2.2. Values and Data Types

A **value** is one of the fundamental things — like a word or a number — that a program manipulates. Some values are 5 (the result when we add 2 + 3), and "Hello, World!". These objects are classified into different classes, or data types: 4 is an integer, and “Hello, World!” is a string, so-called because it contains a string or sequence of letters. You (and the interpreter) can identify strings because they are enclosed in quotation marks.

We can specify values directly in the programs we write. For example we can specify a number as a **literal** just by (literally) typing it directly into the program (e.g., 5 or 4.32). In a program, we specify a word, or more generally a string of characters, by enclosing the characters inside quotation marks (e.g., "Hello, World!").

During execution of a program, the Python interpreter creates an internal representation of literals that are specified in a program. It can then manipulate them, for examply by multiplying two numbers. We call the internal representations **objects** or just **values**.

You can’t directly see the internal representations of values. You can, however, use the print function to see a printed representation in the output window.

The printed representation of a number uses the familiar decimal representation (reading [Roman Numerals](http://en.wikipedia.org/wiki/Roman_numerals) is a fun challenge in museums, but thank goodness the Python interpreter doesn’t present the number 2014 as MMXIV in the output window). Thus, the printed representation of a number shown in the output window is the same as the literal that you specify in a program.

The printed representation of a character string, however, is not exactly the same as the literal used to specify the string in a program. For the literal in a program, you enclose the string in quotation marks. The printed representation, in the output window, omits the quotation marks.

**Note**

**Literals** appear in programs. The Python interpreter turns literals into **values**, which have internal representations that people never get to see directly. **Outputs** are external representations of values that appear in the output window. When we are being careful, we will use the terms this way. Sometimes, however, we will get a little sloppy and refer to literals or external reprsentations as values.

Numbers with a decimal point belong to a class called **float**, because these numbers are represented in a format called *floating-point*. At this stage, you can treat the words *class* and *type* interchangeably. We’ll come back to a deeper understanding of what a class is in later chapters.

You will soon encounter other types of objects as well, such as lists and dictionaries. Each of these has its own special representation for specifying an object as a literal in a program, and for displaying an object when you print it. For example, list contents are enclosed in square brackets [ ]. You will also encounter some more complicated objects that do not have very nice printed representations: printing those won’t be very useful.

**Check your understanding**

data-2-1: What appears in the output window when the following statement executes?

**print**("Hello World!")

Top of Form

A. Nothing is printed. It generates a runtime error.  
B. "Hello World!"  
C. Hello World!

Bottom of Form

# 2.3. Operators and Operands

You can build complex expressions out of simpler ones using **operators**. Operators are special tokens that represent computations like addition, multiplication and division. The values the operator works on are called **operands**.

The following are all legal Python expressions whose meaning is more or less clear:

20 + 32

5 \*\* 2

(5 + 9) \* (15 - 7)

The tokens +, -, and \*, and the use of parentheses for grouping, mean in Python what they mean in mathematics. The asterisk (\*) is the token for multiplication, and \*\* is the token for exponentiation. Addition, subtraction, multiplication, and exponentiation all do what you expect.

Remember that if we want to see the results of the computation, the program needs to specify that with the word print. The first three computations occur, but their results are not printed out.

In Python 3, which we will be using, the division operator / produces a floating point result if the result is not an integer (e.g., 1/2). If you want truncated division, you can use the // operator.

print(9/)

print(9 // 5)

The **modulus operator**, sometimes also called the **remainder operator** or **integer remainder operator** works on integers (and integer expressions) and yields the remainder when the first operand is divided by the second. In Python, the modulus operator is a percent sign (%). The syntax is the same as for other operators.

print(7 // 3) # This is the integer division operator

print(7 % 3) # This is the remainder or modulus operator

In the above example, 7 divided by 3 is 2 when we use integer division and there is a remainder of 1.

The modulus operator turns out to be surprisingly useful. For example, you can check whether one number is divisible by another—if x % y is zero, then x is divisible by y. Also, you can extract the right-most digit or digits from a number. For example, x % 10 yields the right-most digit of x (in base 10). Similarly x % 100 yields the last two digits.

**Check your understanding**

data-3-1: What value is printed when the following statement executes?

**print**(18 / 4)

Top of Form

A. 4.5  
B. 5  
C. 4  
D. 4.0  
E. 2

Bottom of Form

data-3-2: What value is printed when the following statement executes?

**print**(18.0 // 4)

Top of Form

A. 4.5  
B. 5  
C. 4  
D. 4.0  
E. 2

Bottom of Form

data-3-3: What value is printed when the following statement executes?

**print**(18 % 4)

Top of Form

A. 4.25  
B. 5  
C. 4  
D. 2

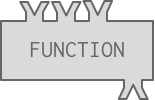
# 2.4. Function Calls

The Python interpreter can compute new values with function calls. You are familiar with the idea of functions from high school algebra. There you might define a function f by specifying how it transforms an input into an output, f(x) = 3x + 2. Then, you might write f(5) and expect to get the value 17.

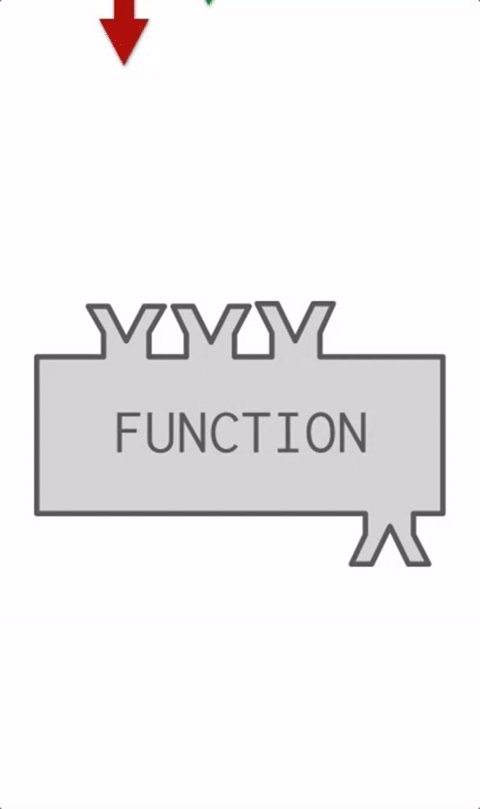
Python adopts a similar syntax for invoking functions. If there is a named function foo that takes a single input, we can invoke foo on the value 5 by writing foo(5).

There are many built-in functions available in Python. You’ll be seeing some in this chapter and the next couple of chapters.

Functions are like factories that take in some material, do some operation, and then send out the resulting object.



In this case, we refer to the materials as arguments or inputs and the resulting object is refered to as output or return value. This process of taking input, doing something, and then sending back the output is demonstrated in the gif below.



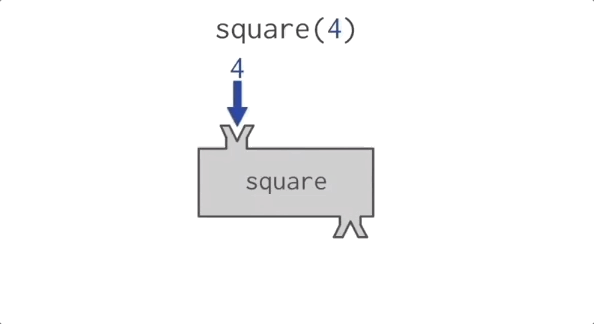
**Note**

Don’t confuse the “output value” of a function with the output window. The output of a function is a Python value and we can never really see the internal representation of a value. But we can draw pictures to help us imagine what values are, or we can print them to see an external representation in the output window.

To confuse things even more, print is actually a function. All functions produce output values. Only the print function causes things to appear in the output window.

It is also possible for programmers to define new functions in their programs. You will learn how to do that later in the course. For now, you just need to learn how to invoke, or call, a function, and understand that the execution of the function returns a computed value.

We’ve defined two functions above. The code is hidden so as not to bother you (yet) with how functions are defined. square takes a single input parameter, and returns that input multiplied by itself. sub takes two input parameters and returns the result of subtracting the second from the first. Obviously, these functions are not particularly useful, since we have the operators \* and - available. But they illustrate how functions work. The visual below illustrates how the square function works.



print(square(3))

square(5)

print(sub(6, 4))

print(sub(5, 9))

​

(ac2\_4\_2)

Notice that when a function takes more than one input parameter, the inputs are separated by a comma. Also notice that the order of the inputs matters. The value before the comma is treated as the first input, the value after it as the second input.

Again, remember that when Python performs computations, the results are only shown in the output window if there’s a print statement that says to do that. In the activcode window above, square(5) produces the value 25 but we never get to see that in the output window, because it is not printed.

## **2.4.1. Function calls as part of complex expressions**

Anywhere in an expression that you can write a literal like a number, you can also write a function invocation that produces a number.

1

print(square(3) + 2)

print(sub(square(3), square(1+1)))

Notice that we always have to resolve the expression inside the innermost parentheses first, in order to determine what input to provide when calling the functions.

print(sub(square(3), square(1+1)))

## **2.4.2. Functions are objects; parentheses invoke functions**

Remember the earlier mention that some kinds of Python objects don’t have a nice printed representation? Functions are themselves just objects. If you tell Python to print the function object, rather than printing the results of invoking the function object, you’ll get one of those not-so-nice printed representations.

Just typing the name of the function refers to the function as an object. Typing the name of the function followed by parentheses () invokes the function.

print(square)

print(square(3))

​

2.5. Data Types

If you are not sure what class (data type) a value falls into, Python has a function called **type** which can tell you.

print(type("Hello, World!"))

print(type(17))

print("Hello, World")

print(type(3.2))

​

(ac2\_5\_1)

What about values like "17" and "3.2"? They look like numbers, but they are in quotation marks like strings.

print(type("17"))

print(type("3.2"))

​

They’re strings!

Strings in Python can be enclosed in either single quotes (') or double quotes ("), or three of each (''' or """)

print(type('This is a string.'))

print(type("And so is this."))

print(type("""and this."""))

print(type('''and even this...'''))

​

Double quoted strings can contain single quotes inside them, as in "Bruce's beard", and single quoted strings can have double quotes inside them, as in 'The knights who say "Ni!"'. Strings enclosed with three occurrences of either quote symbol are called triple quoted strings. They can contain either single or double quotes:

print('''"Oh no", she exclaimed, "Ben's bike is broken!"''')

​

Triple quoted strings can even span multiple lines:

print("""This message will span

several lines

of the text.""")

​

Python doesn’t care whether you use single or double quotes or the three-of-a-kind quotes to surround your strings. Once it has parsed the text of your program or command, the way it stores the value is identical in all cases, and the surrounding quotes are not part of the value.

print('This is a string.')

print("""And so is this.""")

​

So the Python language designers usually chose to surround their strings by single quotes. What do you think would happen if the string already contained single quotes?

When you type a large integer, you might be tempted to use commas between groups of three digits, as in 42,000. This is not a legal integer in Python, but it does mean something else, which is legal:

print(42500)

print(42,500)

​

Well, that’s not what we expected at all! Because of the comma, Python chose to treat this as a *pair* of values. In fact, a print statement can print any number of values as long as you separate them by commas. Notice that the values are separated by spaces when they are displayed.

1

print(42, 17, 56, 34, 11, 4.35, 32)

print(3.4, "hello", 45)

​

Remember not to put commas or spaces in your integers, no matter how big they are. Also revisit what we said in the previous chapter: formal languages are strict, the notation is concise, and even the smallest change might mean something quite different from what you intended.

**Note**

The examples in this online text describe how print works in Python 3. If you install Python 2.7 on your machine, it will work slightly differently. One difference is that print is not called as a function, so there are no parentheses around the values to be printed.

**Check your understanding**

data-5-1: How can you determine the type of a variable?

Top of Form

A. Print out the value and determine the data type based on the value printed.  
B. Use the type function.  
C. Use it in a known equation and print the result.  
D. Look at the declaration of the variable.

Bottom of Form

data-5-2: What is the data type of ‘this is what kind of data’?

Top of Form

A. Character  
B. Integer  
C. Float  
D. String

2.6. Type conversion functions

Sometimes it is necessary to convert values from one type to another. Python provides a few simple functions that will allow us to do that. The functions int, float and str will (attempt to) convert their arguments into types int, float and str respectively. We call these **type conversion** functions.

The int function can take a floating point number or a string, and turn it into an int. For floating point numbers, it *discards* the decimal portion of the number - a process we call *truncation towards zero* on the number line. Let us see this in action:

print(3.14, int(3.14))

print(3.9999, int(3.9999)) # This doesn't round to the closest int!

print(3.0, int(3.0))

print(-3.999, int(-3.999)) # Note that the result is closer to zero

​

print("2345", int("2345")) # parse a string to produce an int

print(17, int(17)) # int even works on integers

print(int("23bottles"))

​

The last case shows that a string has to be a syntactically legal number, otherwise you’ll get one of those pesky runtime errors. Modify the example by deleting the bottles and rerun the program. You should see the integer 23.

The type converter float can turn an integer, a float, or a syntactically legal string into a float.

print(float("123.45"))

print(type(float("123.45")))

​

The type converter str turns its argument into a string. Remember that when we print a string, the quotes are removed in the output window. However, if we print the type, we can see that it is definitely str.

print(str(17))

print(str(123.45))

print(type(str(123.45)))

​

Because we can only print one type at a time, if we want to print out a string and an integer then we have to convert the type. Think about which type you’d need to convert, a string to an integer or an integer to a string?

**Check your understanding**

data-6-1: What value is printed when the following statement executes?

**print**(int(53.785))

Top of Form

A. Nothing is printed. It generates a runtime error.  
B. 53  
C. 54  
D. 53.785

2.7. Variables

One of the most powerful features of a programming language is the ability to manipulate **variables**. A variable is a name that refers to a value.

**Assignment statements** create new variables and also give them values to refer to.

message = "What's up, Doc?"

n = 17

pi = 3.14159

This example makes three assignments. The first assigns the string value "What's up, Doc?" to a new variable named message. The second assigns the integer 17 to n, and the third assigns the floating-point number 3.14159 to a variable called pi.

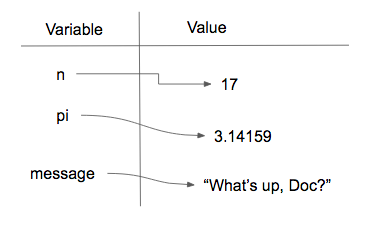
The **assignment token**, =, should not be confused with *equality* (we will see later that equality uses the == token). The assignment statement links a *name*, on the left hand side of the operator, with a *value*, on the right hand side. This is why you will get an error if you enter:

17 = n

**Tip:**

When reading or writing code, say to yourself “n is assigned 17” or “n gets the value 17” or “n is a reference to the object 17” or “n refers to the object 17”. **Don’t say “n equals 17”**.

A common way to represent variables on paper is to write the name with an arrow pointing to the variable’s value. This kind of figure, known as a **reference diagram**, is often called a **state snapshot** because it shows what state each of the variables is in at a particular instant in time. (Think of it as the variable’s state of mind). This diagram shows the result of executing the assignment statements shown above.



If your program includes a variable in any expression, whenever that expression is executed it will produce the value that is linked to the variable at the time of execution. In other words, evaluating a variable looks up its value.

message = "What's up, Doc?"

n = 17

pi = 3.14159

​

print(message)

print(n)

print(pi)

​

Now, as you step through the statements, you can see the variables and the values they reference as those references are created.

We use variables in a program to “remember” things, like the current score at the football game. But variables are *variable*. This means they can change over time, just like the scoreboard at a football game. You can assign a value to a variable, and later assign a different value to the same variable.

**Note**

This is different from math. In algebra, if you give x the value 3, it cannot change to refer to a different value half-way through your calculations!

To see this, read and then run the following program. You’ll notice we change the value of day three times, and on the third assignment we even give it a value that is of a different type.

A great deal of programming is about having the computer remember things. For example, we might want to keep track of the number of missed calls on your phone. Each time another call is missed, we will arrange to update or change the variable so that it will always reflect the correct value.

Any place in a Python program where a number or string is expected, you can put a variable name instead. The python interpreter will substitute the value for the variable name.

For example, we can find out the data type of the current value of a variable by putting the variable name inside the parentheses following the function name type.

message = "What's up, Doc?"

n = 17

pi = 3.14159

​

print(type(message))

print(type(n))

print(type(pi))

​

**Note**

If you have programmed in another language such as Java or C++, you may be used to the idea that *variables* have types that are declared when the variable name is first introduced in a program. Python doesn’t do that. Variables don’t have types in Python; *values* do. That means that it is acceptable in Python to have a variable name refer to an integer and later have the same variable name refer to a string. This is almost never a good idea, because it will confuse human readers (including you), but the Python interpreter will not complain.

**Check your understanding**

data-7-1: What is printed when the following statements execute?

day = "Thursday"

day = 32.5

day = 19

**print**(day)

Top of Form

A. Nothing is printed. A runtime error occurs.  
B. Thursday  
C. 32.5  
D. 19

2.8. Variable Names and Keywords

**Variable names** can be arbitrarily long. They can contain both letters and digits, but they have to begin with a letter or an underscore. Although it is legal to use uppercase letters, by convention we don’t. If you do, remember that case matters. Bruce and bruce are different variables.

**Caution:**

Variable names can never contain spaces.

The underscore character ( \_) can also appear in a name. It is often used in names with multiple words, such as my\_name or price\_of\_tea\_in\_china. There are some situations in which names beginning with an underscore have special meaning, so a safe rule for beginners is to start all names with a letter.

If you give a variable an illegal name, you get a syntax error. In the example below, each of the variable names is illegal.

76trombones = "big parade"

more$ = 1000000

class = "Computer Science 101"

76trombones is illegal because it does not begin with a letter. more$ is illegal because it contains an illegal character, the dollar sign. But what’s wrong with class?

It turns out that class is one of the Python **keywords**. Keywords define the language’s syntax rules and structure, and they cannot be used as variable names. Python has thirty-something keywords (and every now and again improvements to Python introduce or eliminate one or two):

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| and | as | assert | break | class | continue |
| def | del | elif | else | except | exec |
| finally | for | from | global | if | import |
| in | is | lambda | nonlocal | not | or |
| pass | raise | return | try | while | with |
| yield | True | False | None |  |  |

You might want to keep this list handy. If the interpreter complains about one of your variable names and you don’t know why, see if it is on this list.

**Check your understanding**

data-8-1: True or False: the following is a legal variable name in Python: A\_good\_grade\_is\_A+

Top of Form

A. True  
B. False

# 2.9. 👩‍💻 Choosing the Right Variable Name

Programmers generally choose names for their variables that are meaningful to the human readers of the program — they help the programmer document, or remember, what the variable is used for. Beginning programmers sometimes think it is funny to use strange or obscene names for their variables. This is not good practice and will not amuse your professor. Get in the habit of using meaningful names right away.

**Caution**

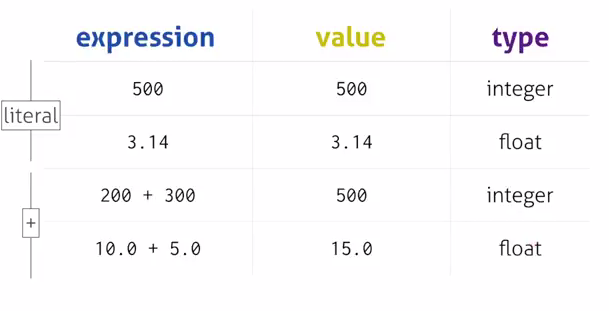
Beginners sometimes confuse “meaningful to the human readers” with “meaningful to the computer”. So they’ll wrongly think that because they’ve called some variable average or pi, it will somehow automagically calculate an average, or automagically associate the variable pi with the value 3.14159. No! The computer doesn’t attach semantic meaning to your variable names.

So you’ll find some instructors who deliberately don’t choose meaningful names when they teach beginners — not because they don’t think it is a good habit, but because they’re trying to reinforce the message that you, the programmer, have to write some program code to calculate the average, or you must write an assignment statement to give a variable the value you want it to have.

2.10. Statements and Expressions

A **statement** is an instruction that the Python interpreter can execute. You have only seen the assignment statement so far. Some other kinds of statements that you’ll see in future chapters are while statements, for statements, if statements, and import statements. (There are other kinds too!)

An **expression** is a combination of literals, variable names, operators, and calls to functions. Expressions need to be evaluated. The result of evaluating an expression is a *value* or *object*.



If you ask Python to print an expression, the interpreter **evaluates** the expression and displays the result.

1

print(1 + 1 + (2 \* 3))

print(len("hello"))

​

In this example len is a built-in Python function that returns the number of characters in a string.

The *evaluation of an expression* produces a value, which is why expressions can appear on the right hand side of assignment statements. A literal all by itself is a simple expression, and so is a variable.

y = 3.14

x = len("hello")

print(x)

print(y)

​

# 2.11. Order of Operations

When more than one operator appears in an expression, the order of evaluation depends on the **rules of precedence**. Python follows the same precedence rules for its mathematical operators that mathematics does.

1. Parentheses have the highest precedence and can be used to force an expression to evaluate in the order you want. Since expressions in parentheses are evaluated first, 2 \* (3-1) is 4, and (1+1)\*\*(5-2) is 8. You can also use parentheses to make an expression easier to read, as in (minute \* 100) / 60: in this case, the parentheses don’t change the result, but they reinforce that the expression in parentheses will be evaluated first.
2. Exponentiation has the next highest precedence, so 2\*\*1+1 is 3 and not 4, and 3\*1\*\*3 is 3 and not 27. Can you explain why?
3. Multiplication and both division operators have the same precedence, which is higher than addition and subtraction, which also have the same precedence. So 2\*3-1 yields 5 rather than 4, and 5-2\*2 is 1, not 6.
4. Operators with the same precedence are evaluated from left-to-right. In algebra we say they are left-associative. So in the expression 6-3+2, the subtraction happens first, yielding 3. We then add 2 to get the result 5. If the operations had been evaluated from right to left, the result would have been 6-(3+2), which is 1.

**Note**

Due to some historical quirk, an exception to the left-to-right left-associative rule is the exponentiation operator \*\*. A useful hint is to always use parentheses to force exactly the order you want when exponentiation is involved:

print(2 \*\* 3 \*\* 2) # the right-most \*\* operator gets done first!

print((2 \*\* 3) \*\* 2) # use parentheses to force the order you want!

​

**Note**

This is a second way that parentheses are used in Python. The first way you’ve already seen is that () indicates a function call, with the inputs going inside the parentheses. How can Python tell when parentheses specify to call a function, and when they are just forcing the order of operations for ambiguous operator expressions?

The answer is that if there’s a an expression to the left of the parentheses that evaluates to a function object, then the parentheses indicate a function call, and otherwise not. You will have to get used to making the same inference when you see parentheses: is this a function call, or just specifying precedence?

**Check your understanding**

data-11-1: What is the value of the following expression:

16 - 2 \* 5 // 3 + 1

Top of Form

A. 14  
B. 24  
C. 3  
D. 13.667

2.12. Reassignment

As we have mentioned previously, it is legal to make more than one assignment to the same variable. A new assignment makes an existing variable refer to a new value (and stop referring to the old value).

bruce = 5

print(bruce)

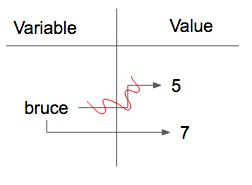
bruce = 7

print(bruce)

​

The first time bruce is printed, its value is 5, and the second time, its value is 7. The assignment statement changes the value (the object) that bruce refers to.

Here is what **reassignment** looks like in a reference diagram:



It is important to note that in mathematics, a statement of equality is always true. If a is equal to b now, then a will always equal to b. In Python, an assignment statement can make two variables refer to the same object and therefore have the same value. They appear to be equal. However, because of the possibility of reassignment, they don’t have to stay that way:

a = 5

b = a # after executing this line, a and b are now equal

print(a,b)

a = 3 # after executing this line, a and b are no longer equal

print(a,b)

​

Line 4 changes the value of a but does not change the value of b, so they are no longer equal.

2.13. Input

Our programs get more interesting if they don’t do exactly the same thing every time they run. One way to make them more interesting is to get **input** from the user. Luckily, in Python there is a built-in function to accomplish this task. It is called input.

n = input("Please enter your name: ")

The input function allows the programmer to provide a **prompt string**. In the example above, it is “Please enter your name: “. When the function is evaluated, the prompt is shown (in the browser, look for a popup window). The user of the program can type some text and press return. When this happens the text that has been entered is returned from the input function, and in this case assigned to the variable n. Run this example a few times and try some different names in the input box that appears.

n = input("Please enter your name: ")

print("Hello", n)

​

It is very important to note that the input function returns a string value. Even if you asked the user to enter their age, you would get back a string like "17". It would be your job, as the programmer, to convert that string into an int or a float, using the int or float converter functions we saw earlier.

**Note**

We often use the word “input” (or, synonymously, argument) to refer to the values that are passed to any function. Do not confuse that with the input function, which asks the user of a program to type in a value. Like any function, input itself takes an input argument and produces an output. The input is a character string that is displayed as a prompt to the user. The output is whatever character string the user types.

This is analogous to the potential confusion of function “outputs” with the contents of the output window. Every function produces an output, which is a Python value. Only the print function puts things in the output window. Most functions take inputs, which are Python values. Only the input function invites users to type something.

Here is a program that turns a number of seconds into more human readable counts of hours, minutes, and seconds. A call to input() allows the user to enter the number of seconds. Then we convert that string to an integer. From there we use the division and modulus operators to compute the results.

str\_seconds = input("Please enter the number of seconds you wish to convert")

total\_secs = int(str\_seconds)

​

hours = total\_secs // 3600

secs\_still\_remaining = total\_secs % 3600

minutes = secs\_still\_remaining // 60

secs\_finally\_remaining = secs\_still\_remaining % 60

​

print("Hrs=", hours, "mins=", minutes, "secs=", secs\_finally\_remaining)

​

The variable str\_seconds will refer to the string that is entered by the user. As we said above, even though this string may be 7684, it is still a string and not a number. To convert it to an integer, we use the int function. The result is referred to by total\_secs. Now, each time you run the program, you can enter a new value for the number of seconds to be converted.

**Check your understanding**

data-16-1: What is printed when the following statements execute?

n = input("Please enter your age: ")

*# user types in 18*

**print**(type(n))

Top of Form

A. <class 'str'>  
B. <class 'int'>  
C. <class 18>  
D. 18

# 2.14. Glossary

**assignment statement**

A statement that assigns a value to a name (variable). To the left of the assignment operator, =, is a name. To the right of the assignment token is an expression which is evaluated by the Python interpreter and then assigned to the name. The difference between the left and right hand sides of the assignment statement is often confusing to new programmers. In the following assignment:

n = n + 1

n plays a very different role on each side of the =. On the right it is a value and makes up part of the expression which will be evaluated by the Python interpreter before assigning it to the name on the left.

**assignment token**

= is Python’s assignment token, which should not be confused with the mathematical comparison operator using the same symbol.

**boolean expression**

An expression that is either true or false.

**boolean value**

There are exactly two boolean values: True and False. Boolean values result when a boolean expression is evaluated by the Python interepreter. They have type bool.

**class**

see **data type** below

**comment**

Information in a program that is meant for other programmers (or anyone reading the source code) and has no effect on the execution of the program.

**data type**

A set of values. The type of a value determines how it can be used in expressions. So far, the types you have seen are integers (int), floating-point numbers (float), and strings (str).

**decrement**

Decrease by 1.

**evaluate**

To simplify an expression by performing the operations in order to yield a single value.

**expression**

A combination of operators and operands (variables and values) that represents a single result value. Expressions are evaluated to give that result.

**float**

A Python data type which stores floating-point numbers. Floating-point numbers are stored internally in two parts: a base and an exponent. When printed in the standard format, they look like decimal numbers. Beware of rounding errors when you use floats, and remember that they are only approximate values.

**increment**

Both as a noun and as a verb, increment means to increase by 1.

**initialization (of a variable)**

To initialize a variable is to give it an initial value. Since in Python variables don’t exist until they are assigned values, they are initialized when they are created. In other programming languages this is not the case, and variables can be created without being initialized, in which case they have either default or garbage values.

**int**

A Python data type that holds positive and negative **whole** numbers.

**integer division**

An operation that divides one integer by another and yields an integer. Integer division yields only the whole number of times that the numerator is divisible by the denominator and discards any remainder.

**keyword**

A reserved word that is used by the compiler to parse program; you cannot use keywords like if, def, and while as variable names.

**literal**

A number or string that is written directly in a program. Sometimes these are also referred to as literal values, or just values, but be careful not to confuse a literal value as written in a program with an internal value maintained by the Python interpreter during execution of a program.

**logical operator**

One of the operators that combines boolean expressions: and, or, and not.

**modulus operator**

An operator, denoted with a percent sign ( %), that works on integers and yields the remainder when one number is divided by another.

**object**

Also known as a data object (or data value). The fundamental things that programs are designed to manipulate (or that programmers ask to do things for them).

**operand**

One of the values on which an operator operates.

**operator**

A special symbol that represents a simple computation like addition, multiplication, or string concatenation.

**prompt string**

Used during interactive input to provide the use with hints as to what type of value to enter.

**reference diagram**

A picture showing a variable with an arrow pointing to the value (object) that the variable refers to. See also **state snapshot**.

**rules of precedence**

The set of rules governing the order in which expressions involving multiple operators and operands are evaluated.

**state snapshot**

A graphical representation of a set of variables and the values to which they refer, taken at a particular instant during the program’s execution.

**statement**

An instruction that the Python interpreter can execute. So far we have only seen the assignment statement, but we will soon meet the import statement and the for statement.

**str**

A Python data type that holds a string of characters.

**type conversion function**

A function that can convert a data value from one type to another.

**value**

A number or string (or other things to be named later) that can be stored in a variable or computed in an expression.

**variable**

A name that refers to a value.

**variable name**

A name given to a variable. Variable names in Python consist of a sequence of letters (a..z, A..Z, and \_) and digits (0..9) that begins with a letter. In best programming practice, variable names should be chosen so that they describe their use in the program, making the program self documenting.

# 2.15. Exercises

1. Evaluate the following numerical expressions in your head, then use the active code window to check your results:

1. 5 \*\* 2
2. 9 \* 5
3. 15 / 12
4. 12 / 15
5. 15 // 12
6. 12 // 15
7. 5 % 2
8. 9 % 5
9. 15 % 12
10. 12 % 15
11. 6 % 6
12. 0 % 7

2. Write a program that will compute the area of a rectangle. Prompt the user to enter the width and height of the rectangle. Print a nice message with the answer.

3. Write a program that will convert degrees celsius to degrees fahrenheit.

4. Write a program that will convert degrees fahrenheit to degrees celsius.

5. Write a program that will compute the area of a circle. Prompt the user to enter the radius and print a nice message back to the user with the answer.2.2.Bottom of Form

Bottom of Form

Bottom of Form

Bottom of Form

Bottom of Form

Bottom of Form

Bottom of Form