

Chapter IV

Simple Data Types

Chapter IV Topics

- 4.1 Introduction
- 4.2 Arithmetic Operations
- 4.3 Numeric Data Types
- 4.4 Non-Numeric Data Types
- 4.5 Shortcuts
- 4.6 Swapping Variable Values
- 4.7 Documenting Your Programs
- 4.8 More Syntax Errors
- 4.9 Other Types of Errors
- 4.10 Output Programs, Slides, Exercises & Quizzes

4.1 Introduction

In the early days of your math courses only constants were used. You know what I mean. Numbers were **5**, **13** and **127**. You added, subtracted, multiplied and divided with numbers. Later, you had more fun with fractions and decimal numbers. At some point - and the exact year does not matter - variables were introduced. In science and mathematics it is useful to express formulas and certain relationships with variables that explain some general principle. If you drive at an average rate of **60 mph** and you continue for **5 hours** at that rate, you will cover **300 miles**. On the other hand, if you drive at a rate of **45mph** for **4 hours**, you will travel **180 miles**. These two problems are examples that only use constants. The method used for computing this type of problem can be expressed in a more general formula that states:

$$\text{Distance} = \text{Rate} \times \text{Time}$$

The formula is normally used in its abbreviated form, which is **$d = r \times t$** . In this formula **d**, **r** and **t** are variables. The meaning is literal. A variable stores a value that is "able" to change or "vary". A constant like **5** will always be **5**, but **d** is a variable, which changes based on the values stored in **r** and **t**, which are also variables. Variables make mathematics, science and computer science possible. Without variables you are very limited in the type of programs that you can write. In this chapter you will learn how to use variables in your programs.

4.2 Arithmetic Operations

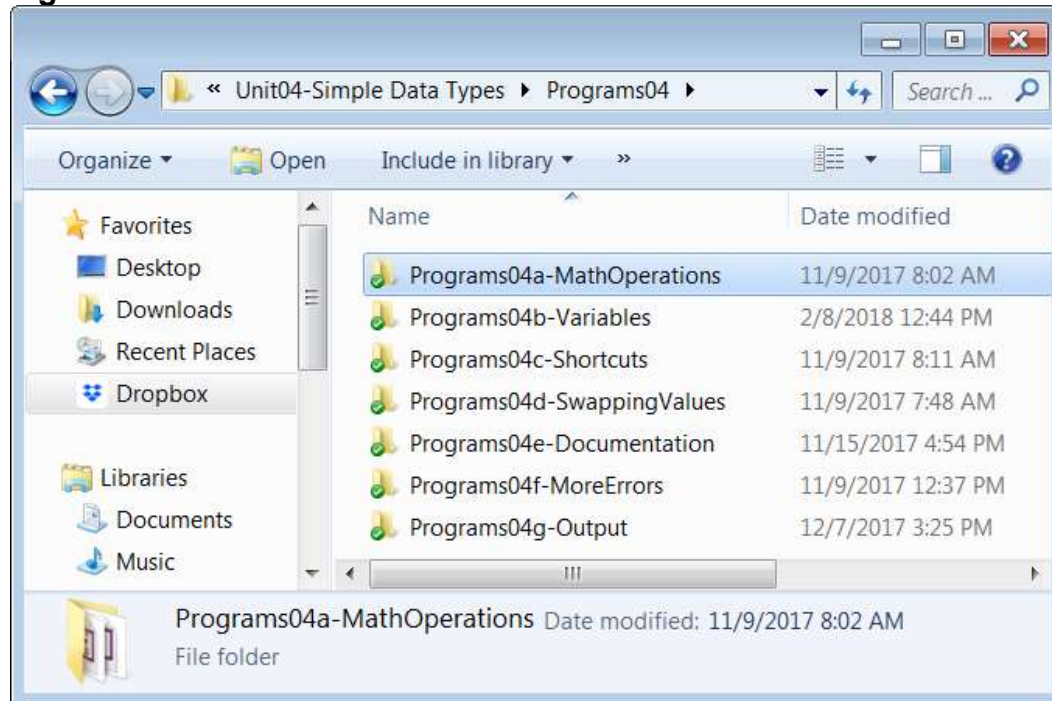
There are operations, operators and operands. In Figure 4.1, **+** is the *operator*. Both **15** and **23** are the *operands* and **15 + 23** is an *expression* that performs the *addition operation*. An operation that involves two operands and one operator, like the one in figure 4.34 is called a *binary operation*. In this section you will learn that Python has many operators. You will also note that the **print** command not only displays text; it can also display the value of various operations.

Figure 4.1


$$15 + 23 = 38$$

Remember that starting with Chapter III, each Unit folder has a **ProgramsXX** subfolder, which is usually divided into sub-subfolders as shown in Figure 4.2.

Figure 4.2



Back in Chapter III you were introduced to the **print** command. You saw that it can display *string literals* (the text between quotes) and it can also skip lines. There is not all that **print** can do. Program **MathOperations01.py**, shown in Figure 4.3, demonstrates some other abilities of **print**. The first **print** command simply skips a line. (I will begin most of my programs like this for the sake of spacing.) The second **print** prints the number **7**. What is different is that there are no *quotes* used. While string literals require quotes, numerical values do not.

In the next 4 program statements, **print** is behaving like a calculator and is displaying the numerical value of each of these mathematical expressions. You probably recognized right away that the **print** commands on lines 11 and 13 are performing *addition* and *subtractions* respectively. Had I not supplied a couple comments near the end of the program, you may not have realized that the **print** on line 15 is performing *multiplication* and the **print** on line 17 is performing *division*. In computer science, we do not use the symbols \times and \div to multiply and divide. The asterisk (`*`) is the standard symbol for multiplication in most programming languages. The same is true about the slash (`/`) and division. In recent years, I have even seen some calculators use the asterisks and slash symbols instead of the older symbols.

Figure 4.3

```
1 # MathOperations01.py
2 # This program demonstrates that the <print>
3 # command can evaluate and display the result
4 # of mathematical expressions.
5
6
7 print()
8
9 print(7)
10
11 print(7 + 2)
12
13 print(7 - 2)
14
15 print(7 * 2)    # Multiplication
16
17 print(7 / 2)    # Division
18
```

```
----jGRASP exec: python MathOperations01.py

9
5
14
3.5

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

Question, what if you literally wanted to display “7 + 2” on the screen instead of the resulting value of 9? That is possible if we put the expression inside quotes. This is demonstrated by program **MathOperations02.py**, shown in Figure 4.4. Now that we have text inside quotes, this has become a *string literal* and is no longer a *mathematical expression*.

Figure 4.4

```
1 # MathOperations02.py
2 # This program demonstrates that whenever
3 # you <print> something in quotes, you get
4 # exactly what is inside the quotes.
5
6
7 print()
8
9 print("7 + 2")
10
```

```
----jGRASP exec: python MathOperations02.py

7 + 2

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

Python actually has 3 different types of *division*. All 3 are demonstrated by program **MathOperations03.py**, shown in Figure 4.5. The first type of division is probably the one with which you are the most familiar. This is *Real Number Division*. This performed with the slash (/) operator and is demonstrated on line 8 of the program. The second type of division is *Integer Division*. This performed with the double-slash (//) operator and is demonstrated on line 10 of the program. The third type of division is *Remainder Division*. This performed with the percent (%) operator and is demonstrated on line 12 of the program.

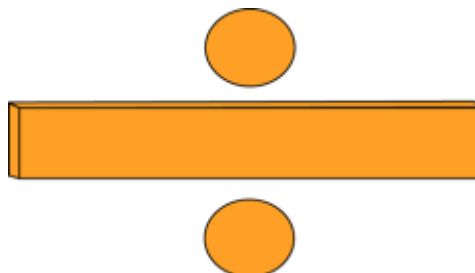


Figure 4.5

```
1 # MathOperations03.py
2 # This program demonstrates the 3
3 # different division operators.
4
5
6 print()
7
8 print(7 / 2)    # Real Number Division
9
10 print(7 // 2)  # Integer Division
11
12 print(7 % 2)   # Remainder Division
13
```

```
----jGRASP exec: python MathOperations03.py

3.5
3
1

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

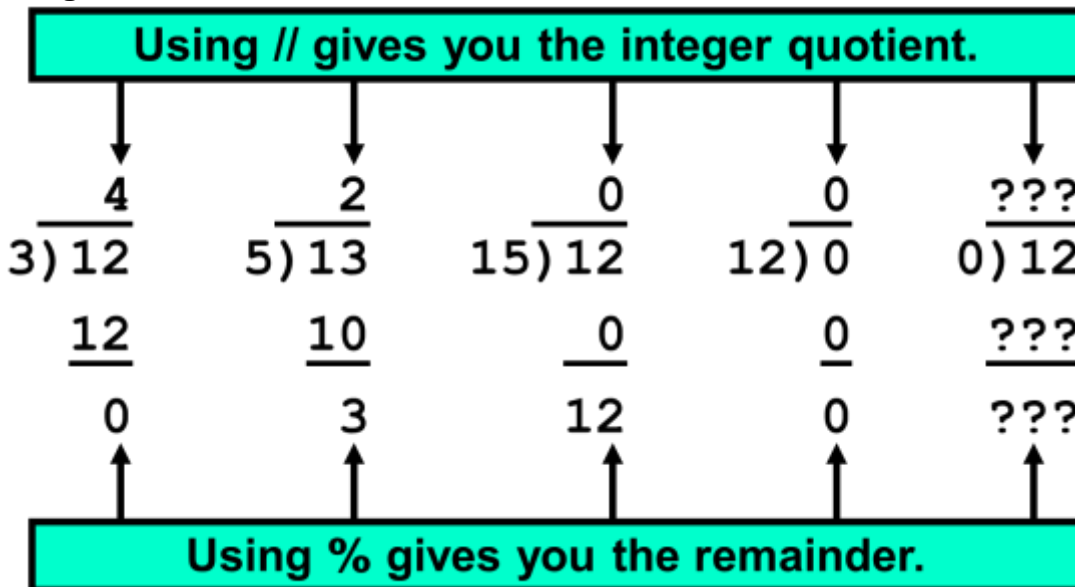
Now, it is possible that you might not see the purpose of the other 2 types of division. *Real Number Division* is the way a calculator divides. Why would you divide any other way? Consider this word problem:

A group of 75 students is going to be transported on 2 buses. How many students should be placed on each bus?

Frequently, I have students, yes high-school students, tell me the answer is **37½**! If you are wondering why that answer is wrong, ask yourself if you want to be that last student who is “divided” so that each of your “halves” can be placed on separate busses. This is a real world example of where *Integer Division* and *Remainder Division* are used. The first 37 students are placed on first bus. The other 37 students, along with the one remaining student are placed on the second bus.

While *Integer Division* and *Remainder Division* may seem strange to you now, they were probably the first types of division that you learned. Long ago, back when you were in elementary school, you learned how to do *Long Division*. Back then, you did not have real number answers with several digits past the decimal point. Your quotients were simple integers – and if the number did not divide evenly, there was remainder. If you look at Figure 4.6, you see several examples of this. The number on the top is the *integer quotient*. This is what you get with the double-slash (`//`) operator. The number on the bottom is what is left over, or as we call it, the *remainder*. This is what you get with the percent (`%`) operator.

Figure 4.6



Suppose you are working with more complex mathematical expressions that involve *exponents*. Exponents are handled in Python with the double asterisk (`**`) operator. The operation 25^2 is handled in Python by using `25 ** 2`. Program **MathOperations04.py**, shown in Figure 4.7, demonstrates this example along with several others. In the process a number of exponent rules are reviewed. The **print** statements on lines 7 and 9 display the values of “25 squared” and “25 cubed”. The next one demonstrates that anything to the power of **1** is itself. The next one demonstrates that anything to the power of **0** is **1**. The **print** statement on line 15 demonstrates that taking a number to the power of **-1** gives you its *inverse* or *reciprocal*. In other words, $25^{-1} = 1/25 = 0.04$. The last one demonstrates that taking a number to the power of $\frac{1}{2}$ or **0.5** gives you its *square root*. In other words, $25^{0.5} = \sqrt{25} = 5$.

Figure 4.7

```
1 # MathOperations04.py
2 # This program demonstrates the Exponent Operator **
3
4
5 print()
6
7 print(25 ** 2)    # 25 * 25
8
9 print(25 ** 3)    # 25 * 25 * 25
10
11 print(25 ** 1)    # Anything to the power of 1 is itself
12
13 print(25 ** 0)    # Anything to the power of 0 is 1
14
15 print(25 ** -1)   # The reciprocal of 25 or 1/25
16
17 print(25 ** 0.5)  # The square root of 25
18
```

```
----jGRASP exec: python MathOperations04.py

625
15625
25
1
0.04
5.0

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


Python Has Seven Arithmetic Operators

Addition	+
Subtraction	-
Multiplication	*
Real Number Division	/
Integer Division	//
Remainder Division	%
Exponents	**

Order Of Operations (PEMDAS)

In elementary school you learned that arithmetic follows an order of operations. Perhaps you also remember a common word **PEMDAS** that is an acronym for the mathematical precedence of operations. This precedence is exactly the same for computer programs written in Python, and just about any other language for that matter. A set of program examples will follow, that will demonstrate how arithmetic expressions are evaluated.

PEMDAS Order of Arithmetic Operations

Parentheses

Exponents

Multiplication
Division

Addition
Subtraction

Program **MathOperations05.py**, shown in Figure 4.8, performs 3 separate calculations. Each of these involves more than one mathematical operation. In each case, the leftmost mathematical operation is not the first operation performed. This is all due to the fact that Python is following “Order of Operations” or “PEMDAS”.

Figure 4.8

```
1 # MathOperations05.py
2 # Python follows "Order of Operations"
3 # a.k.a. "PEMDAS"
4
5
6 print()
7
8 print(2 + 3 * 4)
9
10 print(14 - 6 / 2)
11
12 print(100 / 5 ** 2 * 3)
13
```

```
----jGRASP exec: python MathOperations05.py

14
11.0
12.0

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

Look again at this program, specifically on line 8. Suppose you actually want to first add 2 and 3, and then multiply the resulting value, 5, by 4. This is possible if we remember the “P” in “PEMDAS”. *Parentheses* come before everything else.

Program **MathOperations06.py**, shown in Figure 4.9, performs calculations that look similar to the previous program, but have very different results because *parentheses* have been inserted in various locations to force certain operations to calculate before others.

Figure 4.9

```
1 # MathOperations06.py
2 # Parentheses make a difference.
3
4
5 print()
6
7 print((2 + 3) * 4)
8
9 print((14 - 6) / 2)
10
11 print((100 / 5) ** (2 * 3))
```

```
----jGRASP exec: python MathOperations06.py

20
4.0
64000000.0

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

Before we move on to the next program, consider this math problem. What is **2** to the power of **3** to the power of **4**? That is not as straight forward as you might think. It all comes down to how you write it. Consider these 3 examples:

Figure 4.10

$(2^3)^4$	2^{3*4}	2^3^4
-----------	-----------	---------

If we look at figure 4.10, the left example and the middle example are mathematically equivalent. This is based on one of those laws of exponents that you learned in your math class. Program **MathOperations07.py**, shown in Figure 4.11, demonstrates this on line 8. We see that both expressions yield a result of **4096**. Now, let us remove the parentheses. This would give us the expression on the right side of figure 4.10, which is in no way equal to the other 2. This is demonstrated on line 12. Without the parentheses, it first calculates 3^4 , which is 81. Then it calculates 2^{81} , which is a number that is much, much larger than **4096**.

Figure 4.11

```
1 # MathOperations07.py
2 # When exponents have exponents, parentheses
3 # are very important.
4
5
6 print()
7
8 print((2 ** 3) ** 4, "equals", 2 ** (3 * 4))
9
10 print()
11
12 print((2 ** 3) ** 4, "does not equal", 2 ** 3 ** 4)
13
```

```
----jGRASP exec: python MathOperations06.py

4096 equals 4096

4096 does not equal 2417851639229258349412352

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

4.3 Numeric Data Types

Computer programs simulate procedures that we perform in life. For centuries mathematicians and scientist have solved problems with equations and formulas. Computers are designed to solve those very same problems, but much quicker and more accurately. Equations and formulas use variables, such as the ones shown in Figure 4.12. The first formula computes the distance traveled based on rate and time. The second formula converts Fahrenheit degrees into Celsius degrees.

Figure 4.12

$$d = r * t$$
$$c = (f - 32) * 5/9$$

Integers

Program **Variables01.py**, shown in Figure 4.13, defines 2 integer variables, **x** and **y**, and displays their values. The program statement **x = 50** stores the value of **50** in **x**. Some people like to say that the value of **50** is *bound* to **x**. Here we see another ability of the **print** command. It can display the value stored in a variable. Notice that while lines 10 and 11 say **print(x)** and **print(y)** the output does not show **x** and **y**, it shows **50** and **75**.

Figure 4.13

```
1 # Variables01.py
2 # This program demonstrates how to define
3 # integer variables & display their values.
4
5
6 x = 50
7 y = 75
8
9 print()
10 print(x)
11 print(y)
```

```
----jGRASP exec: python Variables01.py

50
75

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

You might be wondering, “OK, I can display the value of a variable, like **x**, in this way, but what if I want to actually display the letter ‘**x**’ on the screen?” Well, you actually learned how to do that in the previous chapter. Just put the “**x**” in *quotes* and now you are displaying a *string literal*. This is demonstrated in program **Variables02.py**, shown in Figure 4.14.

Figure 4.14

```
1 # Variables02.py
2 # Remember, quotes make output literal.
3
4
5 x = 50
6 y = 75
7
8 print()
9 print("x")
10 print("y")
11
```

```
----jGRASP exec: python Variables02.py

x
y

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

print(x) vs. print("x")	
print(x)	Display the value of the variable x .
print("x")	Display the letter "x" .

In Python, a variable becomes *defined* once a value is assigned. In the previous example, the statement `x = 50` both *defines* variable `x` as an integer variable and *initializes* it with the value of `50`. You cannot use a variable until it has been both defined and initialized. Program **Variables03.py**, in Figure 4.15, demonstrates what happens when you break that rule. Python gives you a syntax error because the variable with “**name 'x' is not defined**”.

Figure 4.15

```

1 # Variables03.py
2 # This program demonstrates that you
3 # cannot use an undefined variable.
4
5
6 print(x)
7

```

```

----jGRASP exec: python Variables03.py
Traceback (most recent call last):
  File "Variables03.py", line 6, in <module>
    print(x)
NameError: name 'x' is not defined

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

```

Another feature of the **print** command is that it displays more than one thing on a line at a time. This is demonstrated in program **Variables04.py**, shown in Figure 4.16. If you look at line 10, you should see that a comma (,) is used to separate the different items to be displayed.

Figure 4.16

```
1 # Variables04.py
2 # This program demonstrates how to print the
3 # values of multiple variables on one line.
4 # By default, these values are separated by
5 # a single space, but this can be changed
6 # by using <sep> at the end of <print>.
7
8
9 x = 50
10 y = 75
11 z = 100
12
13 print()
14 print(x,y)
15 print(x,y,z)
16 print()
17 print(x,y,sep="")
18 print(x,y,z,sep="<:>")
19 print(x,y,z,sep="\t\t")
20
```

```
----jGRASP exec: python Variables04.py

50 75
50 75 100

5075
50<:>75<:>100
50      75      100

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


Since we have the ability to display more than one piece of information on a line, we can make our program output more *user-friendly*. What do I mean by “user-friendly?” Consider this. If **x** is storing **50**, the command **print(x)** will display **50**. But when the “user” of the program looks at the output, what does the **50** mean? There is no way to tell if this 50 printers, 50 gumdrops, or a bill for \$50. Program **Variables05.py**, in Figure 4.17, shows that a single **print** statement can display both a string literal and the value of a variable on the same line. In this case, the command **print("The value of x is", x)** displays **The value of x is 50**, which makes much more sense than the **50** by itself.

Figure 4.17

```
1 # Variables05.py
2 # This program demonstrates how to
3 # make the output more "User-Friendly"
4
5
6 x = 50
7 y = 75
8
9 print()
10 print("The value of x is", x)
11 print("The value of y is", y)
```

```
----jGRASP exec: python Variables05.py

The value of x is 50
The value of y is 75

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

Math operations can be performed with variables as well. There is an important difference between variables in math class and variables in computer science. In math, a variable is always a single letter, like **x** or **y**. In computer science, a variable can also be an entire word like **sum**. This is all demonstrated in program **Variables06.py**, shown in Figure 4.18.

Figure 4.18

```
1 # Variables06.py
2 # This program demonstrates that you can
3 # do math operations with variables.
4 # It also shows that variable names
5 # do not need to be single letters.
6 # They are often words or compound words.
7
8
9 x = 50
10 y = 75
11 sum = x + y
12
13 print()
14 print("The sum of",x,"and",y,"is",sum)
```

```
----jGRASP exec: python Variables06.py

The sum of 50 and 75 is 125

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

Program **Variables07.py**, in Figure 4.19, can be a little confusing. The problem is the `x = x + 20` command on line 10. People who remember Algebra ask “How can `x` be equal to `x + 20`?” Well, it isn’t because `x = x + 20` is NOT an equation. Neither is `x = 10` for that matter. These are both *assignment statements* and should be read as “`x` becomes 10” and “`x` becomes the sum of `x` and 20”.

Figure 4.19

```
1 # Variables07.py
2 # <x = 10> is NOT an equation.
3 # It assigns the value of <10> to <x>.
4 # <x = x + 20> is also NOT an equation.
5 # It adds <20> to the value already
6 # stored in <x>.
7
```

```

8
9 x = 10
10 x = x + 20
11
12 print()
13 print("The value of x is", x)

```

```

[ ----jGRASP exec: python Variables07.py

The value of x is 30

----jGRASP: operation complete.

```

Real Numbers

All of the variables that we have looked at so far have stored *integers*. We are not going to look at some other data types, starting with *real numbers*. Program **Variables08.py**, in Figure 4.20, defines and initializes 3 real number variables. We have **q** which is storing **33.3333**, **r** which is storing **12.5** and **pi** which is storing the value of π . If you look at lines 11, 12 and 13, you see that the way we display the value of real number variables is no different from the way we display the value of integer variables.

Figure 4.20

```

1 # Variables08.py
2 # The program shows that variables can
3 # also store real number values.
4
5
6 q = 77.7777
7 r = 12.5
8 pi = 3.141592653589793
9
10 print()

```

```
11 print(q)
12 print(r)
13 print(pi)
```

```
----jGRASP exec: python Variables08.py

77.7777
12.5
3.141592653589793

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

When it comes to mathematical calculations, you can perform the same operations with both integers and real numbers. This is demonstrated in program **Variables09.py**, shown in Figure 4.21. One important point needs to be made here. While you can technically use the double slash (`//`) and percent (`%`) operators with real numbers, it should be understood that *Integer Division* and *Remainder Division* are meant for integers. They are not meant to be used with real numbers.

Figure 4.21

```
1 # Variables09.py
2 # The program demonstrates that the same
3 # mathematical operations that work with
4 # integers, also work with real numbers.
5 # NOTE: While "integer division" and
6 # "remainder division" operators technically
7 # work with real numbers, they are not
8 # really meant for them and should only
9 # be used with integers.
10
11
12 q = 33.3333
13 r = 12.5
14
```

```

15 a = q + r
16 b = q - r
17 c = r * q
18 d = r / q
19 e = c ** d
20
21 print()
22 print(q, "+", r, "=", a)
23 print(q, "-", r, "=", b)
24 print(r, "*", q, "=", c)
25 print(r, "/", q, "=", d)
26 print(c, "**", d, "=", e)
27

```

```

----jGRASP exec: python Variables09.py

33.3333 + 12.5 = 45.8333
33.3333 - 12.5 = 20.8333
12.5 * 33.3333 = 416.66625
12.5 / 33.3333 = 0.375000375000375
416.66625 ** 0.375000375000375 = 9.603324496267613

----jGRASP: operation complete.

```

We have seen several programs already in this chapter. While each program does demonstrate a new, specific Python feature, none of the programs so far have really been practical. Program **Variables10.py**, in Figure 4.22, is different. This is probably the first “practical” program in the textbook. From your math classes, you may remember that the formula for the area of a circle is πr^2 . Given the operators that we now know, we should be able to perform this calculation. The program has 3 variables. There is **r** for the *radius* which stores **12.5**; there is **pi** which stores **3.141592653589793**, basically; and there is **area** which will store the final result. Look at line 9. Do you see that the statement **area = pi * r**2** is Python’s way of saying “the area of a circle is πr^2 ?”

Figure 4.22

```
1 # Variables10.py
2 # The program demonstrates a
3 # practical use of variables.
4
5
6 r = 12.5
7 pi = 3.141592653589793
8
9 area = pi * r**2
10
11 print()
12 print("The area of a circle with radius",r,
13 "is",area)
```

NOTE: The last **print** command in this program does not actually word-wrap in **jGRASP**. It does here because of the limited space in the textbook. The same applies to the output below. Keep this in mind for future program examples.

```
----jGRASP exec: python Variables10.py


The area of a circle with radius 12.5 is
490.8738521234052

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

Beware of Hidden Operators in Mathematics	
Mathematics	Python Source Code
$5AB$	<code>5 * A * B</code>
$\frac{5}{7}$	<code>5 / 7</code>
$\frac{A + B}{A - B}$	<code>(A + B) / (A - B)</code>

Take a close look at the last example above. You may have thought that it should look like the example below. The mathematical expression on the right does not contain any parentheses. So why should the Python source code? The answer is *Mathematical Precedence*. You may know this as “Order of Operations” or simply “PEMDAS”. Without the parentheses, we are not dividing the entire numerator by the entire denominator. We are simply dividing B by A.

$\frac{A + B}{A - B}$	<code>A + B / A - B</code>
-----------------------	----------------------------



4.4 Non-Numeric Data Types

We already mentioned the fact that in computer science variables can be words (or compound words), but there is another major difference between variables in computer science and variable in math. In your math class, the variables always store a number. “What else would it store?” you may ask.

Strings

Consider this. Have you ever had to enter your name while on a computer? Where is your name stored? It is stored in a variable. Now, it is not an integer variable or a real number variable. Things like names, words, or even sentences are stored in string variables.

Program **Variables11.py**, in Figure 4.23, defines and initializes several string variables and then displays their values. It should come as no surprise that the value stored by a *string variable* is a *string literal*.

There are a couple things I want to point out here. Look at line 17. **zipCode** is not an integer variable storing the number **75,081**. It is a string variable storing the string literal **"75081"**. Also, look at line 12. **middleInitial** stores a single *character*, capital letter **'Q'**. In some languages, you are required to put strings in double quotes and characters in single quotes. Python does not make a distinction between “strings” and “characters”. To Python, they are all strings, even if they only store one character. Also, Python does not care if your string literals are inside single quotes or double quotes.

Figure 4.23

```
1 # Variables11.py
2 # The program shows that variables can also
3 # store text or string values likes words,
4 # phrases, names, addresses or characters.
5
6
7 word1 = "Hello"
8 word2 = "Goodbye"
9 phrase1 = "How are you?"
10 phrase2 = "I am fine."
11 firstName = "John"
```



```

12 middleInitial = 'Q'
13 lastName = "Public"
14 street = "811 Fleming Trail"
15 city = "Richardson"
16 state = "Texas"
17 zipCode = "75081"
18
19 print()
20 print("Words:",word1,word2)
21 print("Phrases:",phrase1,phrase2)
22 print()
23 print(firstName,middleInitial,lastName)
24 print(street)
25 print(city,state,zipCode)
26

```

```

----jGRASP exec: python Variables11.py

Words: Hello Goodbye
Phrases: How are you? I am fine.

John Q Public
811 Fleming Trail
Richardson Texas 75081

----jGRASP: operation complete.

```

NOTE: I have decided to take advantage of Python's flexibility and will continue to use double quotes for multi-character string literals and single quotes for single-character string literals.

Program **Variables12.py**, in Figure 4.24, demonstrates something called *String Concatenation*. By now you know what strings are, but “concatenation” is a different matter. Unfortunately, Computer Science is full of big, yucky terminology. Luckily, a lot of these terms are not that bad once you know what

they mean. For example, “string concatenation” simply means joining two or more strings together.

To demonstrate this, the program combines a **firstName** and a **lastName** to create a *full name* in three ways. Not that the plus sign (+) is used for this. In the first example, on line 8, the first and last names are combined with nothing in-between. This is why the first output is “**SuzySnodgrass**”. In the next example, on line 9, a space is placed in-between. You will see that the second output displays the name properly as “**Suzy Snodgrass**”. The final example, on line 10, switches things around and places the last name first, followed by a comma (,) and a space, and then the first name. This gives us “**Snodgrass, Suzy**”.

Figure 4.24

```
1 # Variables12.py
2 # This program demonstrates "String Concatenation"
3 # which is joining together 2 or more strings.
4
5 firstName = "Suzy"
6 lastName = "Snodgrass"
7
8 fullName1 = firstName + lastName
9 fullName2 = firstName + " " + lastName
10 fullName3 = lastName + ", " + firstName
11
12 print()
13 print(fullName1)
14 print(fullName2)
15 print(fullName3)
```

```
----jGRASP exec: python Variables12.py

SuzySnodgrass
Suzy Snodgrass
Snodgrass, Suzy

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

The plus sign (+) is a versatile little operator. When used with numbers (integers or real numbers) it performs *addition*. When used with strings, it performs *string concatenation*. This is reviewed with program **Variables13.py**, in Figure 4.25. Variable **number1** stored the sum of the integer values **100** and **200**. On the other hand, variable **number2** stores the concatenation of the string values **"100"** and **"200"**. These may look like integers as well; but remember, any text placed inside quotes is a *string literal* – even if that text looks like a number.

Something needs to be understood here about the plus (+) operator. The fact that it performs two jobs – adding numbers and concatenating strings – makes the plus sign (+) an *Overloaded Operator*. Consider yourself. You are a student. You may also be involved in sports, or fine arts, or both. You are a child of your parents and may even have younger siblings. So when you have to deal with homework, practice, rehearsal, chores and babysitting; you can definitely feel “overloaded”.

Figure 4.25

```
1 # Variables13.py
2 # Addition vs. Concatenation
3 # Use <+> with numbers and you get addition.
4 # Use <+> with strings and you get concatenation.
5 # Since <+> can do 2 different things, it is
6 # called an "Overloaded Operator".
7
8
9 number1 = 100 + 200
10 number2 = "100" + "200"
11
12 print()
13 print(number1)
14 print(number2)
15
```

```
----jGRASP exec: python Variables13.py

300
100200

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

String Concatenation

Concatenation is the joining together of two or more strings.

```
"Hello" + "World" = "HelloWorld"
"Hello" + " " + "World" = "Hello World"
"100" + "200" = "100200"
```

The plus operator (+) is used both for *arithmetic addition* and *string concatenation*. The same operator performs two totally different operations. This makes it an *overloaded operator*.

The next program is going to examine arithmetic addition and string concatenation even closer. Program **Variables14.py**, in Figure 4.26, has several examples of addition and concatenation. We will look first at the examples of “Adding Numbers.” Line 6 shows integers being added. Line 7 shows real numbers being added. Line 8 shows that integers and real numbers can be added together as well.

NOTE: When adding numbers, if both numbers are integers, the result is an integer as well; however, if either of the numbers is a real number the result will be a real number. This same rule applies to subtraction and multiplication as well. For division, it depends on the specific division operator that you are using.

Lines 17 and 18 show examples of “Concatenating (Joining) Strings.” In both examples 2 words are joined together with a space in-between. In line 17, the space is at the beginning of second word. In line 18, the space is at the end of the first word. A couple program examples ago, the space a completely separate string between the 2 words. Any of these three techniques work.

Figure 4.26

```
1 # Variables14.py
2 # More Addition and Concatenation
3
4
5 # Adding Numbers
6 sum1 = 19 + 96      # Integers Only
7 sum2 = 2.7 + 7.11  # Real Numbers Only
8 sum3 = 68 + 4.29   # Integers and Real Numbers
```

```

9
10 print()
11 print(sum1)
12 print(sum2)
13 print(sum3)
14
15
16 # Concatenating (Joining) Strings
17 greeting = "Hello" + " There"
18 name = "Tom " + "Jones"
19
20 print()
21 print(greeting)
22 print(name)

```

```

----jGRASP exec: python Variables14.py

115
9.81
72.29

Hello There
Tom Jones

----jGRASP: operation complete.

```

Booleans

More than a century ago there was a mathematician, George Boole, who developed a new branch of mathematics. His mathematics involved neither arithmetic nor Algebra. It did involve logical statements which are either **true** or **false**. This new branch of mathematics was named *Boolean Algebra* after its founder. At the time, no one really saw the point of this. Several decades passed. One day, during the time of large, vacuum tube driven computers like the ENIAC, someone stumble upon his George Boole's note. Then there was a realization. In computers, electricity is either *on* or *off*. This gives us **1** or **0**. Well, Boolean

Algebra, where everything is either **True** or **False** is a perfect match. It eventually became the mathematical foundation for electronic digital computer design. As a consequence, most programming languages today have some sort of *Boolean* data type.

Program **Variables15.py**, in Figure 4.27, demonstrates a couple *Boolean* variables. The first, **passingEnglish**, stores **True**. The second, **passingHistory**, stores **False**. Both values are then displayed.

Figure 4.27

```
1 # Variables15.py
2 # The program shows that variables can
3 # also store Boolean values which are
4 # either <True> or <False>.
5
6
7 passingEnglish = True
8 passingHistory = False
9
10 print()
11 print(passingEnglish)
12 print(passingHistory)
13
```

```
----jGRASP exec: python Variables15.py

True
False

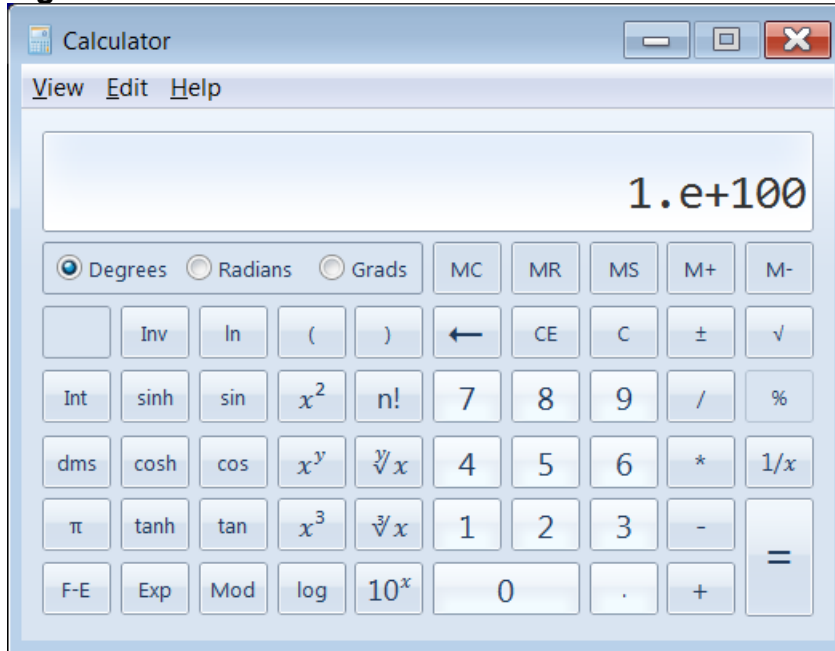
----jGRASP: operation complete.
```

Right now you may be thinking, “Is that it?” It seems like we have a data type whose only purpose is to pay homage to George Boole. Well, there is much more to it than that. Boolean variables are very useful in programming. The problem is right now you are not yet ready to see a practical program which uses Boolean variables. That will have to wait for Chapter 11; which we will get to sometime in the second semester. For now, all you need to know about Boolean variables is that they can only store either **True** or **False**.

Scientific Notation

Chances are, you have seen *scientific notation* before. Maybe it was your math class or your science class. Maybe, you used a calculator to calculate something that had an enormously large answer. Figure 4.28 shows the result of entering 10^{100} (which is a *google* by the way) into the calculator app on my laptop. The **e** means exponent. This is short for $1 * 10^{100}$.

Figure 4.28



Program **Variables16.py**, shown in Figure 4.29 demonstrates a couple ways that we can do scientific notation in Python. The first way, shown in lines 9 & 10, works but it is not the official scientific notation syntax. The second, more explicit way, shown on lines 14 & 15 is preferred. Not only is this syntax shorter, you will see that it is very similar to the calculator example from figure 4.28.

If you take a look at the output, you should notice that regardless of how the expression was typed, if the result is small enough, it will be displayed normally. If the result is too to be displayed normally, it will be displayed using scientific notation. There is something else that you should notice. Mathematically, the expressions on lines 9 & 14 are equivalent. They should produce identical results. What needs to be understood is that real number calculations on computers are not always exact. They are typically accurate to a certain number of decimal places. Python's real numbers are good for about 15 decimal places. Now, there ways to deal with this, but that goes beyond the scope of this course, so we are not going to worry about it right now.

Figure 4.29

```
1 # Variables16.py
2 # The program demonstrates 2 ways to do "Scientific Notation".
3 # The second, more explicit method is generally preferred.
4 # It also shows that the accuracy of real #s is not perfect.
5
6
7 print()
8
9 print(13.7 * 10 ** 9)
10 print(6.02 * 10 ** 23)
11
12 print()
13
14 print(13.7e9)
15 print(6.02e23)
16
```

```
----jGRASP exec: python Variables16.py

13700000000.0
6.019999999999999e+23

13700000000.0
6.02e+23

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


The type Command

Now that we have seen several different data types, we are going to look at the **type** command. **type** will give you the data “type” of a variable or value. For example: **type(2)** will return `<class 'int'>` where **type(2.5)** will return `<class 'float'>`. The term “float” seems weird until you understand that another way to say “real number” is “floating point number”. This and other data types are demonstrated in program **Variables17.py**, shown in Figure 4.30. Note that this program actually proves what was said earlier. Python does not make a distinction between strings and characters. Both give a result of `<class 'str'>`.

Figure 4.30

```
1 # Variables17.py
2 # The program demonstrates the <type> command which
3 # will give you the data type of a particular value.
4 # NOTE: Python does not distinguish between string
5 # and character values. Both are <str>.
6 # ALSO: Another word for a "real number" is a
7 # "floating point number". When scientific notation
8 # is used, the resulting value is a <float>.
9
10
11 a = 7
12 pi = 3.141592653589793
13 name = "John Smith"
14 middleInitial = 'Q'
15 passing = True
16 mole = 6.02e23
17
18 print()
19 print(type(a))
20 print(type(pi))
21 print(type(name))
22 print(type(middleInitial))
23 print(type(passing))
24 print(type(mole))
25
```

```
----jGRASP exec: python Variables17.py

<class 'int'>
<class 'float'>
<class 'str'>
<class 'str'>
<class 'bool'>
<class 'float'>

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



Type Casting

There are times when you need to convert a value from one type to another. One way to do this is called *Type Casting* which is demonstrated by program **Variables18.py**, shown in Figure 4.31. By using the commands **int**, **float**, **str** and **bool**. Note that these are the same words that appear in the output of the previous program. We can force one data type to be treated as another. If you look at the output, you see that when **int** was used with **pi**, it just became **3**. While line 16 adds the values of **a** and **pi**, line 17 concatenates them. This is made possible by the **str** command which converted both values to strings. When **True** is converted to an **int**, it becomes **1**. Converting **0** to a **bool** gives us **False**.

Figure 4.31

```
1 # Variables18.py
2 # The program demonstrates how you can use "Type Casting"
3 # to force one type of value to be treated like another.
4
5
6 a = 7
7 pi = 3.141592653589793
8 name = "John Smith"
9 middleInitial = 'Q'
10 passing = True
11 mole = 6.02e23
12
13 print()
14 print(float(a))
15 print(int(pi))
16 print(a + pi)
17 print(str(a) + str(pi))
18 print(int(passing),float(False))
19 print(bool(1.0),bool(0))
20
```

```
----jGRASP exec: python Variables18.py

7.0
3
10.141592653589793
73.141592653589793
1 0.0
True False

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

4.5 Shortcuts

A numbers of languages like C++, Java and Python have some shortcuts built in for certain frequently used operations. We will look at a few of these shortcuts in this section.

Arithmetic Operator Shortcuts

Program **Shortcuts01.py**, in Figure 4.32, starts by showing the “long way” to do several different calculations. In each of these calculations, we are doing something to the current value of **x** (adding to it, subtracting from it, etc.)

Figure 4.32

```
1 # Shortcuts01.py
2 # This program demonstrates the "long way"
3 # to do several calculations.
4
5
6 print()
7 x = 70
8 print("x =",x)
9 x = x + 7
10 print("x =",x)
11 x = x - 7
12 print("x =",x)
13 x = x * 7
14 print("x =",x)
15 x = x / 7
16 print("x =",x)
17 x = x // 7
18 print("x =",x)
19 x = x % 7
20 print("x =",x)
21 x = x ** 7
22 print("x =",x)
```

```
----jGRASP exec: python Shortcuts01.py

x = 70
x = 77
x = 70
x = 490
x = 70.0
x = 10.0
x = 3.0
x = 2187.0

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

Program **Shortcuts02.py**, in Figure 4.33, demonstrates the “shortcut” way to do the same calculations as the previous program.

Figure 4.33

```
1 # Shortcuts02.py
2 # This program demonstrates the "shortcut"
3 # way to do the same calculations as the
4 # previous program.
5
6
7 print()
8 x = 70
9 print("x =",x)
10 x += 7
11 print("x =",x)
12 x -= 7
13 print("x =",x)
14 x *= 7
15 print("x =",x)
```

```

16 x /= 7
17 print("x =",x)
18 x //= 7
19 print("x =",x)
20 x %= 7
21 print("x =",x)
22 x **= 7
23 print("x =",x)
24

```

This program has the exact same output as the previous program.

Arithmetic Operator Shortcuts

Long Way	Shortcut	Meaning
x = x + 7	x += 7	Add 7 to the current value of x.
x = x - 7	x -= 7	Subtract 7 from the current value of x.
x = x * 7	x *= 7	Multiply the current value of x by 7.
x = x / 7	x /= 7	Divide the current value of x by 7 using real number division.
x = x // 7	x //= 7	Divide the current value of x by 7 using integer division.
x = x % 7	x %= 7	Divide the current value of x by 7 using remainder division.
x = x ** 7	x **= 7	Take the current value of x to the 7 th power.

Chaining

The meaning of “chaining” will be explained in the next program example. For now, program **Shortcuts03.py**, in Figure 4.34, shows the “long way” to do assign the same integer value to several variables.

Figure 4.34

```
1 # Shortcuts03.py
2 # This program demonstrates the "long way" to
3 # assign the same value to several variables.
4
5
6 a = 25
7 b = 25
8 c = 25
9 d = 25
10 e = 25
11
12 print()
13 print(a,b,c,d,e)
14
```

```
----jGRASP exec: python Shortcuts01.py

25 25 25 25 25

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

Program **Shortcuts04.py**, in Figure 4.35, demonstrates the “shortcut” way to assign the same value to the same variables as the previous program. What use to be 5 separate program statements is now one single program statement. This type of shortcut is called “chaining” because multiple assignment statements are now “chained” into one.

Figure 4.35

```
1 # Shortcuts04.py
2 # This program demonstrates the "shortcut" way
3 # to assign the same value to several variables.
4 # This particular shortcut is called "Chaining".
5
6
7 a = b = c = d = e = 25
8
9 print()
10 print(a,b,c,d,e)
```

This program has the exact same output as the previous program.

Program **Shortcuts05.py**, in Figure 4.36, demonstrates that *chaining* works with other data types as well.

Figure 4.36

```
1 # Shortcuts05.py
2 # This program demonstrates that the chaining
3 # shortcut works with other data types as well.
4
5
6 a = b = c = d = e = f = g = 2.5
7 print()
8 print(a,b,c,d,e,f,g)
9
10 p = q = r = s = "Hello"
11 print()
12 print(p,q,r,s)
13
14 j = k = True
15 print()
16 print(j,k)
```



```
----jGRASP exec: python Shortcuts05.py

2.5 2.5 2.5 2.5 2.5 2.5 2.5

Hello Hello Hello Hello

True True

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

Using += with Strings

ThProgram **Shortcuts06.py**, in Figure 4.37, demonstrates that the plus equals operator (+=) also works with strings. Similar to the way += *adds* a number to what is currently stores in an numeric variable, it can also *concatenate* a string to the end of what it currently stored in a string variable. Since += can perform 2 different operations, it also is an *overloaded operator*.

Figure 4.37

```
1 # Shortcuts06.py
2 # This program demonstrates that the <+=>
3 # shortcut can be used with strings to join
4 # a string value to the end of an existing
5 # string. This means that <+=> is also an
6 # "Overloaded Operator".
7
8
9 name = "John"
10 name += "Public"
11 print()
12 print(name)
13
```

```
14 name = "John"
15 space = ' '
16 name += space
17 name += "Public"
18 print()
19 print(name)
20
21 name = "John"
22 name += space
23 name += 'Q'
24 name += '.'
25 name += space
26 name += "Public"
27 print()
28 print(name)
29
```

```
----jGRASP exec: python Shortcuts06.py

JohnPublic

John Public

John Q. Public

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

4.6 Swapping Variable Values

It is often necessary to swap the values stored in two different variables. Next semester, we will look at more advanced topics like *Sorting Algorithms*. Consider that you have a large amount of data that needs to be put in order. In order to do this, you need to be able to move the individual pieces of data around. At its core, this all comes down to the ability to swap two values.

The next 3 program examples will deal with swapping integer values. All 3 programs have the same 2 variables, **number1** and **number2**. Both variables get initial values and these are displayed. Then each program will try to *swap* the values in a different way. To see if the swap was successful, the values will be displayed again.

Program **SwappingValues01.py**, in Figure 4.38, is actually an example of what not to do. It may seem logical to essentially tell the computer, “*The first number becomes the second and the second number becomes the first.*” This may make sense to us. Keep in mind that the computer executes the program in sequence. The moment we tell the computer “*The first number becomes the second.*” we have just lost the value of the first number.

Figure 4.38

```
1 # SwappingValues01.py
2 # Swapping the values of 2 variables
3 # The WRONG way
4
5 print()
6 number1 = 100
7 number2 = 200
8 print(number1,number2)
9
10 number1 = number2
11 number2 = number1
12 print(number1,number2)
```

```
----jGRASP exec: python SwappingValues01.py
100 200
200 200
----jGRASP: operation complete.
```

To better understand the problem, consider this analogy: Suppose you have 2 glasses. One contains milk and the other contains Orange Juice. For whatever reason, you decide you need to swap the contents of the 2 glasses. How do you do this?



If you attempt to apply the logic from the **SwappingValues01.py** example, you would pour the milk glass into the OJ glass, and then pour the OJ glass into the milk glass. The result is a royal mess. If you want to properly swap the contents of these 2 glasses, what do you need? What must you have? You must have a third glass.



Now you can pour the milk in the empty glass, then pour the OJ into the glass that use to have milk, and then finally pour the milk into the glass that use to have OJ. To swap the values of 2 variables. We need something like this third, empty glass. We need a variable that can temporarily hold the value of the first number.

Program **SwappingValues02.py**, in Figure 4.39, employs the “Third Glass” technique described in the previous 2 paragraphs. In this case, the *third glass* is actually a *third variable* called **temp**. It is a common practical to name a variable **temp** when it is used to store something *temporarily*.

Figure 4.39

```
1 # SwappingValues02.py
2 # Swapping the values of 2 variables
3 # Using a temporary variable
4 # This technique works in any language.
5
6
7 print()
8 number1 = 100
9 number2 = 200
10 print(number1,number2)
11
12 temp = number1
13 number1 = number2
14 number2 = temp
15 print(number1,number2)
16
```

```
----jGRASP exec: python SwappingValues02.py

100 200
200 100

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

There is actually a simpler way to swap the values of 2 variables. You may wonder why I bothered to mention the more complicated method if a simpler way exists. I did this for a very important reason. The simpler approach only works in Python. The more complicated approach of using a temporary variable, demonstrates in the previous program example, works in any programming language.

Program **SwappingValues03.py**, in Figure 4.40, demonstrates Python's *Simultaneous Assignment*. By using this shortcut, we can swap in a single step. Remember, this swapping shortcut only works in Python.

Figure 4.40

```
1 # SwappingValues03.py
2 # Swapping the values of 2 variables
3 # Using "Simultaneous Assignment"
4 # This "shortcut" only works in Python.
5
6
7 print()
8 number1 = 100
9 number2 = 200
10 print(number1,number2)
11
12 number1,number2 = number2,number1
13 print(number1,number2)
14
```

```
----jGRASP exec: python SwappingValues03.py

100 200
200 100

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

The past 3 program examples all dealt with swapping integers values. Integers are not the only data types that can be swapped. You can swap the value of 2 variables regardless of their data type. Program **SwappingValues04.py**, in Figure 4.41, demonstrates this by swapping the values of 2 string variables: **name1** and **name2**. The program actually swaps them twice. The first time a **temp** variable is used. The second uses “Simultaneous Assignment”. Note that swapping 2 variables a second time returns both variables to their original values.

Figure 4.41

```
1 # SwappingValues04.py
2 # Swapping the values of 2 string variables
3 # You can swap other data types as well.
4 # NOTE: If you swap 2 variables twice, they
5 # wind up with their original values.
6
7
8 print()
9 name1 = "Tom"
10 name2 = "Sue"
11 print(name1,name2)
12
13 # first swap
14 temp = name1
15 name1 = name2
16 name2 = temp
17 print(name1,name2)
18
19 # second swap
20 name1,name2 = name2,name1
21 print(name1,name2)
22
```

```
----jGRASP exec: python SwappingValues04.py

Tom Sue
Sue Tom
Tom Sue

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

4.7 Documenting Your Programs

Program documentation is a major big deal. Perhaps to you it is a big deal because some irritating computer science teacher keeps after you to document your programs. There also seems to be an irritating author of this Java book who should have stayed in Europe and harass European kids rather than get on a soap-box in the United States.

You will not fully appreciate the need for documentation in a first year course. Once the programs you write reach a certain size, it is not possible to test, debug or alter such programs without proper documentation.

The first form of program documentation is to use comments. You were shown how to create *single-line comments* and *multi-line comments* back in Chapter III. When a program uses variables, another form of program documentation is possible. To illustrate the need for program documentation the next program, **Documentation01.py**, shown in Figure 4.42, has no program documentation whatsoever. When you look at this program, do you have any clue what it does? Even the program's output, tells you little about what is happening with this program.

Figure 4.42

```
1 # Documentation01.py
2 # This is an example of a poorly written
3 # program with single-letter variables.
4 # Do you have any idea what this program does?
5
6
7 a = 35
8 b = 8.75
9 c = a * b
10 d = c * 0.29
11 e = c - d
12
13 print()
14 print("a =",a)
15 print("b =",b)
16 print("c =",c)
17 print("d =",d)
18 print("e =",e)
```



```
----jGRASP exec: python Documentation01.py

a = 35
b = 8.75
c = 306.25
d = 88.8125
e = 217.4375

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

Program **Documentation01.py** makes no sense because it uses *single-letter variables*. Several decades ago, this actually was the proper way to program. Back in the 1960s and 1970s computer memory was scarce and very expensive. Programmers had to do anything they could save every byte possible. This was so extreme that when a year needed to be stored, they would only store the last 2 digits. For example, **1968** was simply stored as **68**. This is what led to the whole Y2K mess just before we hit the *Year 2000*.

Today computer memory is abundant and very cheap. The need to save every possible byte of memory no longer exists. Program *readability* is now a big issue. This is another part of program documentation. Program **Documentation02.py**, shown in Figure 4.43, does the exact same thing as the previous program. There is only one difference. The variables now have different names. Does the program make more sense now?

Figure 4.43

```
1 # Documentation02.py
2 # This program does exactly the same thing
3 # as the previous program. By using self-
4 # documenting variables, the program is
5 # much easier to read and understand.
6
7
8 hoursWorked = 35
9 hourlyRate = 8.75
10 grossPay = hoursWorked * hourlyRate
11 deductions = grossPay * 0.29
12 netPay = grossPay - deductions
```

```

13
14 print()
15 print("Hours Worked: ",hoursWorked)
16 print("Hourly Rate:  ",hourlyRate)
17 print("Gross Pay:    ",grossPay)
18 print("Deductions:   ",deductions)
19 print("Net Pay:      ",netPay)
20

```

```

----jGRASP exec: python Documentation02.py

Hours Worked:  35
Hourly Rate:   8.75
Gross Pay:     306.25
Deductions:    88.8125
Net Pay:       217.4375

----jGRASP: operation complete.

```

NOTE: The output of this program shows dollar amounts that are not rounded to the nearest penny. You will learn how to format your output in a later chapter.

Program **Documentation02.py** should have made more sense than the previous program because the variables are now *self-commenting*. A *self-commenting variable* is a variable whose name describes what the variable is used for. In the previous program the variable for *Net Pay* is **e**. In this program, the variable for *Net Pay* is **netPay**. This is why this program is so much easier to read and understand. The age of single-letter variables is gone. Variables should now be *words* like **deductions** or *compound words* like **hoursWorked**.

Earlier, it was mentioned that *comments* are part of program documentation. Just because your program has variables that are *self-commenting*, it does not mean there is no need for well-placed comments in your program as well. At the start of a program it is a good practice to have a heading that explains some general information about the program. At this place it makes sense to use a *multi-line comment*. There are other places in the program where a quick *single-line comment* provides some needed information.

Program **Documentation03.py**, in Figure 4.44, demonstrates both types of comments. In particular, note how the comments extend the meaning of the self-commenting variables. For instance, the identifier **hoursWorked** is descriptive, but it is the comment which explains that it means the number of hours worked per week.

Figure 4.44

```
1 # Documentation03.py
2 # This program adds a multi-line comment at
3 # the beginning to help explain the program.
4 # Several short single-line comments are also
5 # added to provide more detail for each variable.
6
7
8 """
9 Payroll Program
10 Written by Leon Schram 09-09-09
11
12 This program takes the hours worked and hourly rate
13 of an employee and computes the gross pay earned.
14 Federal deductions are computed as 29% of gross pay.
15 Finally the take-home pay or net pay is computed by
16 subtraction deductions from gross pay.
17 """
18
19
20 hoursWorked = 35    # hours worked per week
21 hourlyRate = 8.75   # pay rate earned per hour
22 grossPay = hoursWorked * hourlyRate    # total earnings
23 deductions = grossPay * 0.29           # federal tax
24 netPay = grossPay - deductions          # take home pay
25
26 print()
27 print("Hours Worked: ",hoursWorked)
28 print("Hourly Rate:  ",hourlyRate)
29 print("Gross Pay:    ",grossPay)
30 print("Deductions:   ",deductions)
31 print("Net Pay:      ",netPay)
32
```

This program has the exact same output as the previous program.

4.8 More Syntax Errors

We have seen a number of compile errors over the past couple chapters. We are actually going to look at a few more. As you become more familiar with different types of syntax errors, you will find it easier to debug your own programs.

Program **MoreErrors01.py**, in Figure 4.45, is very similar to the earlier program **Variables10.py**. One change has been made. The **radius** variable is now misspelled when it is defined.

Figure 4.45

```
1 # MoreErrors01.py
2 # This program demonstrates what happens when
3 # you misspell a variable.
4 # Note that the error messages do not always
5 # identify the correct location of the error.
6
7
8 pi = 3.141592653589793
9
10 radiusu = 12.5
11
12 circleArea = pi * radius ** 2
13
14 print()
15 print("A circle with a radius of",radius,
"has an area of",circleArea)
```

```
----jGRASP exec: python MoreErrors01.py
Traceback (most recent call last):
  File "MoreErrors01.py", line 12, in <module>
    circleArea = pi * radius ** 2
NameError: name 'radius' is not defined

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

There is a very important concept that this program demonstrates. Note that the error is caused by using the variable **radius** in line 12. The actual error is in line 10 where **radiusu** is misspelled.

Program **MoreErrors02.py**, in Figure 4.46, is also very similar to the earlier program **Variables10.py**, but this time a different change has been made. The issue now is *case sensitivity*. On line 13 there is a **print** statement that display several things including the value of **circleArea**; however, it does not actually say **circleArea**, it says **circlearea**. The first ‘A’ is not capitalized. To the computer this is no different from misspelling the variable and you get the same error,

Figure 4.46

```
1 # MoreErrors02.py
2 # This program demonstrates what happens
3 # when you do not follow case-sensitivity.
4
5
6 pi = 3.141592653589793
7
8 radius = 12.5
9
10 circleArea = pi * radius ** 2
11
12 print()
13 print("A circle with a radius of",radius, "has
14 an area of",circlearea)
```

```
----jGRASP exec: python MoreErrors01.py
Traceback (most recent call last):
  File "MoreErrors02.py", line 13, in <module>
    print("A circle with a radius of",radius,
"has an area of",circlearea)
NameError: name 'circlearea' is not defined

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

Earlier in the chapter, we saw examples of adding numbers. Later, we saw examples of concatenating strings. What happens if you wish to combine a number and a string? You may ask “Why would we ever need to do that?” Consider your street address. There is the number of your house, condo, or apartment complex. This number is an integer. Then there is the name of your street, which is a string. Program **MoreErrors03.py**, in Figure 4.47, attempts to do this. When you try to run the program, you see the computer is not happy. We can “add” numbers and we can “concatenate” strings, but we cannot “combine” a number and a string.

Figure 4.47

```
1 # MoreErrors03.py
2 # The Issue with Addresses
3 # Combining numbers & strings does not work
4
5
6 houseNumber = 811
7 streetName = " Fleming Trail"
8
9 streetAddress = houseNumber + streetName
10
11 print()
12 print(streetAddress)
13
```

```
----jGRASP exec: python MoreErrors03.py
Traceback (most recent call last):
  File "MoreErrors03.py", line 9, in <module>
    streetAddress = houseNumber + streetName
TypeError: unsupported operand type(s) for +:
'int' and 'str'

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

So is that it? Does this mean that creating a street address is impossible? Not at all. All we need to do is convert the **houseNumber** to a string value. Once we do that, there is no problem concatenating the two strings. This is a perfect example of where *Type Casting* is useful and it is precisely what done by the **str** command in program **MoreErrors04.py**, shown in Figure 4.43.

Figure 4.43

```
1 # MoreErrors04.py
2 # Fixing the Address Issue
3 # By Type Casting the <houseNumber> as a string,
4 # the computer is able to concatenate it to the
5 # <streetName> to complete the <streetAddress>.
6
7
8 houseNumber = 811
9 streetName = " Fleming Trail"
10 streetAddress = str(houseNumber) + streetName
11
12 print()
13 print(streetAddress)
14
```

```
----jGRASP exec: python MoreErrors04.py

811 Fleming Trail

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

4.9 Other Types of Errors

While syntax errors can be annoying, they are actually the easiest type of error to fix. This is because all syntax errors are caught by the interpreter. There are other types of errors that can be far more annoying.

Run-time Errors

Program **MoreErrors05.py**, in Figure 4.44, demonstrates a *Run-time Error*. Run-time errors are not caught by the interpreter. The program is syntactically correct. It executes, and then it is given an instruction it just cannot handle so it CRASHES. Program **MoreErrors04.py** defines 3 integers. The first 2 definitions, **a = 1** and **b = 0** are fine. The problem is the third which says **c = a / b**. Some of you may already see the issue, but the interpreter does not. Remember the interpreter is only checking for syntax. This statement divides one integer variable by another and stores the result in a third. That is perfectly valid Python syntax.

Now the program executes. You see the proof of this because it does display “**Execution Begins**”. Then it defines and initializes variables **a** and **b**. When it tries to define and initialize variable **c** the problem occurs. The program is dividing by the value of **b** which is **0**. This triggers a “**division by zero**” error and the program crashes.

Figure 4.44

```
1 # MoreErrors05.py
2 # This program demonstrates a "Run-time Error".
3 # While the program has no Syntax Errors & does
4 # execute, it CRASHES when it attempts to divide by 0
5
6
7 print()
8 print("Execution Begins")
9 print()
10
11 a = 1
12 b = 0
13 c = a / b
14
15 print("c =",c)
```



```
----jGRASP exec: python MoreErrors05.py

Execution Begins

Traceback (most recent call last):
  File "MoreErrors05.py", line 13, in <module>
    c = a / b
ZeroDivisionError: division by zero

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

Logic Errors

As annoying as run-time errors can be, there is one type of error that is worse. These are *Logic Errors*. Logic errors are the most difficult to fix because as far as the computer is concerned, there is no problem. The program executes without crashing. The problem is the program is just not doing what you want it to do because you did not write it correctly. Since there is no syntax error or run-time error, the computer provided no error message to help you.

Program **MoreErrors06.py**, in Figure 4.45, demonstrates a common logic error. The program attempts to find the average of **70, 80, 90** and **100**. So we need to add up the number and divide by **4**. When we run the program, we get an answer of **265**. Now hopefully you are not thinking to yourself, “Well that’s what the computer says, so it must be right.” A little bit of *Math Sense* needs to be used here. Is it possible for the average of a group of numbers to be larger than the largest number in the list? No, definitely not. The answer of **265** is completely preposterous. How did this error occur? Remember *Order of Operations* a.k.a. “PEMDAS”? The problem is we want the computer to add the 4 numbers and then divide by 4. That is NOT what the program is doing. Since the computer follows *Order of Operations* it first divides the value of **num4**, which is **100**, by **4** and then it adds the other 3 numbers.

Figure 4.45

```
1 # MoreErrors06.py
2 # This program demonstrates a "Logic Error".
3 # These can be the most frustrating errors
4 # because no error is actually detected.
5 # The program executes just fine.
6 # It just does not do what you want it to do.
7 # In this case the average is wrong because
8 # "Order of Operations" was not considered.
9
10
11 num1 = 70
12 num2 = 80
13 num3 = 90
14 num4 = 100
15
16 average = num1 + num2 + num3 + num4 / 4
17
18 print()
19 print("Average:", average)
20
```

```
----jGRASP exec: python MoreErrors06.py

Average: 265.0

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

To fix this logic error, we need to insert parentheses in this manner:

```
average = (num1 + num2 + num3 + num4) / 4
```

This is done in the program **MoreErrors07.py**, the final program of the chapter, shown in Figure 4.46. Note that now the **average** is correct.

Figure 4.46

```
1 # MoreErrors07.py
2 # This program fixes the PEMDAS issue
3 # of the previous program by adding a
4 # strategic set of parentheses.
5
6
7 num1 = 70
8 num2 = 80
9 num3 = 90
10 num4 = 100
11
12 average = (num1 + num2 + num3 + num4) / 4
13
14 print()
15 print("Average:", average)
16
```

```
----jGRASP exec: python MoreErrors07.py

Average: 85.0

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

Remember Order of Operations a.k.a. PEMDAS!

Please	Parentheses
Excuse	Exponents
My	Multiplication
Dear	Division
Aunt	Addition
Sally	Subtraction

The 3 Kinds of Errors

Syntax Errors

These are errors in the syntax of the program.
Simple typos cause many syntax errors.
The interpreter catches ALL syntax errors.
The syntax error message is not always an accurate indicator of the cause of the error, or its location, or both.

Run-time Errors

These errors are triggered during program execution when an attempt is made to do something improper.
Example: Dividing by 0.
Essentially, the program will execute and then it will CRASH.

Logic Errors

These are the most frustrating errors because there are no error messages. The program executes without crashing; however, the program does not do what you want it to do.
This means the logic of your program is not correct.

Asking For Help – Important Note

There will be times that your program does not work, and you will need to ask for help from your computer science teacher. While you may not know the exactly what is wrong with your program, you should at least be able to tell your teacher whether you have a syntax, run-time or logic error.

4.10 Output Programs, Slides, Exercises & Quizzes

By now, you should be use to how the program examples for each chapter are organized in a folder with subfolders. Figure 4.47 shows how this is done for this chapter. If you look at the bottom, you will see there is one more subfolder that we have not done yet. This is the special “Output” section. Several chapters in this textbook will end with a special “Output” section. This is the first. Unlike the earlier sections, you will not see these programs explained in this or any chapter. You will see them in a separate PowerPoint presentation. For example, you know that in the **Slides04** folder you will find the **Slides04.pptx** presentation for this chapter. What you will also find is the **Slides04-Output.pptx** presentation. See Figure 4.48. This presentation does not introduce new material. It simply displays each of these “output” programs. The mission is for students to figure out the “output” of each of the programs in the presentation.

NOTE: Teachers also have a special file called **Slides04-OutputKey.pptx** which displays the output of each program and, where necessary, “shows the work”.

The intention of this last section is to be something of a review. In several chapters, there is also a corresponding homework assignment, in this case **OutputExercises04.docx** in the **Exercises04** folder, shown in Figure 4.49. Also, when you do “Output Exercises” for homework one night, you should expect an “Output Quiz” is coming soon (typically the next class).

Figure 4.47

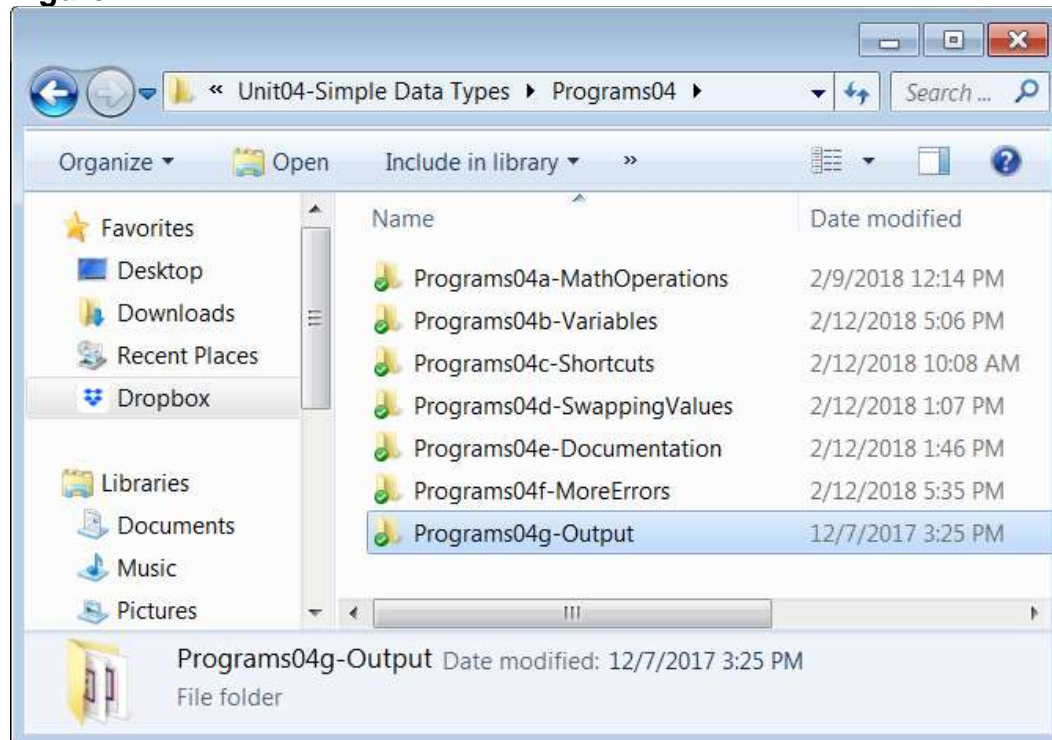


Figure 4.48



Figure 4.49

