>> square(2) + square(3)  
13  
>>> square(square(3))  
81
```

```
def name (arg1, arg2, ... argN) :  
    statements
```

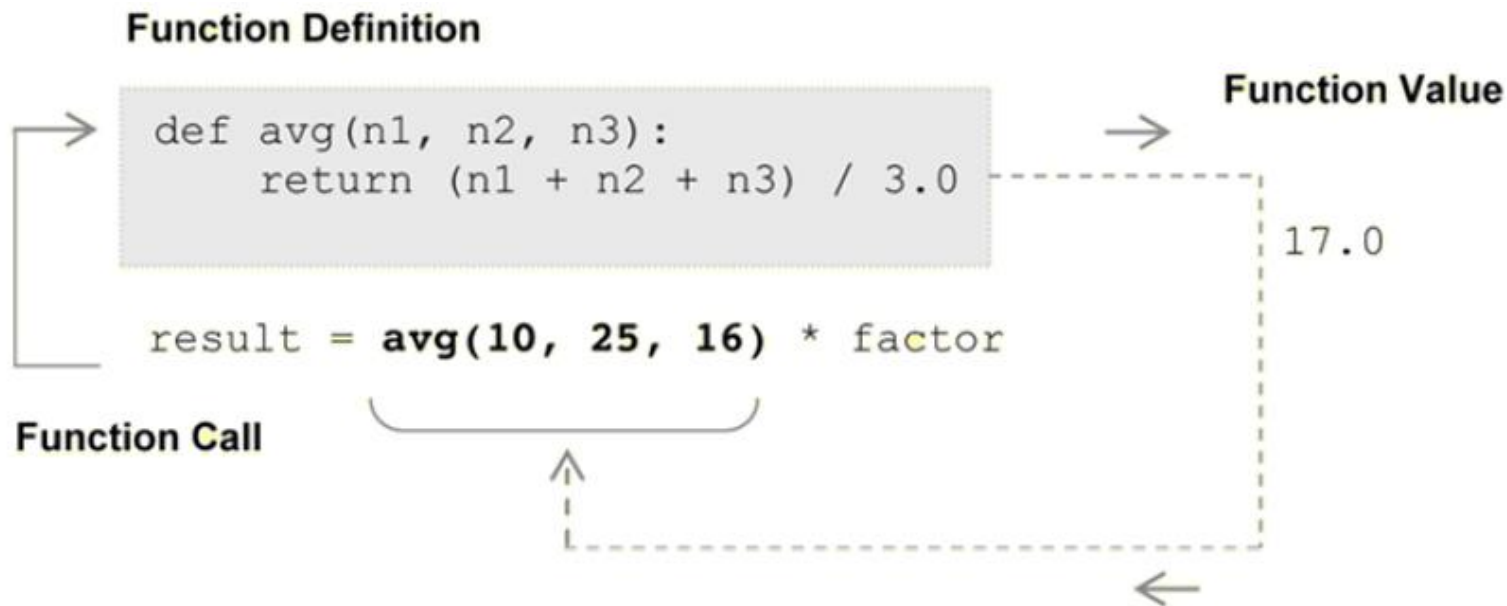
As with all compound Python statements, **def** consists of a header line followed by a block of statements, usually indented (or a simple statement after the colon).

Function Header ➤ `def avg (n1, n2, n3) :`

Function Body (suite) ➤ {  
    -----  
    -----  
    -----  
    -----

```
>>> num1 = 10  
>>> num2 = 25  
>>> num3 = 16  
>>> avg (num1 , num2 , num3)
```

Function names have the **same rules as variable names** (they must start with a letter or `_` and contain only letters, numbers, or `_`).



## Call to Value-Returning Function

define and call another function that has **no parameters** but prints a single word:

```
>>> def make_a_sound():  
    print('quack')
```

```
>>> make_a_sound()  
quack
```

When you called the `make_a_sound()` function, Python ran the code inside its definition. A function that has **no parameters** but **returns** a value:

```
>>> def agree():  
    return True
```



We can call this function and test its returned value by using if:

```
>>> if agree(): # if True:
    print('Splendid!')
else:
    print('That was unexpected.')
```

Splendid!



it's time to put something between those parentheses

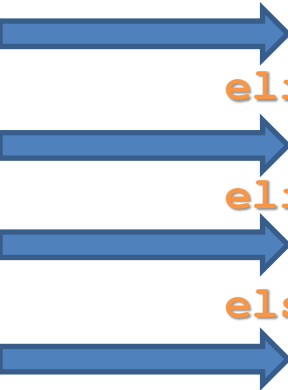
```
>>> def echo(anything):  
    return anything + ' ' + anything
```

```
>>> echo('Is there anybody')  
'Is there anybody Is there anybody'
```

```
>>> echo(1500)  
TypeError: unsupported operand type(s) for +: 'int' and 'str'
```

a function that takes an input argument and does something with it

```
def commentary(color):  
    if color == 'red':  
        return "It's a tomato."  
    elif color == "green":  
        return "It's a green pepper."  
    elif color == 'bee purple':  
        return "I don't know what it is, but only bees can see it."  
    else:  
        return "I've never heard of the color " + color + "."
```



**None** is a special Python value that holds a place when there is nothing to say. It is not the same as the boolean value **False**

```
>>> None == False
False
```

```
>>> thing = None
>>> if thing:
    print("It's some thing")
else: # It is like False
    print("It's no thing")
```

It's no thing

To distinguish **None** from a boolean **False** value, use Python's **is** operator:

```
>>> thing = None
>>> if thing is None:
    print("It's nothing")
else: # It is like False
    print("It's something")
```

It's nothing

**Functions in Python are first-class objects.** Programming language theorists define a “*first-class object*” as a program entity that can be:

- Created at runtime
- Assigned to a variable or element in a data structure
- Passed as an argument to a function
- Returned as the result of a function

## When to Use a Function

**Only one purpose:** A function should be the encapsulation of a single, identifiable operation.

**Readable:** A function should be readable.

**Not too long:** A function shouldn't be too long.

**Reusable:** A function should be reusable in contexts other than the program it was written for originally.

**Complete:** A function should be complete, in that it works in all potential situations. If you write a function to perform one thing, you should make sure that *all the cases* where it might be used are taken into account.

**Able to be refactored:** Refactoring is the process of taking existing code and modifying it such that its structure is somehow improved but the functionality of the code remains the same.



Functions are generally defined at the top of a program. However, every function **must be defined before it is called**.

**def** Executes at Runtime



The Python **def** is a true executable statement: when it runs, it creates a new function object and assigns it to a name. Because it's a statement, a **def** can appear anywhere a statement can—even nested in other statements

```
if test:
```

```
    def func():    # Define func this way
```

```
    ...
```

```
else:
```

```
    def func():    # Or else this way
```

```
    ...
```

```
func()              # Call the version selected and built
```





Generally, **defs** are not evaluated until they are reached and run, and the code inside **defs** is not evaluated until the functions are later called. Because function definition happens at runtime, there's nothing special about the function name. What's important is the object to which it refers:

```
othername = func      # Assign function object
othername()            # Call func again
```

Here, the function was assigned to a different name and called through the new name. Like everything else in Python, functions are just objects; they are recorded explicitly in memory at program execution time.

```
def func(): ...           # Create function object
func()                   # Call object
func.attr = value        # Attach attribute
```



```
>>> def func():  
    pass  
  
>>> func()  
>>> func.value = 1  
>>> dir(func)  
['__annotations__', '__builtins__', '__call__', '__class__', '__closure__',  
 '__code__', '__defaults__', '__delattr__', '__dict__', '__dir__',  
 '__doc__', '__eq__', '__format__', '__ge__', '__get__', '__getattr__',  
 '__getstate__', '__globals__', '__gt__', '__hash__', '__init__',  
 '__init_subclass__', '__kwdefaults__', '__le__', '__lt__', '__module__',  
 '__name__', '__ne__', '__new__', '__qualname__', '__reduce__',  
 '__reduce_ex__', '__repr__', '__setattr__', '__sizeof__', '__str__',  
 '__subclasshook__', '__type_params__', 'value']  
>>> func.value  
1  
>>> func.value += 1  
>>> func.value  
2
```

The first statement in the body of a function is usually a string, which can be accessed with `function_name.__doc__`. This statement is called **Docstring**.

```
>>> def hello(name):  
    """ Greets a person """  
    print('Hello', name + '!')  
  
>>>  
>>> hello(name = 'Serdar')  
>>>  
>>> print("The docstring of the function : "+ Hello.__doc__)
```

**Docstring** ←

We can even create more functions using the existing ones.

```
>>> def sum_of_squares(x, y):  
    return square(x) + square(y)
```

```
>>> sum_of_squares(2, 3)  
13
```

```
>>> def square(x):  
    return x * x
```

Functions are just like other values, they can be assigned, passed as arguments to other functions etc.

```
>>> f = square
```

```
>>> f(4)
```


```
16
```

```
>>> def fxy(f, x, y):  
    return f(x) + f(y)
```

```
>>> fxy(square, 2, 3)
```

```
13
```

```
>>>def cube(x):  
    return x * x * x  
>>> fxy(cube, 2, 3)  
35
```



There is another way of creating functions, using the *lambda* operator.

```
>>>forthpower = lambda x: x ** 4  
>>> fxy(forthpower, 2, 3)  
97  
>>> fxy(lambda x: x ** 5, 2, 3)  
275
```

The *lambda* operator becomes handy when writing small functions to be passed as arguments etc.



Write your Python 3 code here:

```
1 def square(x):  
2     return x * x  
3  
4 def sum_of_squares(x, y):  
5     return square(x) + square(y)  
6  
7 def fxy(f, x, y):  
8     return f(x) + f(y)  
9  
10 def cube(x):  
11     return x * x * x  
12  
13 forthpower = lambda x: x ** 4  
14  
15 f = square  
16 fxy(square, 2, 3)  
17 fxy(cube, 2, 3)  
18 fxy(forthpower, 2, 3)  
19 fxy(lambda x: x ** 5, 2, 3)  
20  
21
```

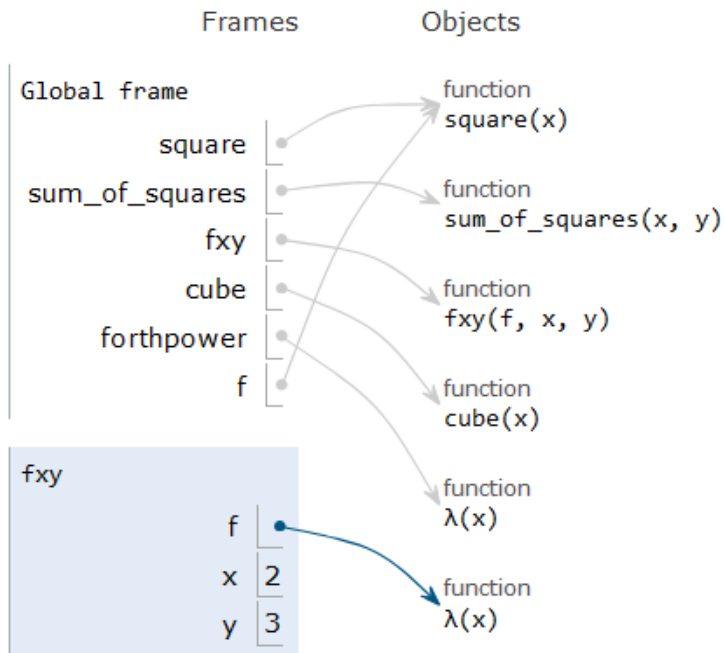
Visualize Execution



# PYTHON PROGRAMMING

```
1 def square(x):  
2     return x * x  
3  
4 def sum_of_squares(x, y):  
5     return square(x) + square(y)  
6  
7 def fxy(f, x, y):  
8     return f(x) + f(y)  
9  
10 def cube(x):  
11     return x * x * x  
12  
13 forthpower = lambda x: x ** 4  
14  
15 f = square  
16 fxy(square, 2, 3)  
17 fxy(cube, 2, 3)  
18 fxy(forthpower, 2, 3)  
19 fxy(lambda x: x ** 5, 2, 3)
```

## Functions





$n! = n * (n-1)!, \text{ if } n > 1 \text{ and } f(1) = 1$

Example:

$4! = 4 * 3!$

$3! = 3 * 2!$

$2! = 2 * 1$

```
>>> def factorial(n):  
    f = 1  
    while (n > 0):  
        f = f * n  
        n = n - 1  
    return f
```

Recursion has  
something to  
do with infinity



## Treating a Function Like an **Object**

```
>>> def factorial(n):  
    '''returns n!'''  
    return 1 if n < 2 else n * factorial(n - 1)  
  
>>> factorial(42)  
1405006117752879898543142606244511569936384000000000  
>>> factorial.__doc__  
'returns n!'  
>>> type(factorial)  
<class 'function'>  
>>> f = lambda x: x and x * f(x - 1) or 1
```

# PYTHON PROGRAMMING

# TURTLE BASICS

```
>>> import turtle as t
```

```
>>> dir (t)
```

```
['Canvas', 'Pen', 'RawPen', 'RawTurtle', 'Screen', 'ScrolledCanvas', 'Shape', 'TK', 'TNavigator',  
'TPen', 'Tbuffer', 'Terminator', 'Turtle', 'TurtleGraphicsError', 'TurtleScreen',  
'TurtleScreenBase', 'Vec2D', '_CFG', '_LANGUAGE', '_Root', '_Screen', '_TurtleImage', '__all__',  
'__builtins__', '__cached__', '__doc__', '__file__', '__forwardmethods__', '__func_body__',  
'__loader__', '__methodDict', '__methods__', '__name__', '__package__', '__spec__', '__stringBody__',  
'_alias_list', '_make_global_funcs', '_screen_docrevise', '_tg_classes', '_tg_screen_functions',  
'_tg_turtle_functions', '_tg_utilities', '_turtle_docrevise', '_ver', 'addshape', 'back',  
'backward', 'begin_fill', 'begin_poly', 'bgcolor', 'bgpic', 'bk', 'bye', 'circle', 'clear',  
'clearscreen', 'clearstamp', 'clearstamps', 'clone', 'color', 'colormode', 'config_dict',  
'deepcopy', 'degrees', 'delay', 'distance', 'done', 'dot', 'down', 'end_fill', 'end_poly',  
'exitonclick', 'fd', 'fillcolor', 'filling', 'forward', 'get_poly', 'get_shapepoly', 'getcanvas',  
'getmethparlist', 'getpen', 'getscreen', 'getshapes', 'getturtle', 'goto', 'heading',  
'hideturtle', 'home', 'ht', 'inspect', 'isdown', 'isfile', 'isvisible', 'join', 'left', 'listen',  
'lt', 'mainloop', 'math', 'mode', 'numinput', 'onclick', 'ondrag', 'onkey', 'onkeypress',  
'onkeyrelease', 'onrelease', 'onscreenclick', 'ontimer', 'pd', 'pen', 'pencolor', 'pendown',  
'pensize', 'penup', 'pos', 'position', 'pu', 'radians', 'read_docstrings', 'readconfig',  
'register_shape', 'reset', 'resetscreen', 'resizemode', 'right', 'rt', 'screensize', 'seth',  
'setheading', 'setpos', 'setposition', 'settiltangle', 'setundobuffer', 'setup',  
'setworldcoordinates', 'setx', 'sety', 'shape', 'shapeseize', 'shapetransform', 'shearfactor',  
'showturtle', 'simplesdialog', 'speed', 'split', 'st', 'stamp', 'sys', 'textinput', 'tilt',  
'tiltangle', 'time', 'title', 'towards', 'tracer', 'turtles', 'turtlesize', 'types', 'undo',  
'undobufferentries', 'up', 'update', 'width', 'window_height', 'window_width', 'write',  
'write_docstringdict', 'xcor', 'ycor']
```

In order to make our drawing appear almost immediately we can make use of the `tracer()` method. `tracer()` takes two arguments. One controls how often screens should be updated and the other controls the delay between these update. To obtain the fastest possible rendering, both these arguments should be set to zero.

```
>>> tracer(0, 0)
```

By calling `tracer()` with both arguments set to zero, we are essentially turning off all animation and our drawings will be drawn “immediately.” However, if we turn the animation off in this way we need to explicitly update the image with the `update()` method after we are done issuing drawing commands.

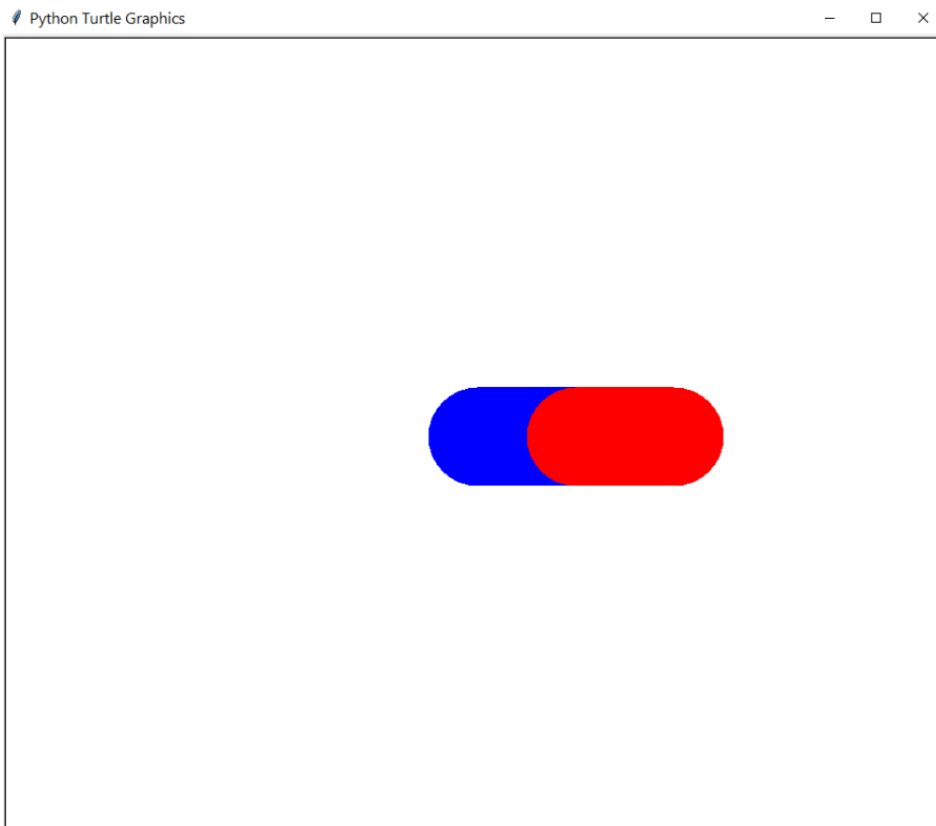
If you want to reset `tracer()` to its original settings, its arguments should be 1 and 10.

```
>>> tracer(1, 10)
```

If we want to erase everything that we previously drew, we can use either the `clear()` or `reset()` methods. `clear()` clears the drawing from the graphics window but it leaves the turtle in its current position with its current heading. `reset()` clears the drawing and also returns the turtle to its starting position in the center of the screen.

# TURTLE BASICS

```
>>> t.reset()  
>>> t.color("blue")  
>>> t.pensize(100)  
>>> t.fd(100)  
>>> t.color("red")  
>>> t.fd(100)
```



```
>>> help(t.seth)
```

Help on function seth in module turtle:

```
seth(to_angle)
```

Set the orientation of the turtle to to\_angle.

Aliases: setheading | seth

Argument:

to\_angle -- a number (integer or float)

Set the orientation of the turtle to to\_angle.

Here are some common directions in degrees:

standard - mode:	logo-mode:
0 - east	0 - north
90 - north	90 - east
180 - west	180 - south
270 - south	270 - west

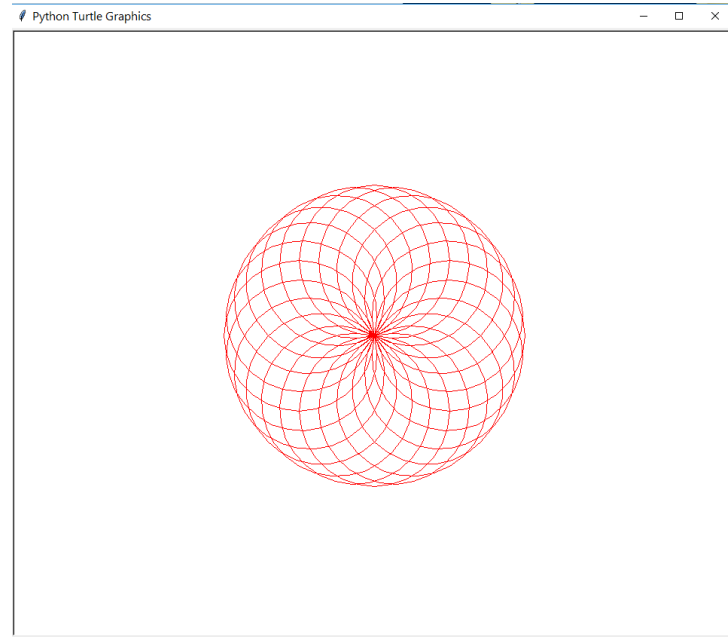
Example:

```
>>> setheading(90)
```

```
>>> heading()
```

```
90
```

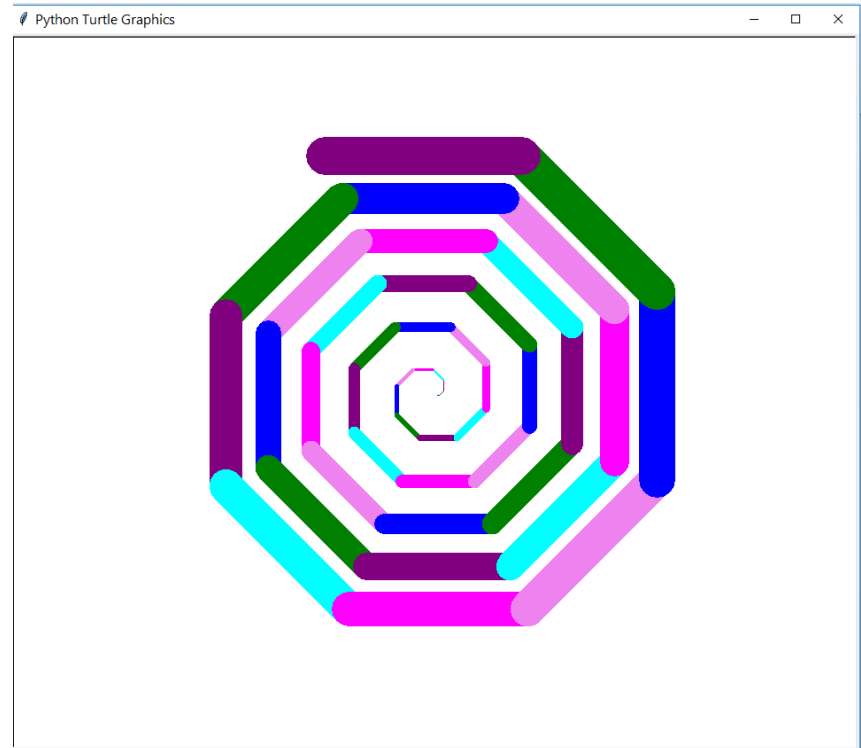
```
>>> t.reset()  
>>> t.color("red")  
>>> for angle in range(0, 360, 15):  
    t.seth(angle)  
    t.circle(100)
```



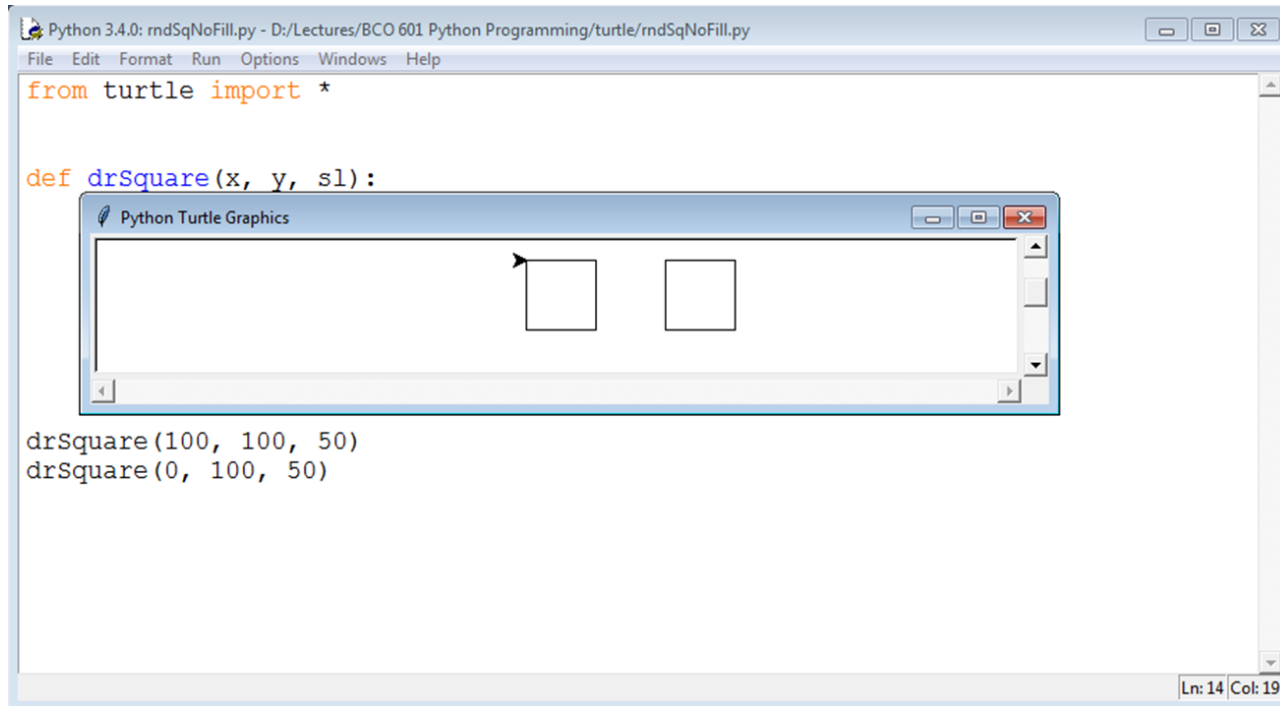


```
>>> t.reset()
>>> colors = ["blue", "green", "purple", "cyan", "magenta", "violet"]
>>> t.tracer(0, 0)
>>> for i in range (45):
    t.color(colors[i % 6])
    t.pendown()
    t.fd(2 + i * 5)
    t.left(45)
    t.width(i)
    t.penup()

>>> t.update()
```



**Write a function that takes a 2D coordinate point (x, y) and side length to draw a square.**



The screenshot shows a Python 3.4.0 IDE window titled "Python 3.4.0: rndSqNoFill.py - D:/Lectures/BCO 601 Python Programming/turtle/rndSqNoFill.py". The code editor contains the following Python code:

```
from turtle import *  
  
def drSquare(x, y, sl):  
  
    drSquare(100, 100, 50)  
    drSquare(0, 100, 50)
```

Below the code editor is a "Python Turtle Graphics" window. It displays two squares: one at the coordinate (100, 100) and another at (0, 100), both with a side length of 50. The status bar at the bottom right of the IDE window indicates "Ln: 14 | Col: 19".

<code>pencolor(<i>a_color</i>)</code>	<i>a_color</i> is either a colorstring or an rgb tuple.
<code>fillcolor(<i>a_color</i>)</code>	<i>a_color</i> is either a colorstring or an rgb tuple.
<code>colormode(255)</code>	Sets rgb values to be 0–255 (defaults to 0.0–1.0, better to set to 255).
<code>color(<i>pen_color</i>, <i>fill_color</i>)</code>	Set both with one function.
<code>begin_fill()</code>	Mark where region to color in begins.
<code>end_fill()</code>	Mark where region to color in ends.

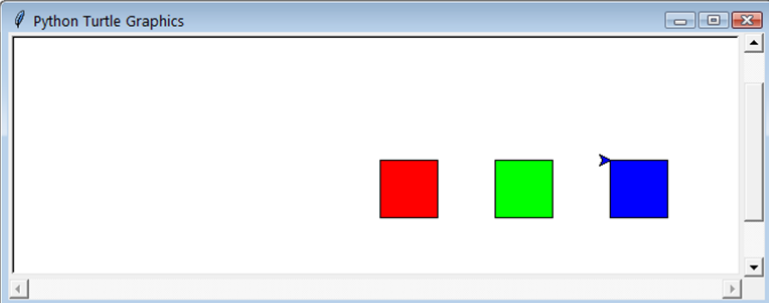
For colorstrings, Python accepts any of the standard color strings provided by Python's built-in drawing module, Tk. Common strings such as “**green**,” “**blue**,” “**red**,” and “**yellow**” are obvious, but others such as “**BlanchedAlmond**” or “**CornflowerBlue**” are less so. The full list can be found at <http://www.tcl.tk/man/tcl8.5/TkCmd/colors.htm>

```
twoSquare.py - D:/Lectures/BCO 601 Python Programming/turtle/twoSquare.py (3.4.2)
File Edit Format Run Options Windows Help

from turtle import *

def square(length, scolor):
    ''' Draw a square , side length , color fill tuple '''
    fillcolor(scolor)
    begin_fill()
    for i in range(4):
        forward(length)
        right(90)
    end_fill()

square(50, (1,0,0))
penup()
forward(100)
pendown()
square(50, (0,1,0))
penup()
forward(100)
pendown()
square(50, (0,0,1))
```



Python Turtle Graphics

Ln: 18 Col: 10

```
chessBoard.py - D:\Lectures\BCO 601 Python Programming\turtle\chessBoard.py (3.4.2)
File Edit Format Run Options Windows Help

from turtle import *
import random

colormode(255) # colors in range 0-255

def square(length, scolor):
    ''' Draw a square , side length , color fill tuple '''
    fillcolor(scolor)
    begin_fill()
    for i in range(4):
        forward(length)
        right(90)
    end_fill()

for i in range(8):
    # init values , fun to change
    red = random.randrange(1, 255)
    green = random.randrange(1, 255)
    blue = random.randrange(1, 255)
    square(50, (red, green, blue))
    forward(50)

Python Turtle Graphics

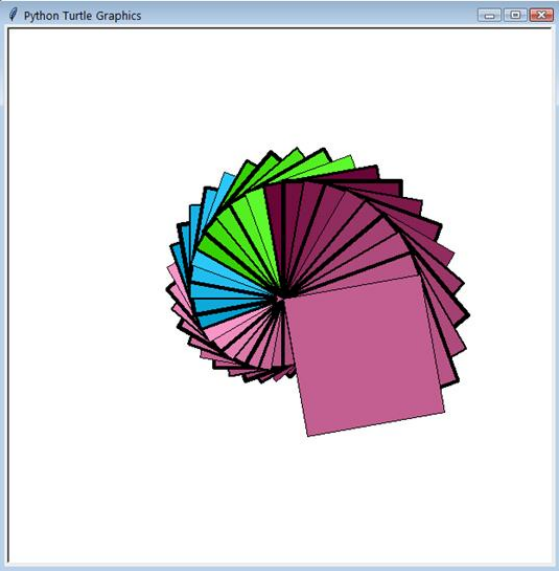
Ln: 19 Col: 35
```

```
drawSquare.py - D:/Lectures/BCO 601 Python Programming/turtle/drawSquare.py (3.4.2)
File Edit Format Run Options Windows Help
from turtle import *

colormode(255) # colors in range 0-255


def square(length, scolor):
    for i in range(4):
        forward(length)
        right(90)
        color(scolor)
        scolor = (scolor + 1) % 255

    side_length += 3
    # range 1-pen limit
    pen_width = ((pen_width + pen_inc) % pen_limit) + 1
    pensize(pen_width)
```



# PYTHON PROGRAMMING

# TURTLE BASICS



```
import turtle
# set up a title
turtle.title("BC0601 Python Programming")
# set up our stage or canvas
turtle.setup(500, 500, 0, 0)
# Square - 4 sides
turtle.forward(50)    # move forward
turtle.right(90)       # turn 90 degrees
turtle.forward(50)    # move forward again
turtle.right(90)       # turn 90 degrees again
turtle.forward(50)    # move forward again !
turtle.right(90)       # turn 90 degrees again !
turtle.forward(50)    # move forward again !
turtle.right(90)       # turn 90 degrees again
# let the user close the turtle window when they click
turtle.exitonclick()
```

# PYTHON PROGRAMMING

# TURTLE BASICS

```
import turtle
```

```
# set up a title
```

```
turtle.title("BCO601 Python Programming")
```

```
# set up your stage or canvas
```

```
turtle.setup(500, 500, 0, 0)
```

```
# Pentagon - 5 sides
```

```
turtle.forward(50)    # move forward
```

```
turtle.right(360/5)   # turn 360/5 degrees
```

```
turtle.forward(50)    # move forward again
```

```
turtle.right(360/5)   # turn 360/5 degrees again
```

```
turtle.forward(50)    # move forward again !
```

```
turtle.right(360/5)   # turn 360/5 degrees again !
```

```
turtle.forward(50)    # move forward again !
```

```
turtle.right(360/5)   # turn 360/5 degrees again
```

```
turtle.forward(50)    # move forward again !
```

```
turtle.right(360/5)   # turn 360/5 degrees again
```

```
turtle.exitonclick()
```



# PYTHON

# TURTLE BASICS

## PROGRAMMING

```
import turtle
turtle.title("BC0601 Python Programming")
turtle.setup(500, 500, 0, 0) # set up your stage or canvas
# Octagon - 8 sides
turtle.forward(50)          # move forward
turtle.right(360/8)         # turn 360/8 degrees
turtle.forward(50)          # move forward again
turtle.right(360/8)         # turn 360/8 degrees again
turtle.forward(50)          # move forward again !
turtle.right(360/8)         # turn 360/8 degrees again !
turtle.forward(50)          # move forward again !
turtle.right(360/8)         # turn 360/8 degrees again
turtle.forward(50)          # move forward again !
turtle.right(360/8)         # turn 360/8 degrees again
turtle.forward(50)          # move forward again !
turtle.right(360/8)         # turn 360/8 degrees again
turtle.forward(50)          # move forward again !
turtle.right(360/8)         # turn 360/8 degrees again
turtle.forward(50)          # move forward again !
turtle.right(360/8)         # turn 360/8 degrees again
turtle.exitonclick()
```

# PYTHON PROGRAMMING

# TURTLE BASICS

```
import turtle

# polygon function
# input:  accepts two integers
#         sides - defines how many sides we need to draw
#         length - defines the length of each side
def polygon(sides, length):
    for x in range(sides):# loop through each side
        # draw the side
        turtle.forward(length)
        # rotate the requisite number of degrees
        turtle.right(360/sides)

# set up a title
turtle.title(" BCO601 Python Programming ")

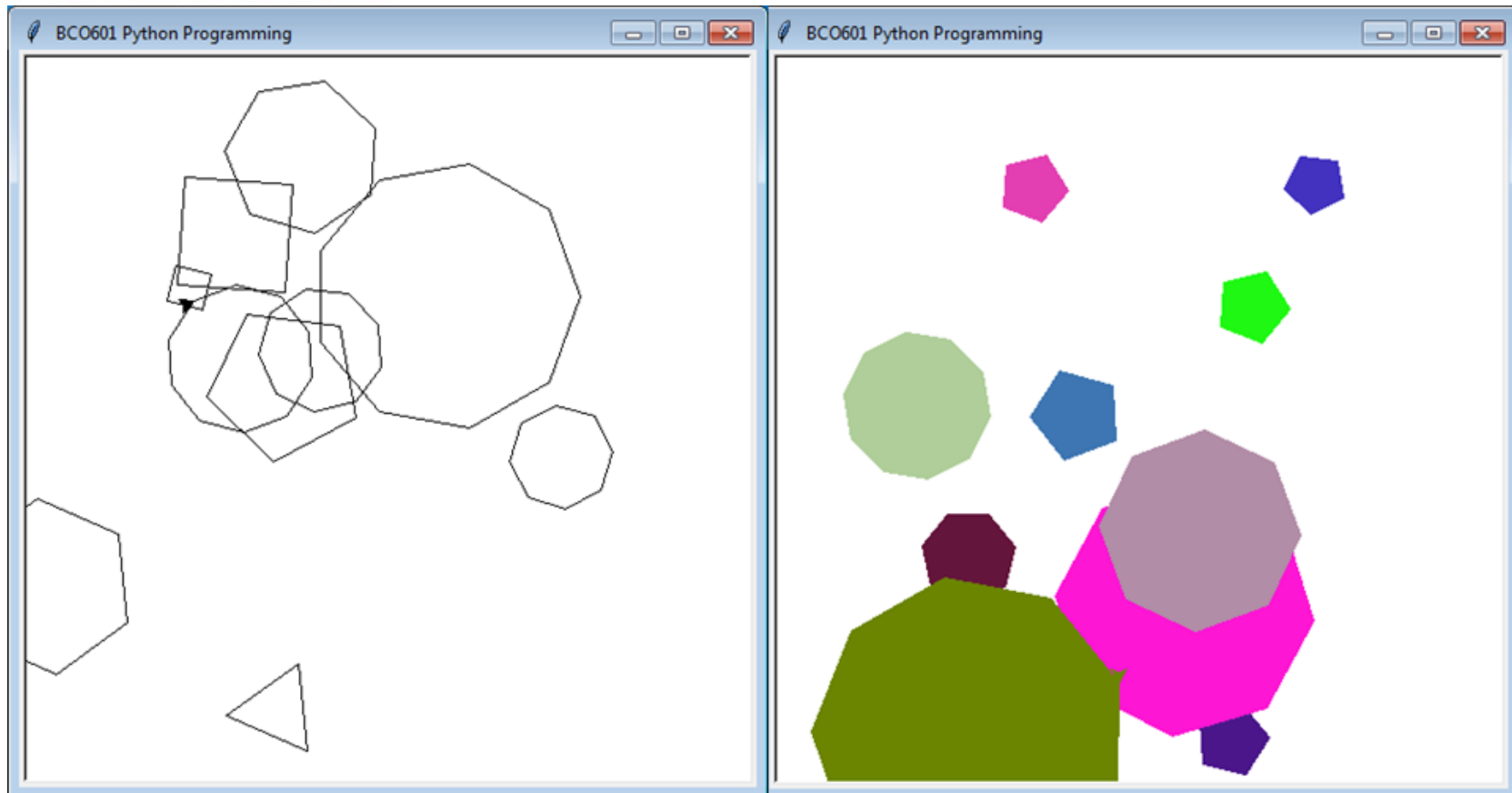
# set up your stage or canvas
turtle.setup(500, 500, 0, 0)

# Octagon - 8 sides
polygon(8, 50)

# let the user close the turtle window when they click
turtle.exitonclick()
```

# PYTHON PROGRAMMING

# TURTLE BASICS



```
def polygon(sides, length):  
    for x in range(sides):# loop through each side  
        # draw the side  
        turtle.forward(length)  
        # rotate the requisite number of degrees  
        turtle.right(360/sides)
```

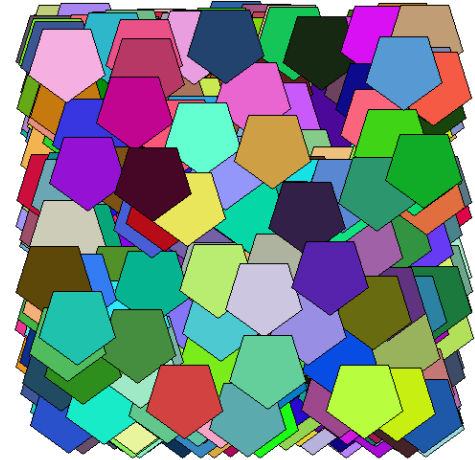
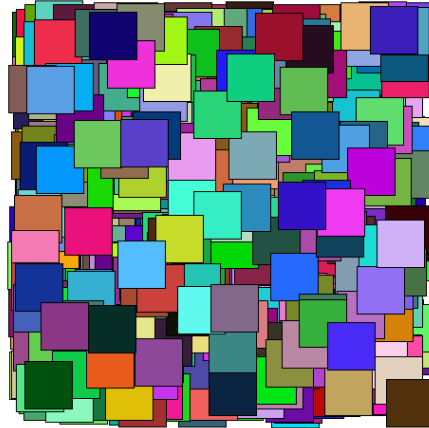
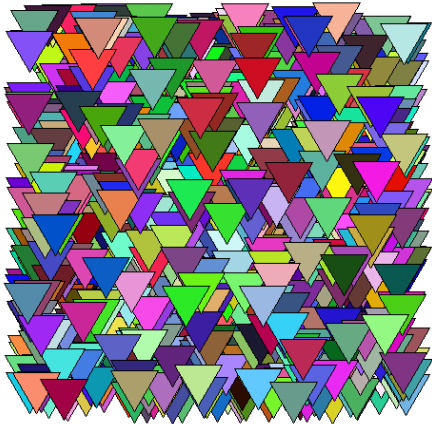
```
def besgen(): return polygon(5, 50)  
def kare(): return polygon(4, 50)  
def ucgen(): return polygon(3, 50)
```

```
def wrapper(func, *args): # The * next to args means
    func(*args)           # "take the rest of the
                           # parameters given and put them
                           # in a list called args

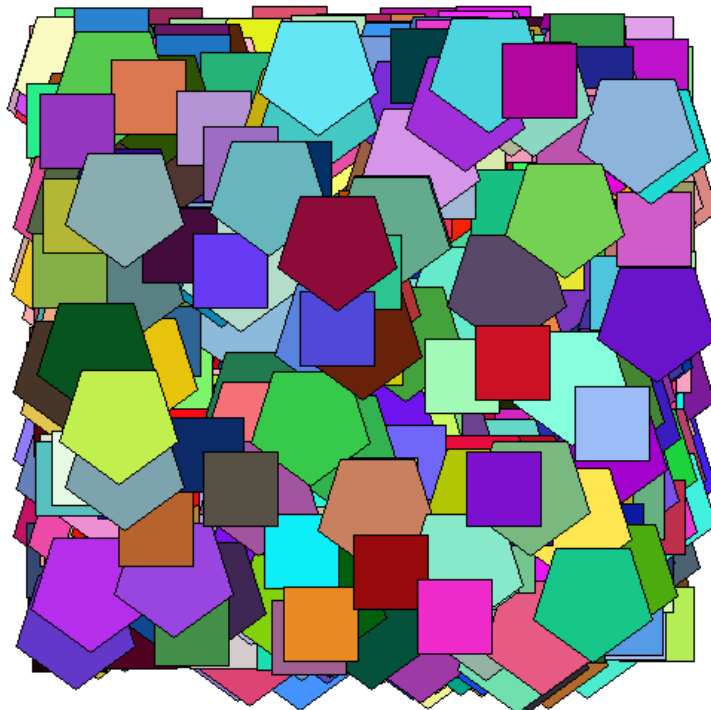
def polygon(sides, length):
    for x in range(sides): # loop through each side
        # draw the side
        turtle.forward(length)
        # rotate the requisite number of degrees
        turtle.right(360/sides)

# create wrapper functions
def besgen(): wrapper(polygon, 5, 50)
def kare(): wrapper(polygon, 4, 50)
def ucgen(): wrapper(polygon, 3, 50)
```

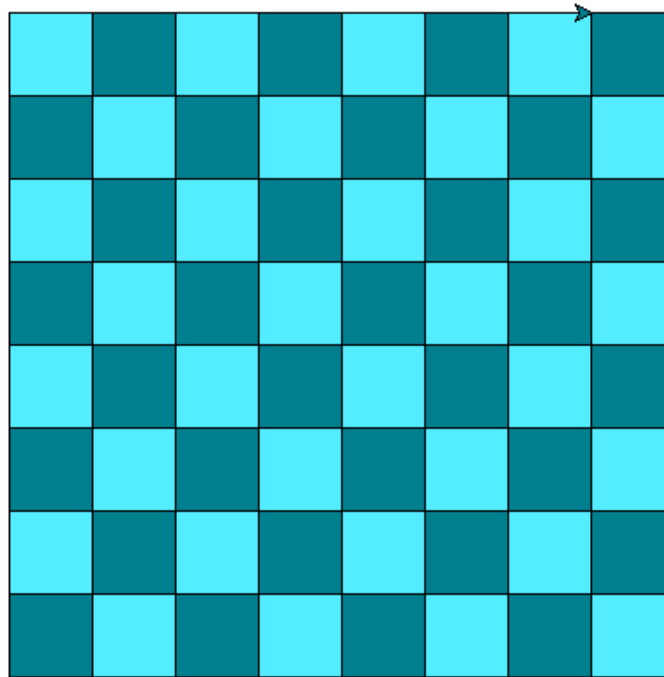
**Homework : Try to write randomly position shapes with random color.**



**Homework : Try to write randomly position random shapes with random color.**



**Homework : Try to write checker board with random color.**







```
turtle.shape(name=None)
```

Initially there are the following polygon shapes: “arrow”, “turtle”, “circle”, “square”, “triangle”, “classic”.

```
>>> turtle.shape()  
'classic'  
>>> turtle.shape("square")  
>>> turtle.shape()  
'square'
```

```
turtle.shapesize(stretch_wid=None, stretch_len=None, outline=None)
```

Return or set the pen's attributes x/y-stretchfactors and/or outline.

```
>>> turtle.shapesize(5, 5, 12)
>>> turtle.shapesize()
(5, 5, 12)
>>> turtle.shapesize(outline=8)
>>> turtle.shapesize()
(5, 5, 8)
```

```
turtle.onkey(fun, key)
```

Bind fun to key-release event of key. If fun is None, event bindings are removed.

```
>>> def f():  
...     fd(50)  
...     lt(60)  
...  
>>> screen.onkey(f, "Up")  
>>> screen.listen()
```

```
turtle.listen(xdummy=None, ydummy=None)
```

Set focus on TurtleScreen

```
turtle.tracer(n=None, delay=None)
```

Turn turtle animation on/off and set delay for update drawings. If n is given, only each n-th regular screen update is really performed. (Can be used to accelerate the drawing of complex graphics.)

```
>>> screen.tracer(8, 25)
>>> dist = 2
>>> for i in range(200):
...     fd(dist)
...     rt(90)
...     dist += 2
```

```
turtle.mainloop()
```

Starts event loop - calling Tkinter's mainloop function. Must be the last statement in a turtle graphics program.

```
>>> screen.mainloop()
```

```
turtle.bye()
```

Shut the turtlegraphics window.



# PYTHON PROGRAMMING

# TURTLE BASICS

```
from turtle import *
```

```
width, height = 800, 600  
maxx, maxy = width/2, height/2  
minx, miny = -maxx, -maxy  
startx, starty = 0, 0  
dx, dy = 10, 10
```

```
wn = Screen()  
box=Turtle()
```

```
def setup(width, height, startx, starty, shape, col, s):  
    box.penup()  
    box.shape(shape)  
    box.shapesize(*s)  
    box.color(col)  
    wn.setup(width, height, startx, starty)  
    wn.title("BCO601")  
    wn.bgcolor('grey')  
    wn.onkey(up, "Up")  
    wn.onkey(left, "Left")  
    wn.onkey(right, "Right")  
    wn.onkey(back, "Down")  
    wn.onkey(quitTurtles, "Escape")  
    wn.listen()  
    wn.mainloop()
```



```
#Event handlers
```

```
def up():  
    box.fd(45)
```

```
def left():  
    box.lt(45)
```

```
def right():  
    box.rt(45)
```

```
def back():  
    box.bk(45)
```

```
def quitTurtles():  
    wn.bye()
```

```
setup(width, height, startx, starty, "square", "red", (3,3,3))
```

# TURTLE BASICS

