Exposure CS 2021 for CS1

Chapter 6 Section 1-3 Slides

More Python Libraries: math Library Functions

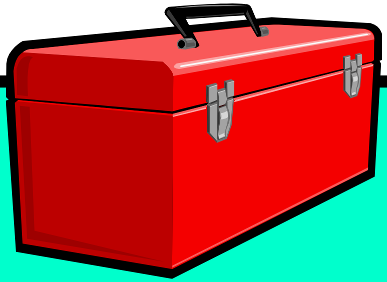
PowerPoint Presentation created by: Mr. John L. M. Schram and Mr. Leon Schram Authors of Exposure Computer Science

Section 6.2

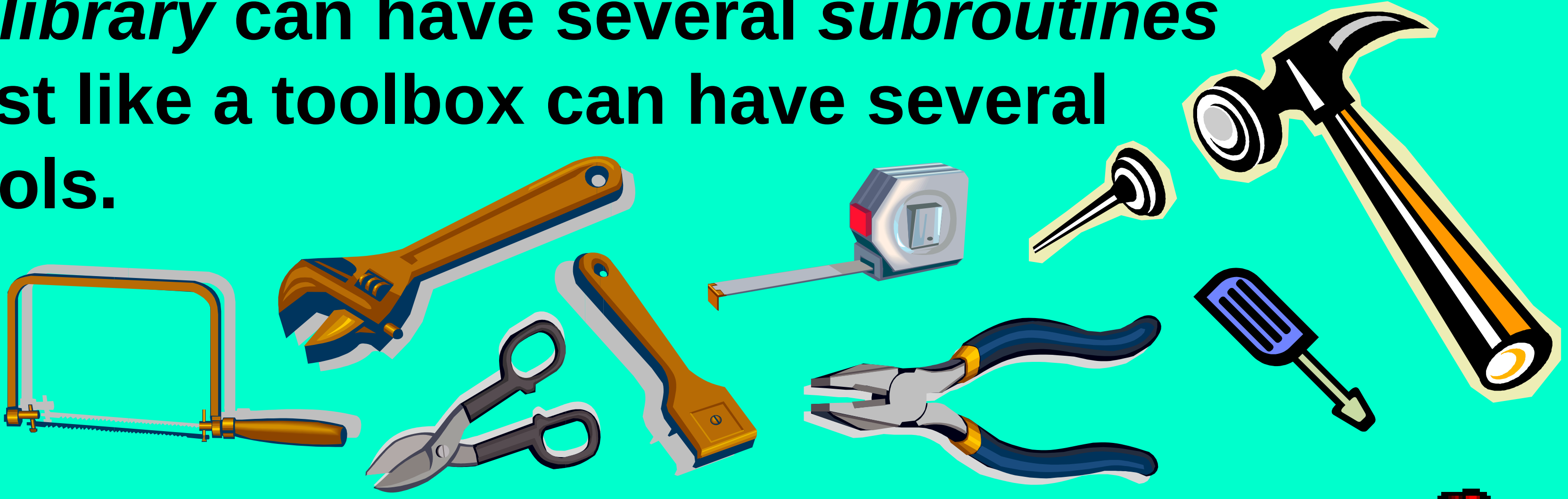
Library Components

The Toolbox Analogy

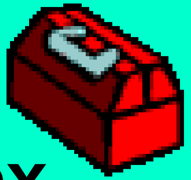
A *library* is like a toolbox.



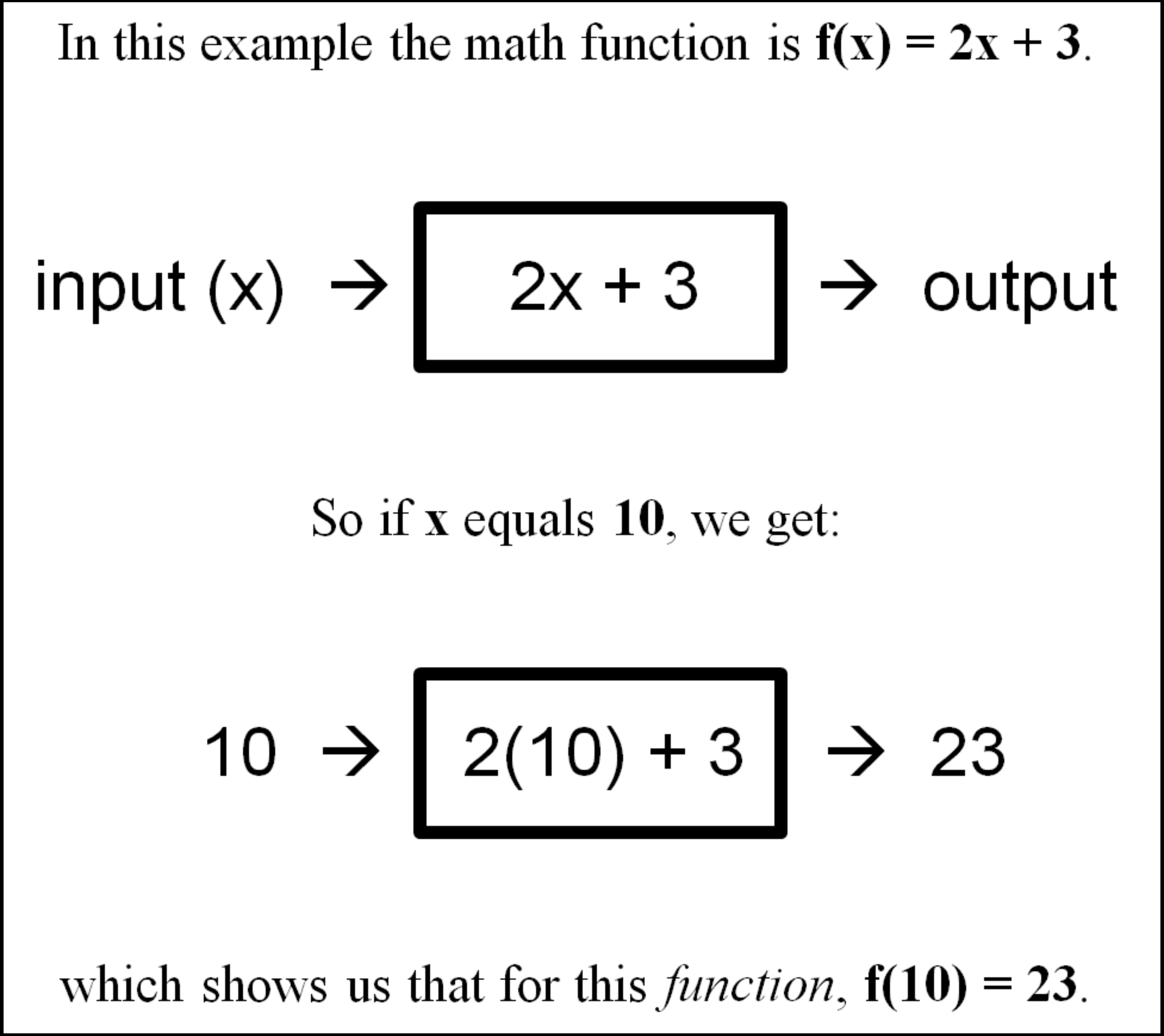
A *library* can have several *subroutines* just like a toolbox can have several tools.



Before any of these tools can be used, you must first find (**import**) their specific toolbox.



Review of a Mathematical Function



In this example the math function is **f(x) = 2x + 3**.

$$\text{input } (x) \rightarrow \boxed{2x + 3} \rightarrow \text{output}$$

So if **x** equals **10**, we get:

$$10 \rightarrow \boxed{2(10) + 3} \rightarrow 23$$

which shows us that for this *function*, **f(10) = 23**.

Subroutines, Functions and Procedures

A *subroutine* is a series of programming commands that performs a specific task.

A *function* is a subroutine that returns a value.

A *procedure* is a subroutine that does not return a value.

Section 6.3

math Library Functions

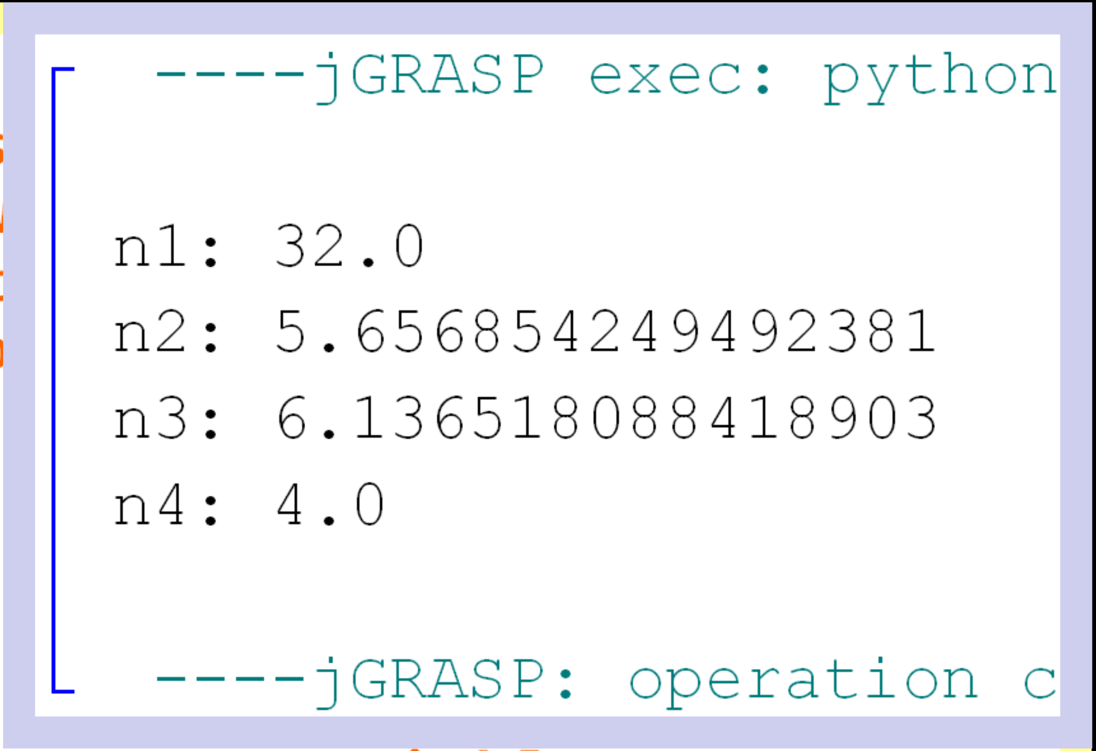
```python
# mathLibrary01.py
# This program demonstrates the <sqrt> function
# of the <math> library, which returns the
# principal square root of the argument.

# NOTE: Most of the functions in the <math> library
#       work with both integers and real numbers.


# Required to have access to most
# of the <math> library functions.
from math import *

print()

print("The square root of 625 is",sqrt(625))

print("The square root of 6.25 is",sqrt(6.25))
```

```
 ----jGRASP exec: python mathLibrary01.py

The square root of 625 is 25.0
The square root of 6.25 is 2.5

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

```python

# Required to have access to most
# of the <math> library functions.
from math import *

print()

print("The square root of 625 is",sqrt(625))

print("The square root of 6.25 is",sqrt(6.25))
```

```python
# mathLibrary02.py
# This program shows several different arguments
# that can be used with the <sqrt> function.
# Note how one function call can be the argument
# of another function call.


from math import *

n1 = sqrt(1024)         # constant argument
n2 = sqrt(n1)           # variable argument
n3 = sqrt(n1 + n2)      # expression argument
n4 = sqrt(sqrt(256))    # function argument

print()
print("n1:",n1)
print("n2:",n2)
print("n3:",n3)
print("n4:",n4)
```

```python
# mathLibrary02.py
# This program shows
# that can be used w
# Note how one funct
# of another functio


from math import *

n1 = sqrt(1024)
n2 = sqrt(n1)         # variable argument
n3 = sqrt(n1 + n2)    # expression argument
n4 = sqrt(sqrt(256))  # function argument

print()
print("n1:",n1)
print("n2:",n2)
print("n3:",n3)
print("n4:",n4)
```



```
 ----jGRASP exec: python

n1: 32.0
n2: 5.656854249492381
n3: 6.136518088418903
n4: 4.0

 ----jGRASP: operation c
```

```
# mathLibrary02.py
# This program shows
# that can be used w
# Note how one funct
# of another functio


from math import *

n1 = sqrt(1024)
n2 = sqrt(n1)           # variable argument
n3 = sqrt(n1 + n2)      # expression argument
n4 = sqrt(sqrt(256))    # function argument

print()
print("n1:",n1)
print("n2:",n2)
print("n3:",n3)
print("n4:",n4)
```

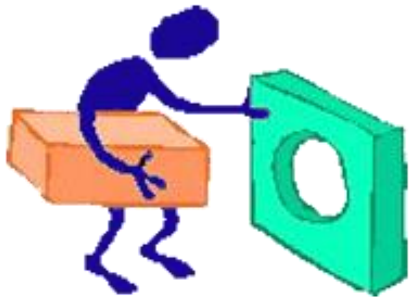
```
 ----jGRASP exec: python

n1: 32.0
n2: 5.656854249492381
n3: 6.136518088418903
n4: 4.0

 ----jGRASP: operation c
```

$$n3 = \sqrt{32 + 5.656\ldots}$$

$$n4 = \sqrt{\sqrt{256}} = \sqrt{16} = 4$$

```python
# mathLibrary03.py
# This program demonstrates what happens when a
# function is called with the wrong data type.


from math import *

n = sqrt("Fish")  # invalid argument

print()
print("The square root of 'Fish' is:",n)

```

```python
1 # mathLibrary03.py
2 # This program demonstrates what happens when a
3 # function is called with the wrong data type.
4
5
6 from math import *
7
8 n = sqrt("Fish")  # invalid argument
9
```



```
 ----jGRASP exec: python mathLibrary03.py
Traceback (most recent call last):
  File "mathLibrary03.py", line 9, in <module>
    n = sqrt("Fish")  # invalid argument
TypeError: must be real number, not str


 ----jGRASP wedge2: exit code for process is 1.
 ----jGRASP: operation complete.
```

```python
# mathLibrary04.py
# This program demonstrates what happens when
# you take the square root of a negative number
# in Python. It causes a Run-time Error, similar
# to what happens when you divide by zero.


from math import *

print()
print("Execution Begins")

n = sqrt(-1)  # invalid argument

print()
print("The square root of -1 is ",n)
```

```
 ----jGRASP exec: python mathLibrary04.py

Execution Begins
Traceback (most recent call last):
  File "mathLibrary04.py", line 13, in <module>
    n = sqrt(-1)  # invalid argument
ValueError: math domain error

 ----jGRASP wedge2: exit code for process is 1.
 ----jGRASP: operation complete.
```

```
11 print("Execution Begins")
12
13 n = sqrt(-1)  # invalid argument
14
15 print()
16 print("The square root of -1 is ",n)
```

Subroutine Arguments

The information, which is passed to a subroutine (function or procedure) is called an *argument*.

Arguments are placed between parentheses immediately following the subroutine identifier.

Arguments can be constants, variables, expressions or they can be other function calls.

The only requirement is that the correct data type value is passed to the subroutine.

In other words, **sqrt(x)** can compute the square root of **x**, if **x** stores any non-negative number (integer or real number), but not if **x** stores a negative number or string value like **"Fish"**.

```python
# mathLibrary05.py
# This program demonstrates the <abs>
# function, which returns the absolute
# value of the argument.  It also shows
# how the <abs> function can be used to
# prevent the issue of a negative argument
# with the <sqrt> function.


from math import *

print()
print("The absolute value of -25 is",abs(-25))
print("The absolute value of 100 is",abs(100))
print("The absolute value of 3.7 is",abs(3.7))
print("The absolute value of -.5 is",abs(-.5))
print("The absolute value of 0.0 is",abs(0.0))
print()
print("The square root of the absolute value")
print("of -256 is",sqrt(abs(-256)))
```

```python
# mathLib
# This pr
# functio
# value o
# how the
# prevent
# with th


from math

print()
print("The absolute value of -25 is",abs(-25))
print("The absolute value of 100 is",abs(100))
print("The absolute value of 3.7 is",abs(3.7))
print("The absolute value of -.5 is",abs(-.5))
print("The absolute value of 0.0 is",abs(0.0))
print()
print("The square root of the absolute value")
print("of -256 is",sqrt(abs(-256)))
```

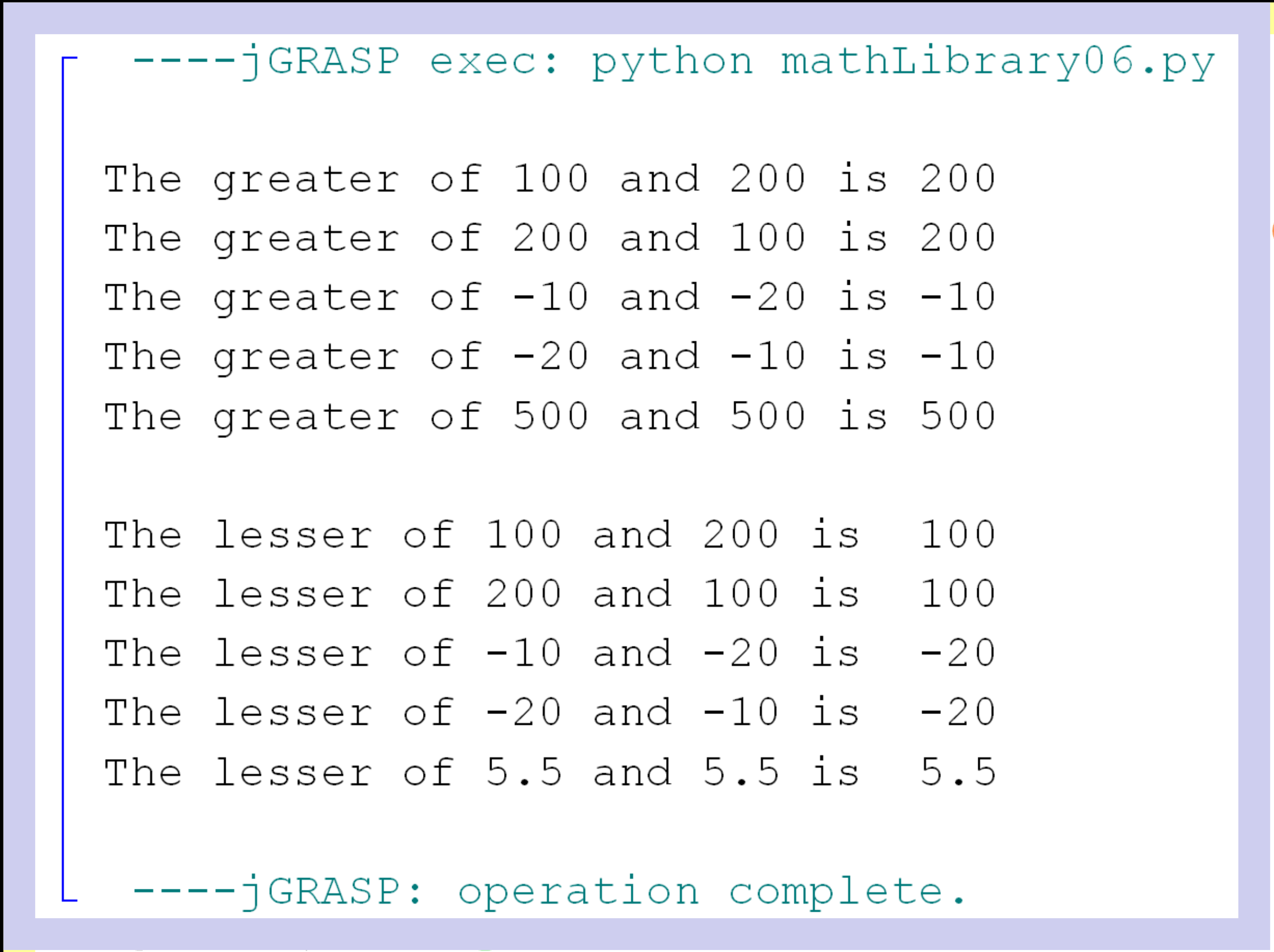
```
 ----jGRASP exec: python mathLibrary05.py

The absolute value of -25 is 25
The absolute value of 100 is 100
The absolute value of 3.7 is 3.7
The absolute value of -.5 is 0.5
The absolute value of 0.0 is 0.0

The square root of the absolute value
of -256 is 16.0

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

```python
# mathLib
# This pr
# functio
# value o
# how the
# prevent
# with th


from math

print()
print("The absolute value of -25 is",abs(-25))
print("The absolute value of 100 is",abs(100))
print("The absolute value of 3.7 is",abs(3.7))
print("The absolute value of -.5 is",abs(-.5))
print("The absolute value of 0.0 is",abs(0.0))
print()
print("The square root of the absolute value")
print("of -256 is",sqrt(abs(-256)))
```

```
 ----jGRASP exec: python mathLibrary05.py

 The absolute value of -25 is 25
 The absolute value of 100 is 100
 The absolute value of 3.7 is 3.7
 The absolute value of -.5 is 0.5
 The absolute value of 0.0 is 0.0

 The square root of the absolute value
 of -256 is 16.0

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

$$\sqrt{|-256|} = \sqrt{256} = 16$$

```python
# mathLibrary06.py
# This program demonstrates the <max> and <min>
# functions of the <math> library.
# <max> returns the greater of the two arguments.
# <min> returns the lesser of the two arguments.


from math import *

print()
print("The greater of 100 and 200 is",max(100,200))
print("The greater of 200 and 100 is",max(200,100))
print("The greater of -10 and -20 is",max(-10,-20))
print("The greater of -20 and -10 is",max(-20,-10))
print("The greater of 500 and 500 is",max(500,500))
print()
print("The lesser of 100 and 200 is ",min(100,200))
print("The lesser of 200 and 100 is ",min(200,100))
print("The lesser of -10 and -20 is ",min(-10,-20))
print("The lesser of -20 and -10 is ",min(-20,-10))
print("The lesser of 5.5 and 5.5 is ",min(5.5,5.5))
```



```
----jGRASP exec: python mathLibrary06.py

The greater of 100 and 200 is 200
The greater of 200 and 100 is 200
The greater of -10 and -20 is -10
The greater of -20 and -10 is -10
The greater of 500 and 500 is 500

The lesser of 100 and 200 is  100
The lesser of 200 and 100 is  100
The lesser of -10 and -20 is  -20
The lesser of -20 and -10 is  -20
The lesser of 5.5 and 5.5 is  5.5

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

```
and <min>

o arguments.
 arguments.

,max(100,200))
,max(200,100))
,max(-10,-20))
,max(-20,-10))
,max(500,500))
print()
print("The lesser of 100 and 200 is ",min(100,200))
print("The lesser of 200 and 100 is ",min(200,100))
print("The lesser of -10 and -20 is ",min(-10,-20))
print("The lesser of -20 and -10 is ",min(-20,-10))
print("The lesser of 5.5 and 5.5 is ",min(5.5,5.5))
```

```python
# mathLibrary07.py
# This program demonstrates the <pow> function
# of the <math> library which does the same
# thing as the exponent operator <**>.

# NOTE: Like <max> and <min>, <pow> uses 2
# arguments.  However, unlike <max> and <min>,
# with <pow> the order of the 2 arguments is
# VERY significant.
# The first argument is the "base".
# The second argument is the "exponent".
# <pow> returns the first argument to the
# "power" of the second argument.


from math import *

print()
print("3 to the 4th power using ** is",3 ** 4)
print("3 to the 4th power with pow is",pow(3,4))
print()
print("4 to the 3rd power using ** is",4 ** 3)
print("4 to the 3rd power with pow is",pow(4,3))
```

```
 ----jGRASP exec: python mathLibrary07.py

3 to the 4th power using ** is 81
3 to the 4th power with pow is 81.0

4 to the 3rd power using ** is 64
4 to the 3rd power with pow is 64.0

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

```python
from math import *

print()
print("3 to the 4th power using ** is",3 ** 4)
print("3 to the 4th power with pow is",pow(3,4))
print()
print("4 to the 3rd power using ** is",4 ** 3)
print("4 to the 3rd power with pow is",pow(4,3))
```

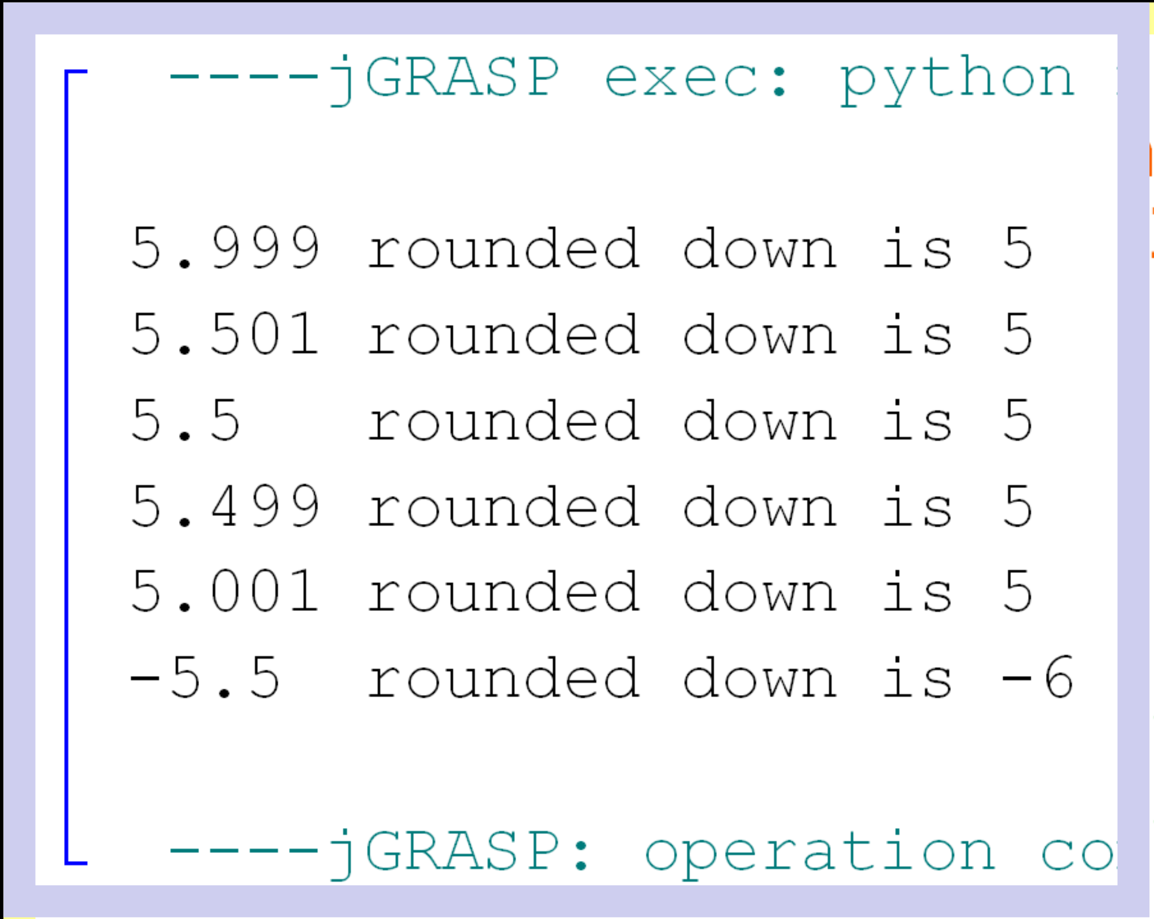
```
 ----jGRASP exec: python mathLibrary07.py


3 to the 4th power using ** is 81
3 to the 4th power with pow is 81.0


4 to the 3rd power using ** is 64
4 to the 3rd power with pow is 64.0


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

```python
from math import *

print()
print("3 to the 4th power using ** is",3 ** 4)
print("3 to the 4th power with pow is",pow(3,4))
print()
print("4 to the 3rd power using ** is",4 ** 3)
print("4 to the 3rd power with pow is",pow(4,3))
```

$3^4 = 81$ $4^3 = 64$

```python
# mathLibrary08.py
# This program demonstrates the <floor> function of
# the <math> library which always "rounds down".


from math import *

print()
print("5.999 rounded down is",floor(5.999))
print("5.501 rounded down is",floor(5.501))
print("5.5   rounded down is",floor(5.5))
print("5.499 rounded down is",floor(5.499))
print("5.001 rounded down is",floor(5.001))
print("-5.5  rounded down is",floor(-5.5))

```

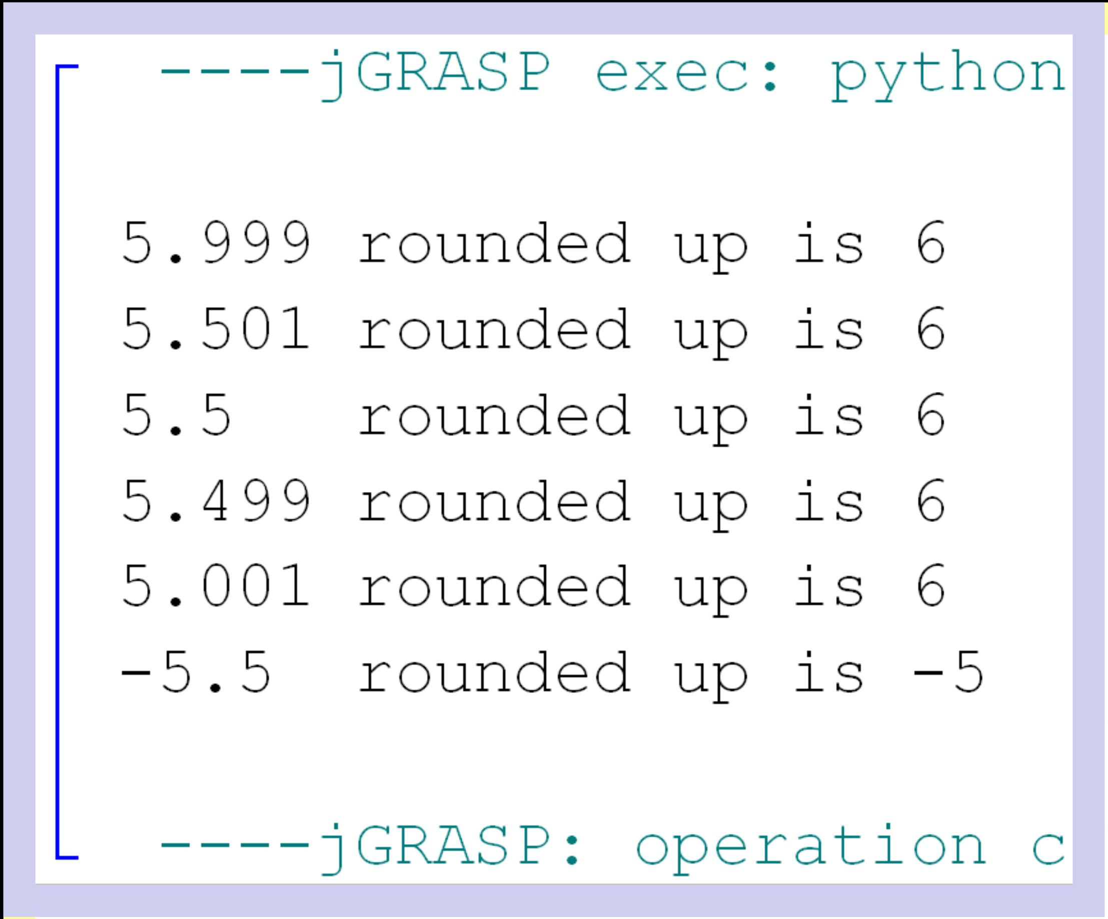
e <floor> function of
lways "rounds down".



```
 ----jGRASP exec: python

5.999 rounded down is 5
5.501 rounded down is 5
5.5   rounded down is 5
5.499 rounded down is 5
5.001 rounded down is 5
-5.5  rounded down is -6

 ----jGRASP: operation co
```

```
",floor(5.999))
",floor(5.501))
print("5.5   rounded down is",floor(5.5))
print("5.499 rounded down is",floor(5.499))
print("5.001 rounded down is",floor(5.001))
print("-5.5  rounded down is",floor(-5.5))

```

```python
# mathLibrary09.py
# This program demonstrates the <ceil> function of
# the <math> library which always "rounds up".


from math import *

print()
print("5.999 rounded up is",ceil(5.999))
print("5.501 rounded up is",ceil(5.501))
print("5.5   rounded up is",ceil(5.5))
print("5.499 rounded up is",ceil(5.499))
print("5.001 rounded up is",ceil(5.001))
print("-5.5  rounded up is",ceil(-5.5))

```

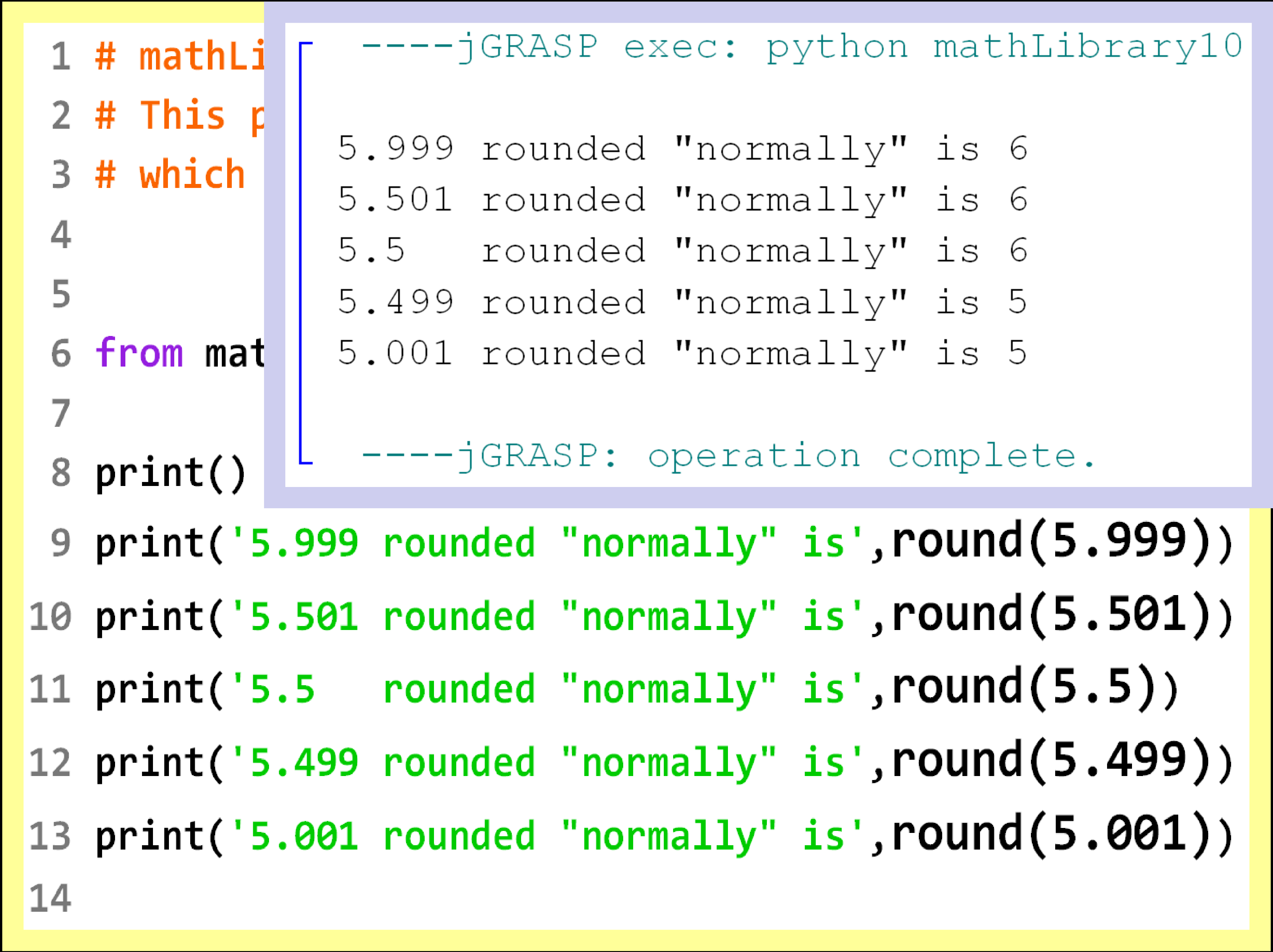


```
 ----jGRASP exec: python

 5.999 rounded up is 6
 5.501 rounded up is 6
 5.5   rounded up is 6
 5.499 rounded up is 6
 5.001 rounded up is 6
 -5.5  rounded up is -5

 ----jGRASP: operation c
```

the <ceil> function of always "rounds up".

```
                           ,ceil(5.999))
                           ,ceil(5.501))
print("5.5   rounded up is",ceil(5.5))
print("5.499 rounded up is",ceil(5.499))
print("5.001 rounded up is",ceil(5.001))
print("-5.5  rounded up is",ceil(-5.5))

```

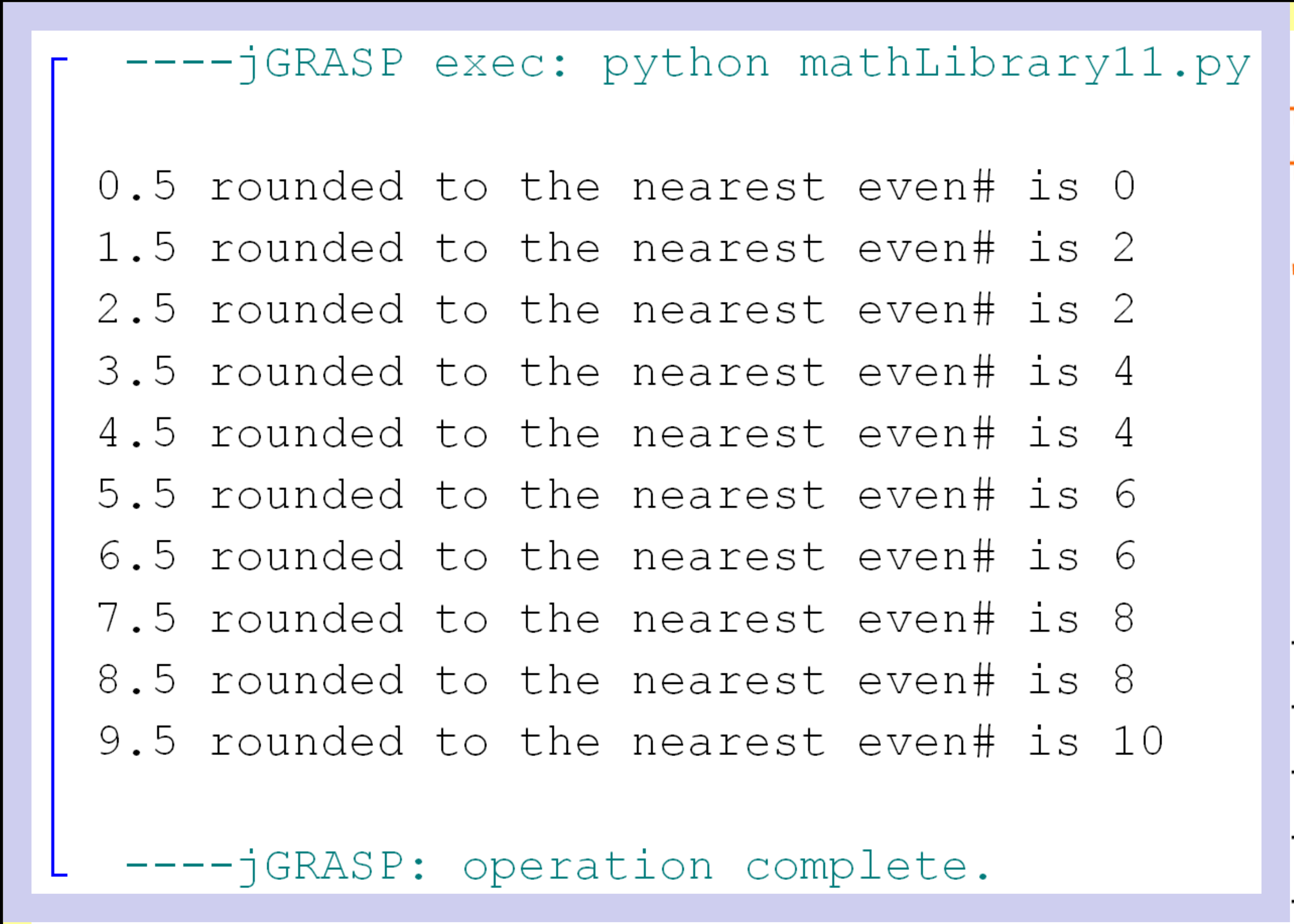
```python
# mathLibrary10.py
# This program demonstrates the <round> function
# which seems to round "normally".


from math import *

print()
print('5.999 rounded "normally" is',round(5.999))
print('5.501 rounded "normally" is',round(5.501))
print('5.5   rounded "normally" is',round(5.5))
print('5.499 rounded "normally" is',round(5.499))
print('5.001 rounded "normally" is',round(5.001))

```



```
# mathLi
# This p
# which


from mat

print()
print('5.999 rounded "normally" is',round(5.999))
print('5.501 rounded "normally" is',round(5.501))
print('5.5   rounded "normally" is',round(5.5))
print('5.499 rounded "normally" is',round(5.499))
print('5.001 rounded "normally" is',round(5.001))

```

```
 ----jGRASP exec: python mathLibrary10

5.999 rounded "normally" is 6
5.501 rounded "normally" is 6
5.5   rounded "normally" is 6
5.499 rounded "normally" is 5
5.001 rounded "normally" is 5

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

```python
# mathLibrary11.py
# When the fractional value is exactly .5, the
# <round> function does not behave as expected.
# This is because Python's <round> function uses
# "banker's rounding" a.k.a. "round-to-even".


from math import *

print()
print("0.5 rounded to the nearest even# is",round(0.5))
print("1.5 rounded to the nearest even# is",round(1.5))
print("2.5 rounded to the nearest even# is",round(2.5))
print("3.5 rounded to the nearest even# is",round(3.5))
print("4.5 rounded to the nearest even# is",round(4.5))
print("5.5 rounded to the nearest even# is",round(5.5))
print("6.5 rounded to the nearest even# is",round(6.5))
print("7.5 rounded to the nearest even# is",round(7.5))
print("8.5 rounded to the nearest even# is",round(8.5))
print("9.5 rounded to the nearest even# is",round(9.5))
```

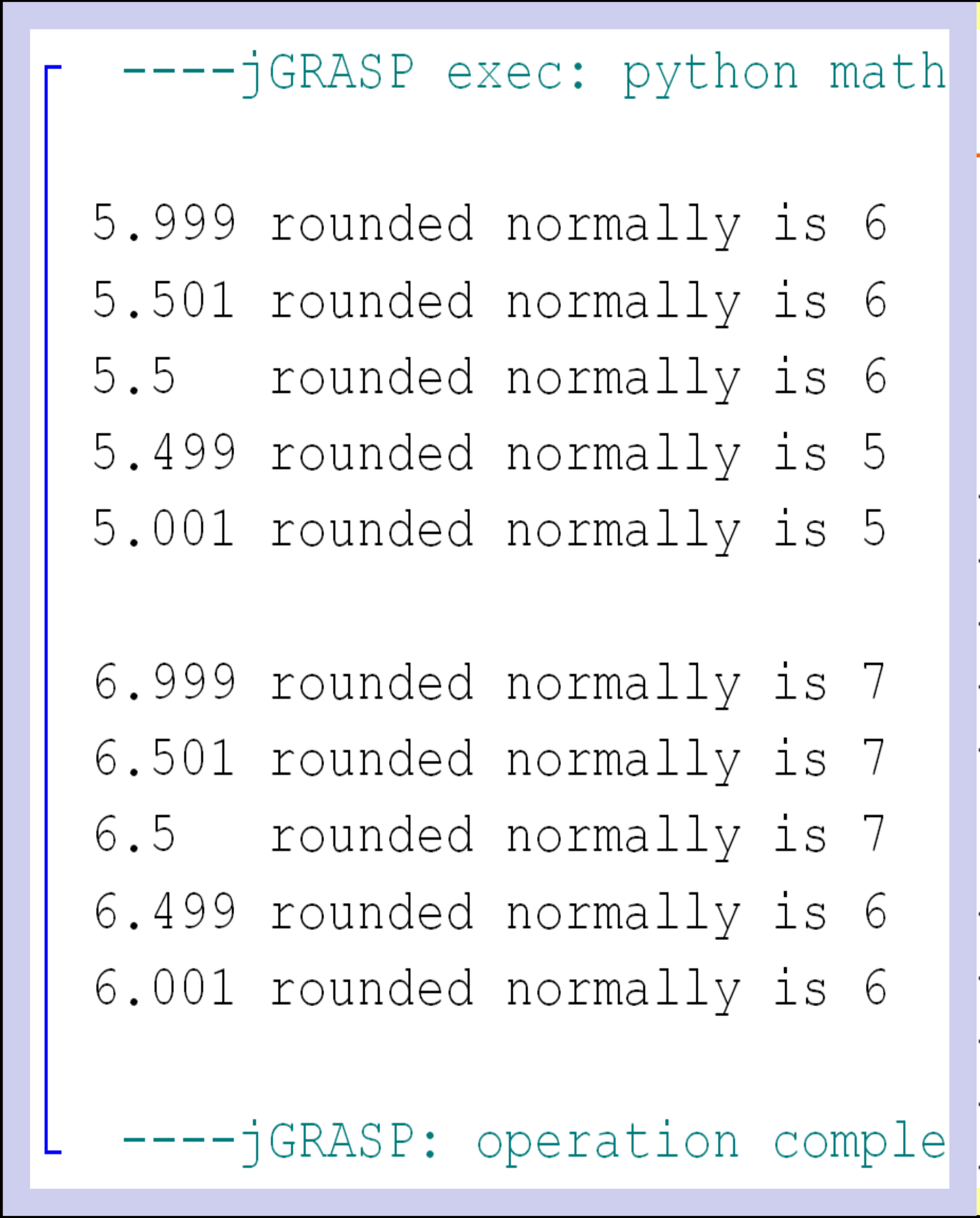


```
 ----jGRASP exec: python mathLibrary11.py

0.5 rounded to the nearest even# is 0
1.5 rounded to the nearest even# is 2
2.5 rounded to the nearest even# is 2
3.5 rounded to the nearest even# is 4
4.5 rounded to the nearest even# is 4
5.5 rounded to the nearest even# is 6
6.5 rounded to the nearest even# is 6
7.5 rounded to the nearest even# is 8
8.5 rounded to the nearest even# is 8
9.5 rounded to the nearest even# is 10

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

the
ted.
 uses
".

```
,round(0.5))
,round(1.5))
,round(2.5))
,round(3.5))
,round(4.5))
print("5.5 rounded to the nearest even# is",round(5.5))
print("6.5 rounded to the nearest even# is",round(6.5))
print("7.5 rounded to the nearest even# is",round(7.5))
print("8.5 rounded to the nearest even# is",round(8.5))
print("9.5 rounded to the nearest even# is",round(9.5))
```

```python
# mathLibrary12.py
# The secret to "normal rounding" is to use the
# <floor> function and add .5 to the argument.


from math import *

print()
print('5.999 rounded normally is',floor(5.999 + .5))
print('5.501 rounded normally is',floor(5.501 + .5))
print('5.5   rounded normally is',floor(5.5   + .5))
print('5.499 rounded normally is',floor(5.499 + .5))
print('5.001 rounded normally is',floor(5.001 + .5))

print()
print('6.999 rounded normally is',floor(6.999 + .5))
print('6.501 rounded normally is',floor(6.501 + .5))
print('6.5   rounded normally is',floor(6.5   + .5))
print('6.499 rounded normally is',floor(6.499 + .5))
print('6.001 rounded normally is',floor(6.001 + .5))
```



```
 ----jGRASP exec: python math

5.999 rounded normally is 6
5.501 rounded normally is 6
5.5   rounded normally is 6
5.499 rounded normally is 5
5.001 rounded normally is 5

6.999 rounded normally is 7
6.501 rounded normally is 7
6.5   rounded normally is 7
6.499 rounded normally is 6
6.001 rounded normally is 6

 ----jGRASP: operation comple
```

 is to use the
the argument.

```
,floor(5.999 + .5))
,floor(5.501 + .5))
,floor(5.5   + .5))
,floor(5.499 + .5))
,floor(5.001 + .5))

,floor(6.999 + .5))
,floor(6.501 + .5))
,floor(6.5   + .5))
,floor(6.499 + .5))
,floor(6.001 + .5))
```

```python
# mathLibrary13.py
# The program demonstrates the <trunc> function
# which "chops-off" or "truncates" the fractional
# part of a real number.  While this may seem
# identical to the <floor> function, it does
# behave differently with negative numbers.


from math import *

print()
print("5.678 rounded down is",floor(5.678))
print('5.678 "truncated" is ',trunc(5.678))

print()
print("-5.678 rounded down is",floor(-5.678))
print('-5.678 "truncated" is ',trunc(-5.678))

```

```
# mathLi
# The pr
# which
# part o
# identi
# behave


from mat

print()
print("5.678 rounded down is",floor(5.678))
print('5.678 "truncated" is ',trunc(5.678))

print()
print("-5.678 rounded down is",floor(-5.678))
print('-5.678 "truncated" is ',trunc(-5.678))

```

```
 ----jGRASP exec: python mathLibrary13

5.678 rounded down is 5
5.678 "truncated" is  5

-5.678 rounded down is -6
-5.678 "truncated" is  -5

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

```python
# mathLibrary14.py
# The program demonstrates the <factorial>
# function of the <math> library which returns
# the mathematical factorial of its argument.

# Example: factorial(n) = n * (n-1) * ... * 2 * 1

# NOTE: This program also demonstrates that in
#       Python integer values can be VERY large.


from math import *

print()
print("5! is",factorial(5))
print()
print("10! is",factorial(10))
print()
print("40! is",factorial(40))
```

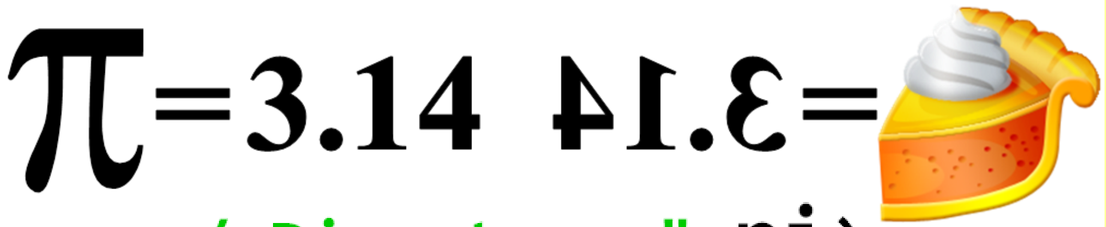
```
 ----jGRASP exec: python mathLibrary14.py

 5! is 120

 10! is 3628800

 40! is 815915283247897734345611269596115894272000000000

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

```python
print()
print("5! is",factorial(5))
print()
print("10! is",factorial(10))
print()
print("40! is",factorial(40))
```

```python
# mathLibrary15.py
# The <math> library also stores 2 important
# values.  These are the value of <pi> and
# the value of <e>.


from math import *

print()
print("Circumference / Diameter =",pi)
print()
print("Base of the natural log is",e);

```

```
 ----jGRASP exec: python mathLibrary15.py

Circumference / Diameter = 3.141592653589793

Base of the natural log is 2.718281828459045

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

```python

print()
print("Circumference / Diameter =",pi)
print()
print("Base of the natural log is",e);

```



```python
# mathLibrary16.py
# The <math> library contains many more functions
# that we will not be using in this class.  Some
# trigonometric functions are demonstrated below:


from math import *

print()
print("The sine of half pi    ",sin(pi/2))
print("The cosine of pi is    ",cos(pi))
print("The tangent of pi/4 is ",tan(pi/4))
print("The natural log of e is",log(e))
print("Log base 10 of 1000 is ",log10(1000))

```

```
 ----jGRASP exec: python mathLibrary16.py

The sine of half pi     1.0
The cosine of pi is     -1.0
The tangent of pi/4 is  0.9999999999999999
The natural log of e is 1.0
Log base 10 of 1000 is  3.0

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

```python
print("The sine of half pi    ",sin(pi/2))
print("The cosine of pi is    ",cos(pi))
print("The tangent of pi/4 is ",tan(pi/4))
print("The natural log of e is",log(e))
print("Log base 10 of 1000 is ",log10(1000))
```

Python math Library Disclaimer

Not every function that is *mathematical* in nature is actually part of the **math** library. There are a couple functions, like **round** and **abs**, that can be used without importing **math** or any other library.

For the sake of simplicity, in this first year class, we will make no attempt to differentiate the *mathematical* functions that require importing the **math** library from the ones that do not. Instead, we will simply import the **math** library any time we use any function that is *mathematical* in nature… which does not hurt anything.