Exposure CS 2021 for CS1

Chapter 9 Section 7-11 Slides

Modular Programming: Creating Procedures & Functions with Parameters

PowerPoint Presentation
created by:
Mr. John L. M. Schram
and Mr. Leon Schram
Authors of Exposure
Computer Science

Section 9.7

Procedures with a Single Argument & Parameter

Subroutine Calls With Arguments

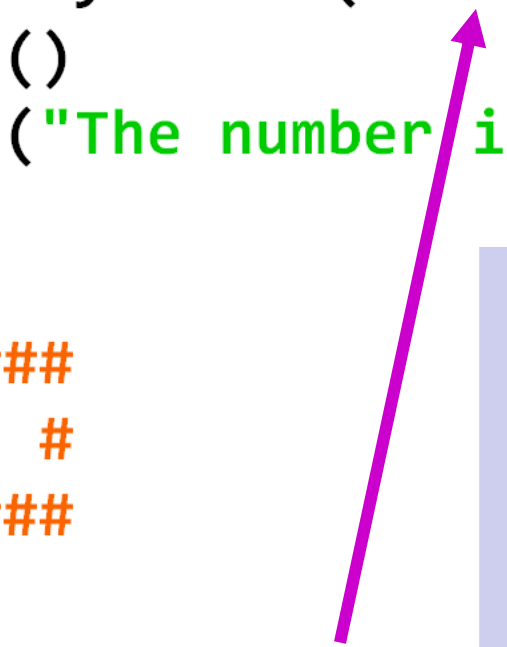
```
result1 = sqrt(100)
result2 = pow(2,5)
result3 = max(result1,result2)

setColor("red")
drawCircle(650,350,200)
delay(3000)
fillRectangle(100,200,900,600)
```

Subroutine Calls Without Arguments

```
displayName()
displayStreetAddress()
displayCityStateZip()

drawFloors()
drawRoof()
drawChimney()
drawDoor()
drawWindows()
```

```
 1 # ParameterProcedures01.py
 2 # This program passes the argument 100 to the
 3 # parameter <num> in procedure <displayNumber>
 4 # and then displays it.
 5
 6
 7 def displayNumber(num): # Procedure Heading
 8     print()
 9     print("The number is",num)
10
11
12 ############
13 #  MAIN  #
14 ############
15
16 displayNumber(100)      # Procedure Call
17
```

```
1  # ParameterProcedures01.py
2  # This program passes the argument 100 to the
3  # parameter <num> in procedure <displayNumber>
4  # and then displays it.
5
6
7  def displayNumber(num): # Procedure Heading
8      print()
9      print("The number is",num)
10
11
12 ###########
13 #  MAIN   #
14 ###########
15
16 displayNumber(100)
17
```



```
----jGRASP exec:

The number is 100

----jGRASP: opera
```

Arguments vs. Parameters

Subroutine Heading Example

```
def displayNumber(num):
```

Subroutine Call Example

```
displayNumber(100)
```

The *arguments* are all the values between the parentheses in the subroutine call.

Example: **`100`**

The *parameters* are all the variables between the parentheses in the subroutine heading.

Example: **`num`**

When you call a subroutine, a copy of each *argument* is passed to its corresponding *parameter*.

Argument/Parameter Disclaimer

If you learned another programming language before taking this course. You may have learned about *arguments* in the subroutine call and *parameters* in the subroutine heading. For the sake of this course, and to be consistent with the terminology used on the **PCEP** Certification Exam, just do the following translation in your head:

Other Languages	Python
Actual Parameters →	Arguments
Formal Parameters →	Parameters

```
 1 # ParameterProcedures02.py
 2 # This program demonstrates that an argument
 3 # can be: a constant, like <100> or <pi>,
 4 # a variable, like <x>, an expression with
 5 # constants and/or variables, like <20 + 30> or
 6 # <4 * x>, and a function call like <sqrt(225)>.
 7
 8
 9 from math import *
10
11
12 def displayNumber(num):
13     print()
14     print("The number is",num)
15
16
17
18 ##########
19 #  MAIN  #
20 ##########
21
22 x = 200
23 displayNumber(100)
24 displayNumber(x)
25 displayNumber(20 + 30)
26 displayNumber(4 * x)
27 displayNumber(pi)
28 displayNumber(sqrt(225))
```

```
 1 # ParameterProcedures02.py
 2 # This program demonstrates that an argument
 3 # can be: a constant, like <100> or <pi>,
 4 # a variable, like <x>, an expression with
 5 # constants and/or variables, like <20 + 30> or
 6 # <4 * x>, and a function call like <sqrt(225)>.
 7
 8
 9 from math import *
10
11
12 def displayNumber(num):
13     print()
14     print("The number is",num)
15
16
17
18 ##########
19 #  MAIN  #
20 ##########
21
22 x = 200
23 displayNumber(100)
24 displayNumber(x)
25 displayNumber(20 + 30)
26 displayNumber(4 * x)
27 displayNumber(pi)
28 displayNumber(sqrt(225))
```

```
 ----jGRASP exec: python Parameter

The number is 100

The number is 200

The number is 50

The number is 800

The number is 3.141592653589793

The number is 15.0

 ----jGRASP: operation complete.
```

Argument Formats

Arguments can be	Example:
• constants	**100** or **pi**
• variables	**x**
• expressions with constants only	**20 + 30**
• expressions with variables & constants	**x + 5**
• function calls	**sqrt(225)**

The Football Analogy

The Quarterback - The Argument

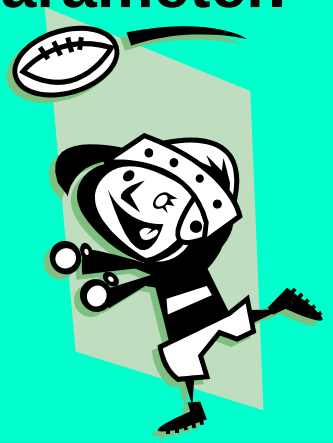
```
displayNumber(x)
```

The Football - A copy of the data

The argument passes a copy of the data to the parameter.

The Receiver - The Parameter

```
def displayNumber(num):
```



The Football Analogy

The Quarterback - The Argument

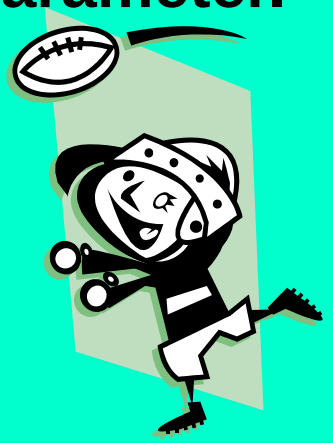
```
displayNumber(x)
```

The Football - A copy of the data

The argument passes a copy of the data to the parameter.

The Receiver - The Parameter

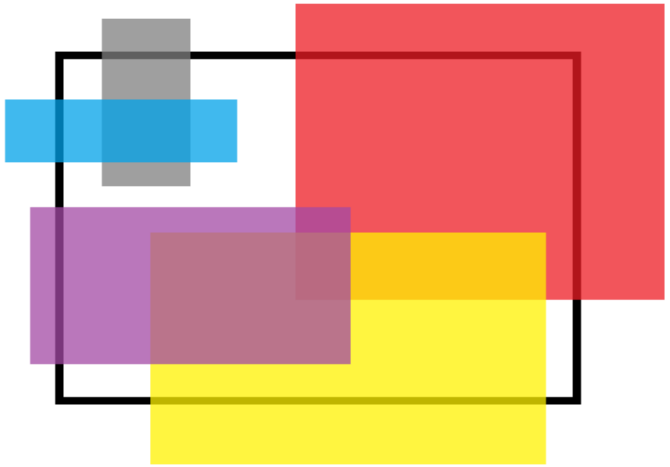
```
def displayNumber(num):
```



Section 9.8

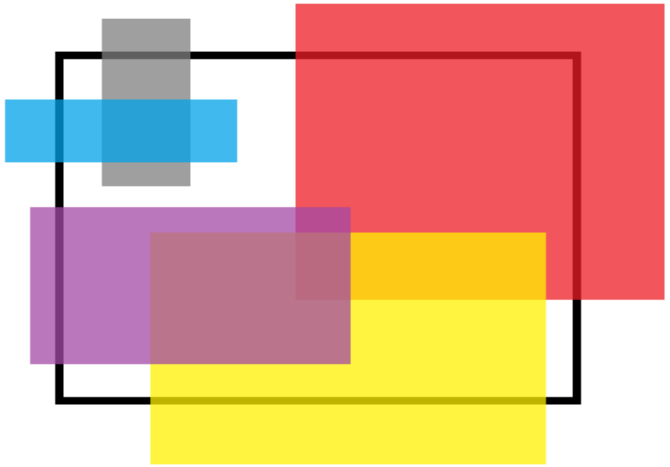
Procedures with Multiple Arguments & Parameters

```
1  # ParameterProcedures03.py
2  # This program demonstrates passing two arguments to a
3  # procedure.  Procedure <showRectangleArea> is called twice.
4  # In this case reversing the sequence of the arguments
5  # is not a problem.
6
7
8  def showRectangleArea(L,W):
9      area = L * W
10     print()
11     print("The rectangle area is",area)
12
13
14
15 ##########
16 #  MAIN  #
17 ##########
18
19 length = 100
20 width = 50
21 showRectangleArea(length,width)
22 showRectangleArea(width,length)
```



```
1  # ParameterProcedures03.py
2  # This program demonstrates
3  # procedure.  Procedure <sh
4  # In this case reversing th
5  # is not a problem.
6
7
8  def showRectangleAre
9      area = L * W
10     print()
11     print("The rectangle area is",area)
12
13
14
15 ##########
16 #  MAIN  #
17 ##########
18
19 length = 100
20 width = 50
21 showRectangleArea(length,width)
22 showRectangleArea(width,length)
```

```
----jGRASP exec: python Pa

The rectangle area is 5000

The rectangle area is 5000

----jGRASP: operation comp
```



```
 1 # ParameterProcedures04.py
 2 # This program demonstrates that argument sequence
 3 # matters. In this example procedure <showDifference>
 4 # will display different results when the calling
 5 # arguments are reversed.
 6
 7
 8 def showDifference(a,b):
 9     difference = a - b
10     print()
11     print("The difference is",difference)
12
13
14
15 ###########
16 #  MAIN   #
17 ###########
18
19 num1 = 100
20 num2 = 50
21 showDifference(num1,num2)
22 showDifference(num2,num1)
```

```
1  # ParameterProcedures04.py
2  # This program demonstrates
3  # matters. In this example p
4  # will display different res
5  # arguments are reversed.
6
7
8  def showDifference(a,b):
9      difference = a - b
10     print()
11     print("The difference is",difference)
12
13
14
15 ###########
16 #  MAIN   #
17 ###########
18
19 num1 = 100
20 num2 = 50
21 showDifference(num1,num2)
22 showDifference(num2,num1)
```

```
----jGRASP exec: pyth

The difference is 50

The difference is -50

----jGRASP: operation
```

100 − 50 = 50

50 − 100 = −50

Argument Sequence Matters

The first argument passes information to the first parameter.

The second argument passes information to the second parameter.

Arguments placed out of sequence may result in *syntax errors*, *run-time errors* or *logic errors*.

```
1  # ParameterProcedures05.py
2  # This program demonstrates that argument data types also
3  # matter. In this example 2 string arguments were passed
4  # to procedure <showDifference>.  This does not work as
5  # "subtraction" is something that only works with numbers.
6
7
8  def showDifference(a,b):
9      difference = a - b
10     print()
11     print("The difference is",difference)
12
13
14
15 ###########
16 #  MAIN   #
17 ###########
18
19 num1 = "John"
20 num2 = "Smith"
21 showDifference(num1,num2)
22 showDifference(num2,num1)
```

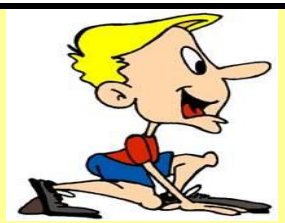
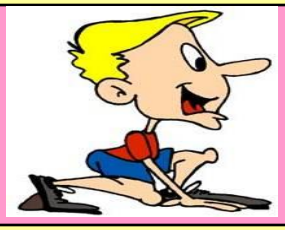
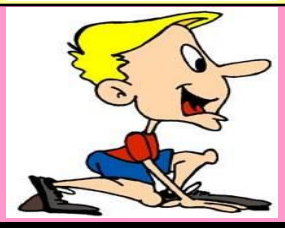
```
1  # ParameterProcedures05.py
2  # This program demonstrates that argument data types also
3  # matter. In this example 2 string arguments were passed
4  # to procedure <showDifference>.  This does not work as
5  # "subtraction" is something that only works with numbers.
6
7
8  def showDifference(a,b):
9      difference = a - b
10     print()
11     print("The difference is",difference)
12
13
14
15 ##########
16 #  MAIN  #
17 ##########
18
19 num1 = "John"
20 num2 = "Smith"
21 showDifference(num1,num2)
22 showDifference(num2,num1)
```

```
 ----jGRASP exec: python ParameterProc
Traceback (most recent call last):
File "ParameterProcedures05.py",
line 21, in <module>
    showDifference(num1,num2)
  File "ParameterProcedures05.py",
  line 9, in showDifference
    difference = a - b
TypeError: unsupported operand type(s)
for -: 'str' and 'str'

 ----jGRASP wedge2: exit code for proc
 ----jGRASP: operation complete.
```

```
1  # ParameterProcedures06.py
2  # This program demonstrates that different data types
3  # can be passed to the same procedure.
4  # The <showStudentInfo> procedure is called 3 times.
5  # The first two procedure calls are proper.
6  # The third one has the arguments out of order
7  # which causes strange output.
8
9
10 def showStudentInfo(name, age, gpa, inState):
11     print()
12     print("Student Information:")
13     print("Name:      ",name)
14     print("Age:       ",age)
15     print("GPA:       ",gpa)
16     print("In-State:  ",inState)
17
18
19
20 ###########
21 #  MAIN   #
22 ###########
23
24 showStudentInfo("John Smith", 22, 2.875, True)
25 showStudentInfo("Suzy Brown", 29, 3.999, False)
26 showStudentInfo(1.763, True, "Tom Jones", 27)
```

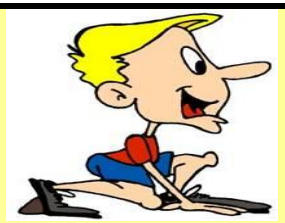
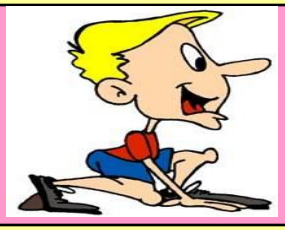
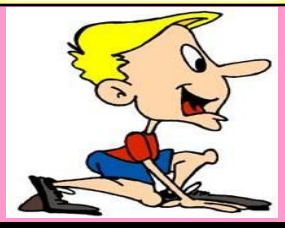
```
  ----jGRASP exec: python ParameterProcedures06

Student Information:
Name:       John Smith
Age:        22
GPA:        2.875
In-State:   True

Student Information:
Name:       Suzy Brown
Age:        29
GPA:        3.999
In-State:   False

Student Information:
Name:       1.763
Age:        True
GPA:        Tom Jones
In-State:   27

  ----jGRASP: operation complete.
```

The Track Relay Analogy – Race 1

US		US	
GB		GB	
FR		FR	
NL			

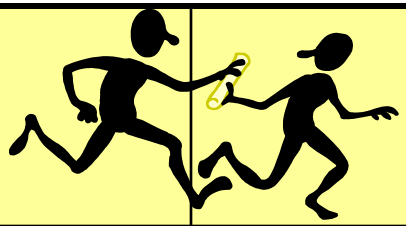
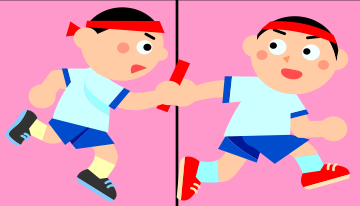
The second runner from the Netherlands is missing.
The number of arguments and parameters do not match.

The Track Relay Analogy – Race 2

US		US	
GB		GB	
FR		NL	
NL		FR	

The second runners from the Netherlands and France are in the wrong lane. ***The parameters are not in the same order as the arguments. They must correspond.***

The Track Relay Analogy – Race 3

US (John)		**US (Greg)**
GB (Charles)		**GB (William)**
FR (Gerald)		**FR (Louis)**
NL (Hans)		**NL (Hans)**

The runners are in proper staring position.
The parameters correspond.
The fact that there are 2 people from the Netherlands with the same name is not a problem.

Important Rules About Using Subroutines with Parameters

The number of *arguments* (values in the subroutine call) must match the number of *parameters* (variables in the subroutine heading).

The corresponding arguments must match the intended data type of the parameters.

The sequence of the arguments must match the sequence of the parameters.

The identifiers of the arguments may be the same as or may be different from the identifiers of the parameters.

Section 9.9

Functions with a Single Parameter

```python
# Functions01.py
# This program introduces a "function" with one parameter.
# Function <getNextNumber> returns the next integer after
# the value passed to its parameter.
# NOTE: A function is a subroutine that returns a value.
# A procedure is a subroutine that does not return a value.


from random import randint


def getNextNumber(current):
    next = current + 1
    return next



###########
#  MAIN   #
###########

for k in range(10):
    randNum = randint(10,99)
    print()
    print("Random number:",randNum)
    print("Next number:  ",getNextNumber(randNum))
```

```
 1 # Functions01.py
 2 # This program introduces a "functio
 3 # Function <getNextNumber> returns t
 4 # the value passed to its parameter.
 5 # NOTE: A function is a subroutine t
 6 # A procedure is a subroutine that d
 7
 8
 9 from random import randint
10
11
12 def getNextNumber(current):
13     next = current + 1
14     return next
15
16
17
18 ###########
19 #  MAIN   #
20 ###########
21
22 for k in range(10):
23     randNum = randint(10,99)
24     print()
25     print("Random number:",randNum)
26     print("Next number:   ",getNextNumber(randNum))
```

```
----jGRASP exec:

Random number:  93
Next number:    94

Random number:  36
Next number:    37

Random number:  47
Next number:    48

Random number:  29
Next number:    30

Random number:  77
Next number:    78

Random number:  20
Next number:    21

Random number:  78
Next number:    79

Random number:  11
Next number:    12

Random number:  58
Next number:    59

Random number:  76
Next number:    77

----jGRASP: opera
```

```
 1 # Functions02.py
 2 # This example returns a Boolean value, which is used
 3 # frequently to check for correct user keyboard input.
 4 # NOTE: This program also demonstrates the true purpose of
 5 # Boolean variables.  They make the program more readable.
 6
 7
 8 def checkPIN(pin):
 9     if pin == 1234:
10         return True
11     else:
12         return False
13
14
15
16 ###########
17 #  MAIN   #
18 ###########
19
20 correctPIN = False
21 while(not correctPIN):
22     pin = eval(input("\nEnter your 4 digit PIN  -->  "))
23     correctPIN = checkPIN(pin)
24     if not correctPIN:
25         print("\nIncorrect PIN.  Please try again.")
26
27 print("\nYou have successfully logged in.")
28 print("Select your bank transaction:")
```

```
 1 # Functions02.py
 2 # This example returns a
 3 # frequently to check for
 4 # NOTE: This program also
 5 # Boolean variables.  The
 6
 7
 8 def checkPIN(pin):
 9     if pin == 1234:
10         return True
11     else:
12         return False
13
14
15
16 ###########
17 #  MAIN   #
18 ###########
19
20 correctPIN = False
21 while(not correctPIN):
22     pin = eval(input("\nEnter your 4 digit PIN  -->  "))
23     correctPIN = checkPIN(pin)
24     if not correctPIN:
25         print("\nIncorrect PIN.  Please try again.")
26
27 print("\nYou have successfully logged in.")
28 print("Select your bank transaction:")
```

```
----jGRASP exec: python Functions02

Enter your 4 digit PIN  -->  9876

Incorrect PIN.  Please try again.

Enter your 4 digit PIN  -->  5555

Incorrect PIN.  Please try again.

Enter your 4 digit PIN  -->  1234

You have successfully logged in.
Select your bank transaction:

----jGRASP: operation complete.
```

Section 9.10

Functions with Multiple Parameters

```
 1 # Functions03.py
 2 # This program has 2 "add" subroutines.
 3 # <add1> is a procedure.  <add2> is a function.
 4 # The purpose is to demonstrate the differences
 5 # between these 2, which are:
 6 # 1) They are called differently.
 7 # 2) Functions end with a <return> command.
 8
 9
10 def add1(n1,n2):
11     sum = n1 + n2
12     print(n1,"+",n2,"=",sum)
13
14
15 def add2(n1,n2):
16     sum = n1 + n2
17     return sum
18
19
20 ###########
21 #  MAIN   #
22 ###########
23
24 num1 = 1000
25 num2 = 100
26 print()
27 add1(num1,num2)
28 print(num1,"+",num2,"=",add2(num1,num2))
```

```
----jGRASP exec:


1000 + 100 = 1100
1000 + 100 = 1100


----jGRASP: opera
```

Procedures and Functions

Procedure Example

```
def add1(n1,n2):
    sum = n1 + n2
    print(n1,"+",n2,"=",sum)
```

Procedures do not return a value.

Function Example

```
def add2(n1,n2):
    sum = n1 + n2
    return sum
```

Functions do return a value. All functions must have a **return** statement, which is usually the last statement in the function.

```
 1 # Functions04.py
 2 # This program demonstrates a 4 "function" calculator.
 3 # NOTE: While it may be good program design to put all
 4 # of these functions in a separate library, most of the
 5 # examples in the book will continue to put the entire
 6 # program in a single file for the same of simplicity.
 7
 8
 9 def add(n1,n2):
10     return n1 + n2
11
12
13 def subtract(n1,n2):
14     return n1 - n2
15
16
17 def multiply(n1,n2):
18     return n1 * n2
19
20
21 def divide(n1,n2):
22     return n1 / n2
23
24
25
26 ###########
27 #  MAIN   #
28 ###########
29
30 num1 = 1000
31 num2 = 100
32 print()
33 print(num1,"+",num2,"=",add(num1,num2))
34 print(num1,"-",num2,"=",subtract(num1,num2))
35 print(num1,"*",num2,"=",multiply(num1,num2))
36 print(num1,"/",num2,"=",divide(num1,num2))
```

```
----jGRASP exec: pyth


1000 + 100 = 1100
1000 - 100 = 900
1000 * 100 = 100000
1000 / 100 = 10.0


----jGRASP: operation
```

```python
# Functions05.py
# This program demonstrates 3 proper ways
# and 1 improper way to call a function.


def add(n1,n2):
    return n1 + n2



###########
#  MAIN   #
###########

print()
print("Sum:",add(200,300))

sum = add(400,500)
print("Sum:",sum)

checking = 600
savings  = 700
if add(checking,savings) <= 0:
    print("We are broke!")
else:
    print("Let's go shopping!")

add(800,900)   # Essentially does nothing
```

```
----jGRASP exec:


Sum: 500
Sum: 900
Let's go shopping!


----jGRASP: opera
```

Section 9.11

Creating Subroutines from Other Subroutines

Bigger Tools & Bigger Stuff

Consider the *Tool* analogy again.

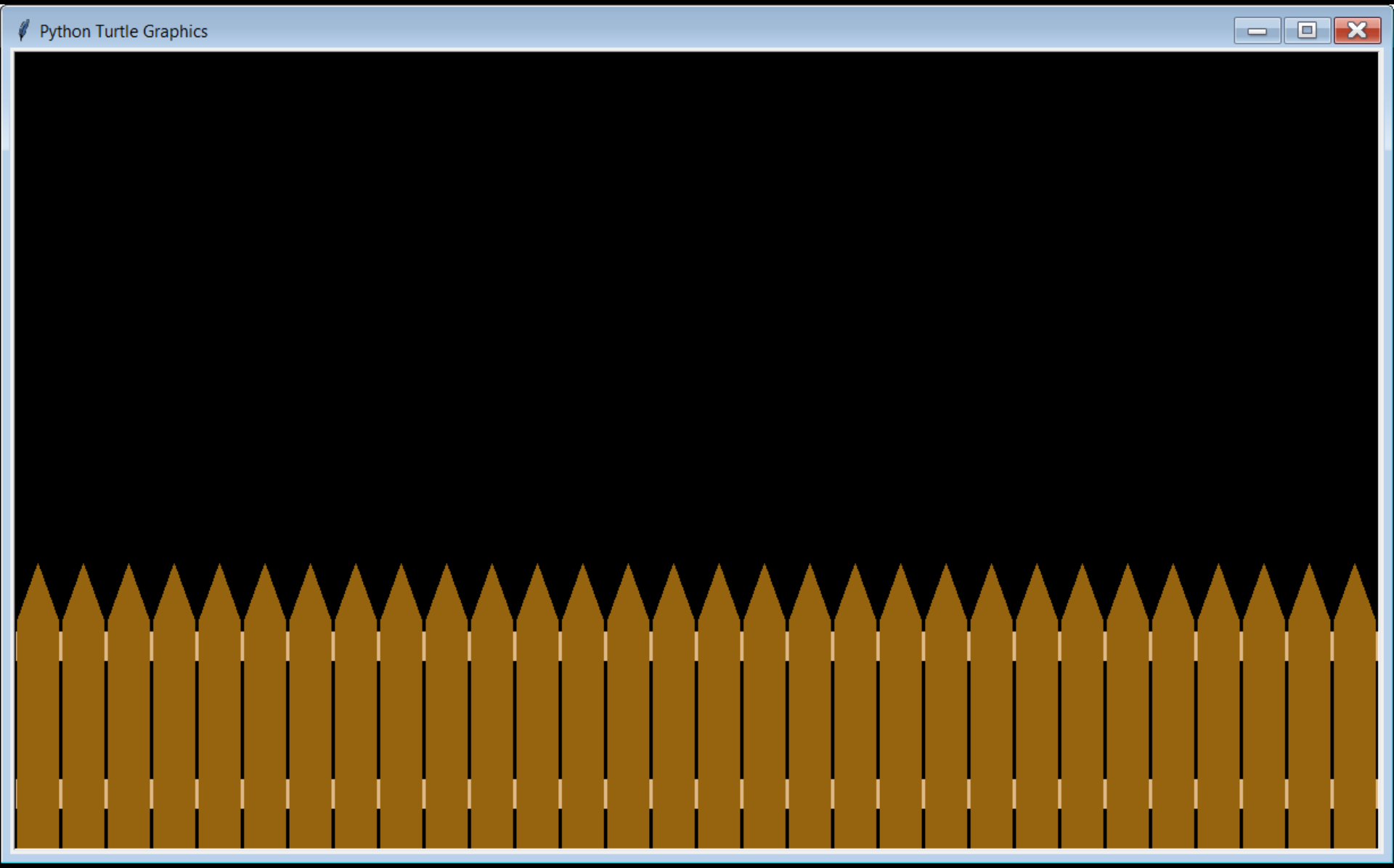
Essentially, we make tools so that we can make the stuff we want.

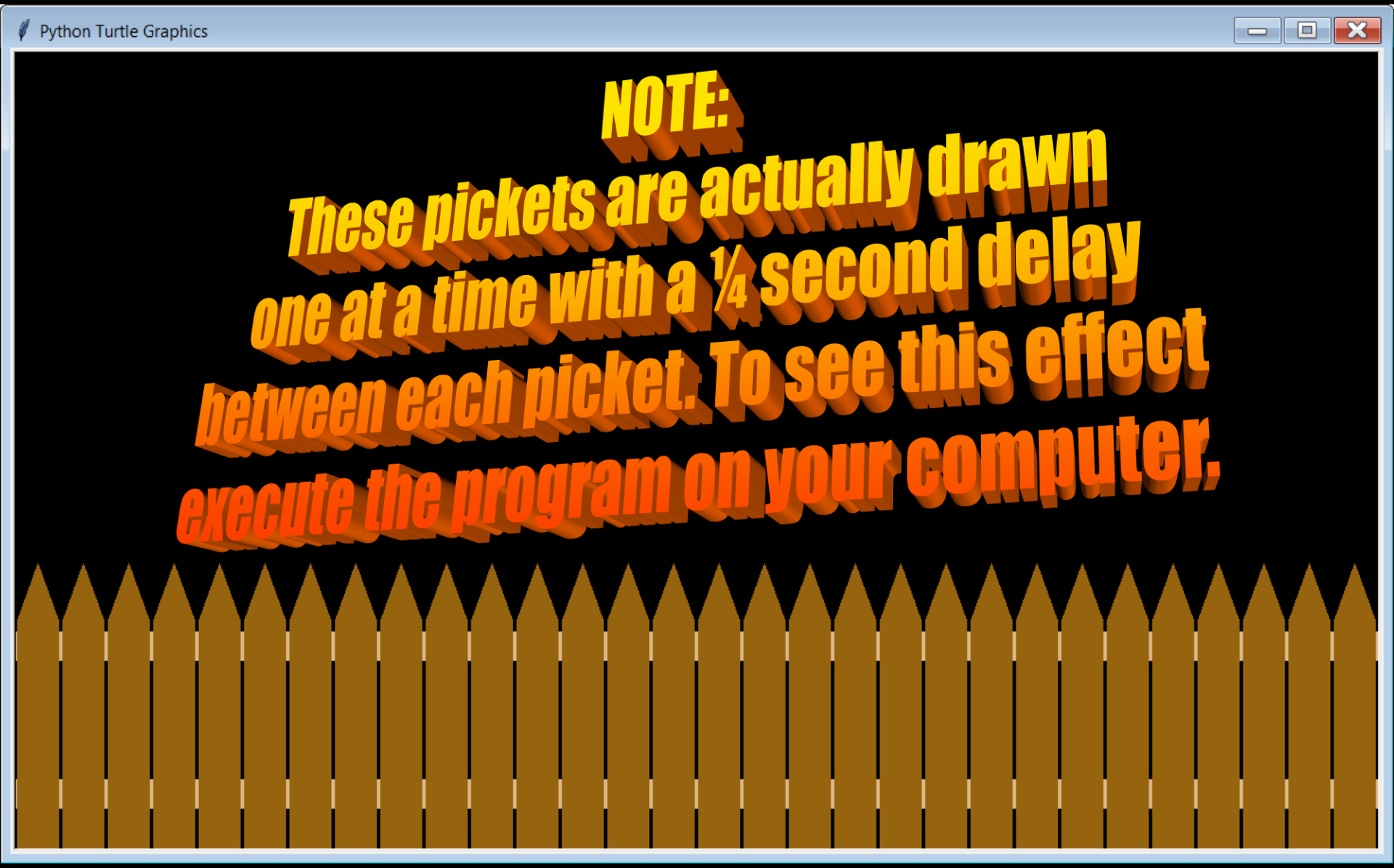
What if we use some tools to make a bigger tool, like a crane?

Now we can make bigger stuff, like a car or even a building.

Once subroutines are created, they can be used to create bigger, more powerful subroutines.

```
 1 # SubFromSub01.py
 2 # This program demonstrates a <picket> procedure
 3 # that will be used to help draw a fence.
 4
 5
 6
 7 from Graphics import *
 8
 9
10 def drawPicket(x):
11     fillPolygon([x,700,x,500,x+18,450,x+36,500,x+36,700])
12     delay(250)  # delay for 1/4 of a second
13
14
15
16 ###########
17 #  MAIN   #
18 ###########
19
20 beginGrfx(1200,700)
21
22 setBackground("black")
23 setColor("burlywood")
24 fillRectangle(0,510,1200,535)
25 fillRectangle(0,640,1200,665)
26
27 setColor("brown")
28 for x in range(2,1200,40):
29     drawPicket(x)
30
31 endGrfx()
```


Python Turtle Graphics


Python Turtle Graphics
NOTE:
These pickets are actually drawn
one at a time with a ¼ second delay
between each picket. To see this effect
execute the program on your computer.

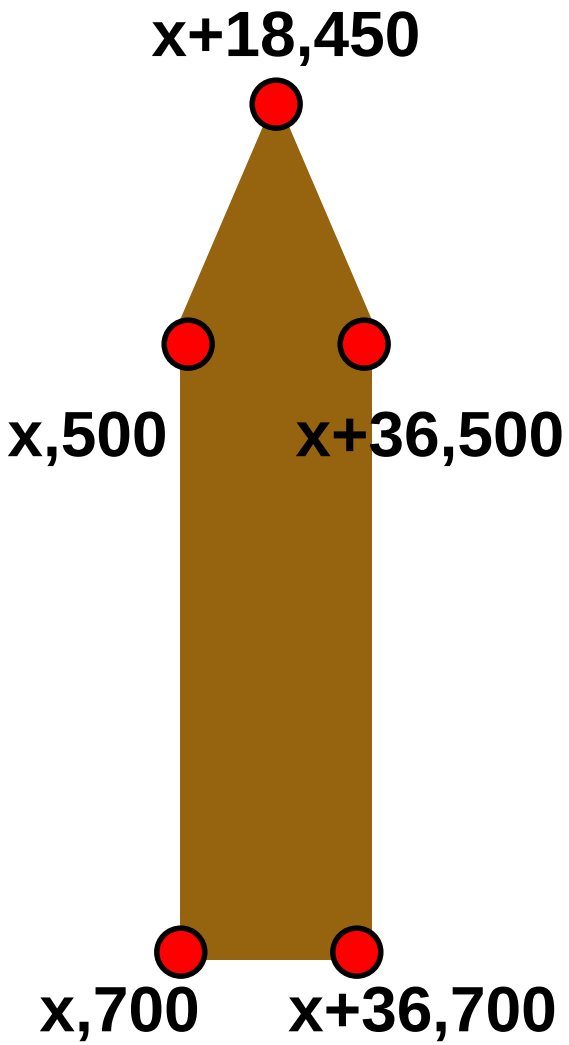
The Logic of drawPicket

drawPicket(x)

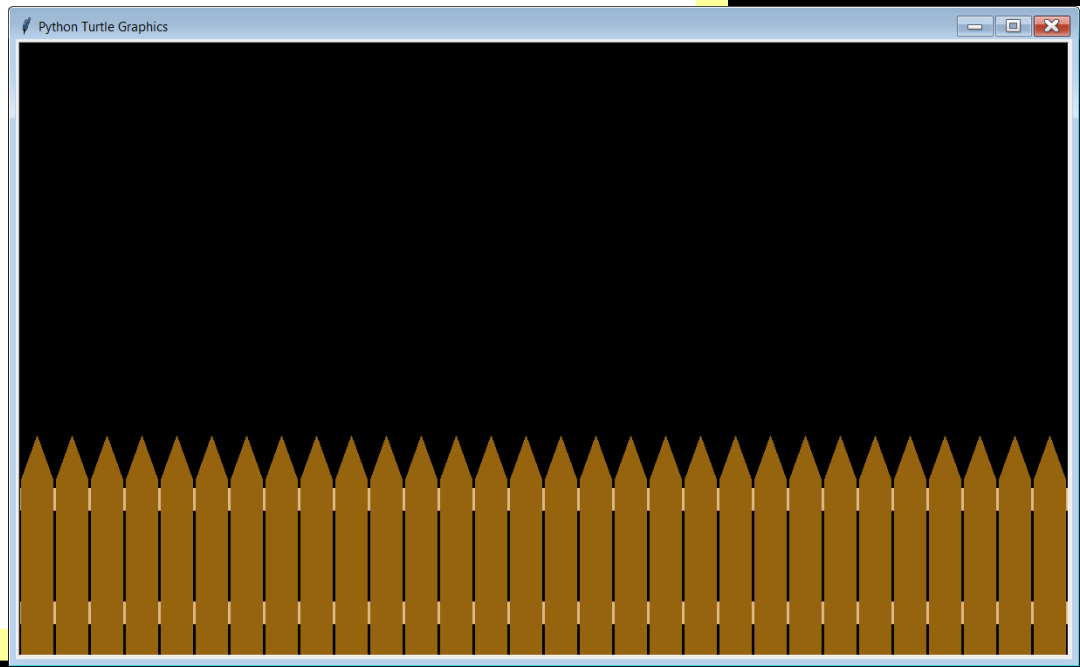
x is the horizontal value of the bottom left corner of the picket.

The y (vertical) value of the bottom left corner is always 700 since all pickets will be at the bottom of the graphics window.

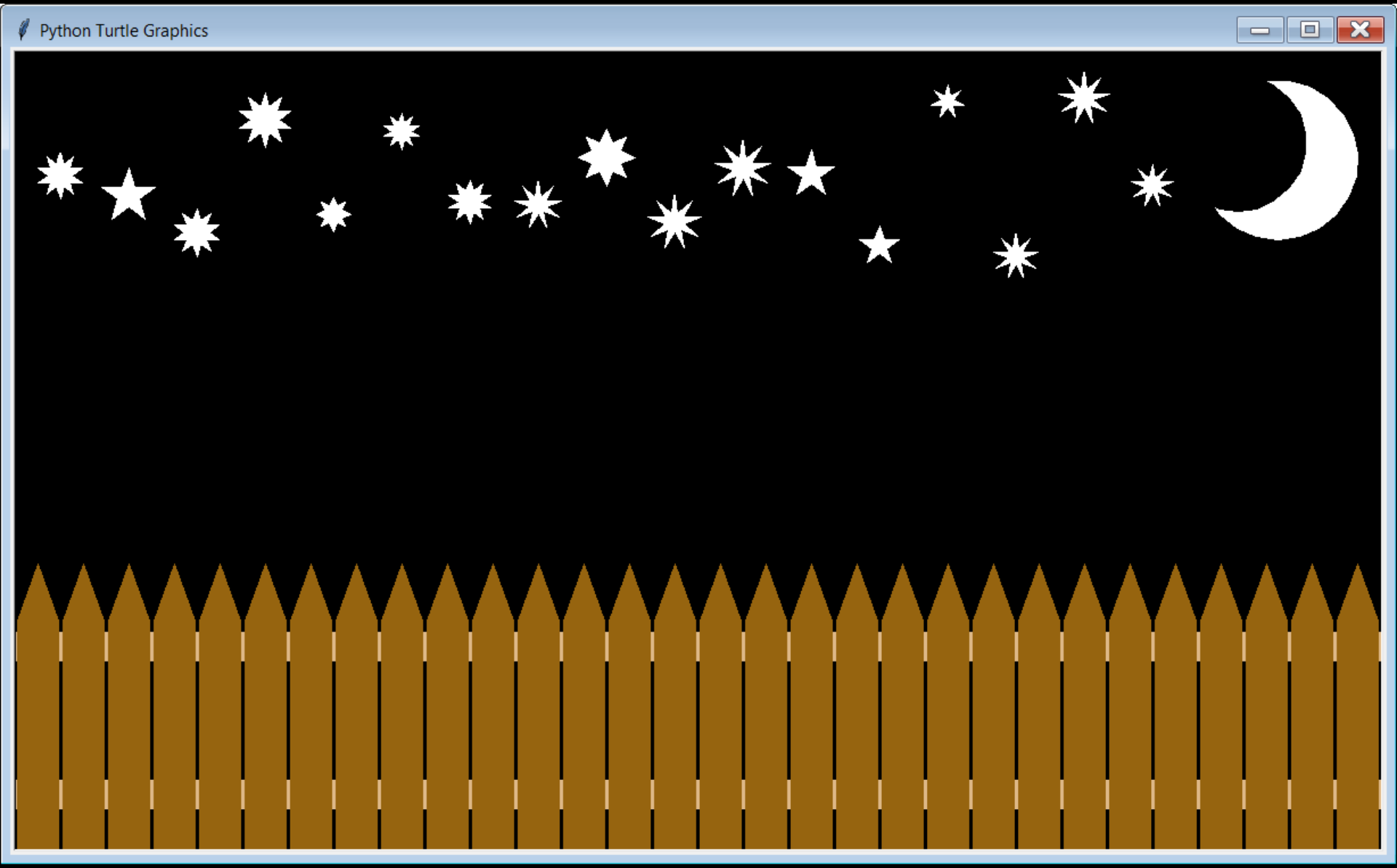
The other 4 coordinates of the picket are relative to the point (x,700).



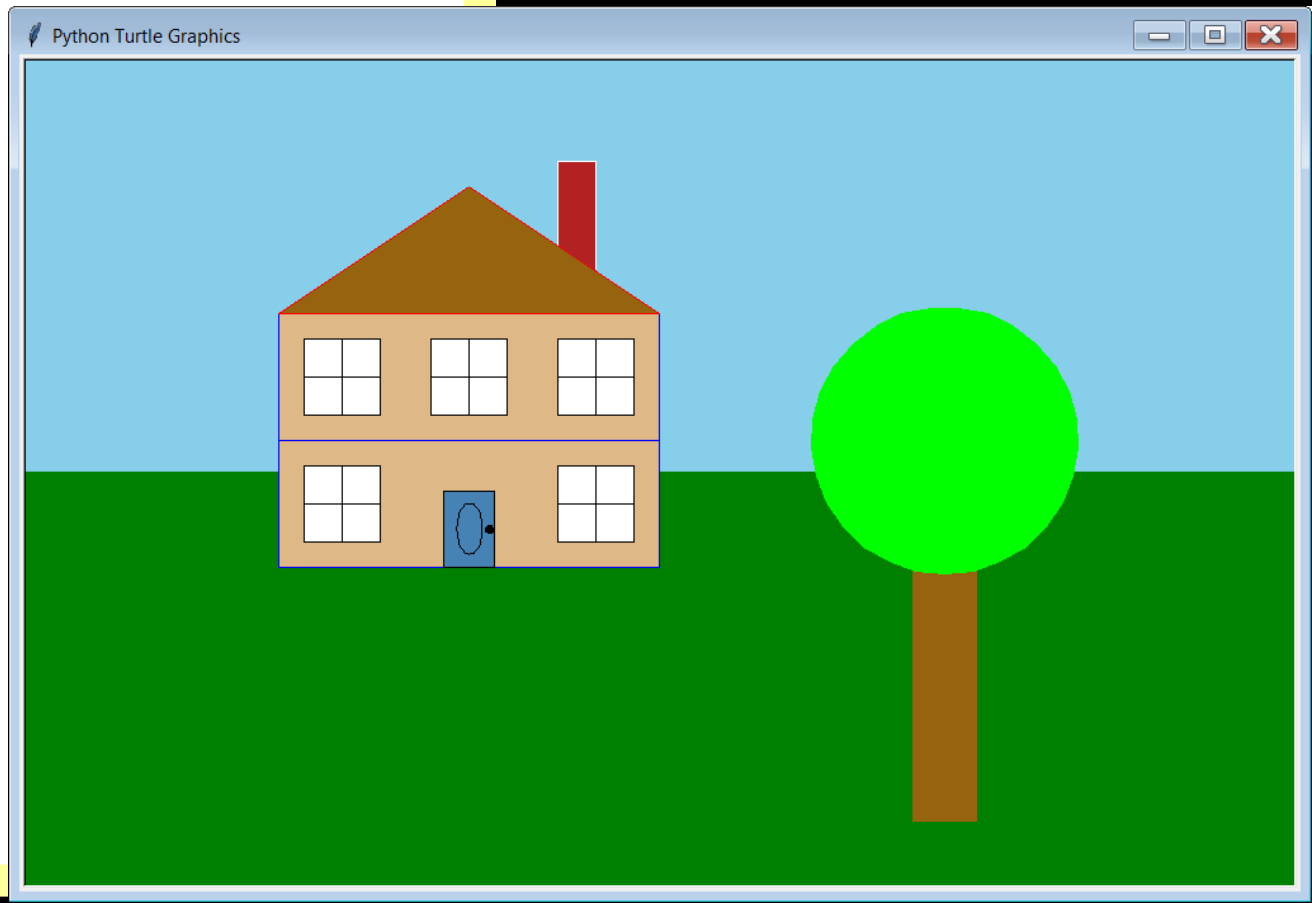
```
 1 # SubFromSub02.py
 2 # This program uses the <drawPicket> procedure
 3 # to create the <drawFence> procedure.
 4
 5
 6 from Graphics import *
 7
 8
 9 def drawPicket(x):
10     fillPolygon([x,700,x,500,x+18,450,x+36,500,x+36,700])
11     delay(250)  # delay for 1/4 of a second
12
13
14 def drawFence():
15     # cross beams
16     setColor("burlywood")
17     fillRectangle(0,510,1200,535)
18     fillRectangle(0,640,1200,665)
19     # pickets
20     setColor("brown")
21     for x in range(2,1200,40):
22         drawPicket(x)
23
24
25 ##########
26 #  MAIN  #
27 ##########
28
29 beginGrfx(1200,700)
30
31 setBackground("black")
32 drawFence()
33
34 endGrfx()
```



```
# SubFromSub03.py
# This program adds procedure <drawNightSky>,
# which is created using procedures <drawMoon>
# and <drawRandomStar>.



from Graphics import *
from random import randint


def drawPicket(x):
    fillPolygon([x,700,x,500,x+18,450,x+36,500,x+36,700])
    delay(250)  # delay for 1/4 of a second


def drawFence():
    # cross beams
    setColor("burlywood")
    fillRectangle(0,510,1200,535)
    fillRectangle(0,640,1200,665)
    # pickets
    setColor("brown")
    for x in range(2,1200,40):
        drawPicket(x)


def drawMoon():
    setColor("white")
    fillCircle(1110,95,70)
    setColor("black")
    fillCircle(1075,80,60)



def drawRandomStar(x):
    y = randint(40,200)
    radius = randint(15,25)
    points = randint(5,10)
    fillStar(x,y,radius,points)
    delay(250)


def drawNightSky():
    setBackground("black")
    drawMoon()
    setColor("white")
    for x in range(40,1001,60):
        drawRandomStar(x)



##########
#  MAIN  #
##########

beginGrfx(1200,700)

drawNightSky()
drawFence()

endGrfx()

```


Python Turtle Graphics

```
  1 # SubFromSub04.py
  2 # This program program revisits the "House"
  3 # from program GraphicsProcedures08.py
  4 # Now instead of 9 separate procedure calls,
  5 # we have 3: <drawBackground>, <drawHouse>
  6 # and <drawTree>.  In a sense, the program
  7 # now resembles a multi-level outline.
  8
  9
 10 from Graphics import *
 11
 12
 13 # House, Tree and Background procedures
    (same as before)

 99 # Procedures made from other procedures
100
101 def drawBackground():
102     drawSky()
103     drawGrass()
104
105
106 def drawHouse():
107     drawFloors()
108     drawRoof()
109     drawChimney()
110     drawDoor()
111     drawWindows()
112
113
114 def drawTree():
115     drawTrunk()
116     drawLeaves()
117
118
119
120 ##########
121 #  MAIN  #
122 ##########
123
124 beginGrfx(1000,650)
125
126 drawBackground()
127 drawHouse()
128 drawTree()
129
130 endGrfx()
```



```
 1 # SubFromSub05.py
 2 # This program demonstrates that a function can also be
 3 # created from another function. In this case, the <gcf>
 4 # function is used to create the <lcm> function.
 5 # Example: LCM(A,B) = A * B / GCF(A,B)
 6 # This program also demonstrates that you can call a
 7 # function if you know what it does and what parameters
 8 # it requires, but you do not necessarily need to know
 9 # how it works. This program also demonstrates how you
10 # can break up a very long command and make it "wrap"
11 # to the next line by using a backslash (\).
12
13
14 def gcf(a,b):
15     while True:
16         rem = a % b
17         if rem == 0:
18             return b
19         else:
20             a,b = b,rem
21
22
23 def lcm(a,b):
24     return a * b // gcf(a,b)
25
26
27
28 ##########
29 #  MAIN  #
30 ##########
31
32 print()
33 num1 = eval(input("Enter 1st number  -->  "))
34 num2 = eval(input("Enter 2nd number  -->  "))
35 print()
36 print("The Greatest Common Factor of",num1, \
37 "and",num2,"is",gcf(num1,num2))
38 print()
39 print("The Least Common Multiple of",num1, \
40 "and",num2,"is",lcm(num1,num2))
```

```
 1 # SubFromSub05.py
 2 # This program demonstr
 3 # created from another
 4 # function is used to c
 5 # Example: LCM(A,B) = A
 6 # This program also dem
 7 # function if you know
 8 # it requires, but you
 9 # how it works. This pr
10 # can break up a very l
11 # to the next line by u
12
13
14 def gcf(a,b):
15     while True:
16         rem = a % b
17         if rem == 0:
18             return b
19         else:
20             a,b = b,rem
21
22
23 def lcm(a,b):
24     return a * b // gcf(a,b)
25
26
27
28 ##########
29 #  MAIN  #
30 ##########
31
32 print()
33 num1 = eval(input("Enter 1st number  -->  "))
34 num2 = eval(input("Enter 2nd number  -->  "))
35 print()
36 print("The Greatest Common Factor of",num1, \
37 "and",num2,"is",gcf(num1,num2))
38 print()
39 print("The Least Common Multiple of",num1, \
40 "and",num2,"is",lcm(num1,num2))
```

```
----jGRASP exec: python SubFromSub05.py

Enter 1st number  -->  4000
Enter 2nd number  -->  625

The Greatest Common Factor of 4000 and 625 is 125

The Least Common Multiple of 4000 and 625 is 20000

----jGRASP: operation complete.
```