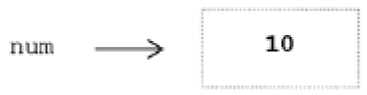
# Variables and Identifiers

So far, you have only looked at literal values in programs. However, the true usefulness of a computer program is the ability to operate on different values each time the program is executed. This is provided by the notion of a **variable**.

## What Is a Variable?

A **variable** is a name (identifier) that is associated with a value, as depicted below:



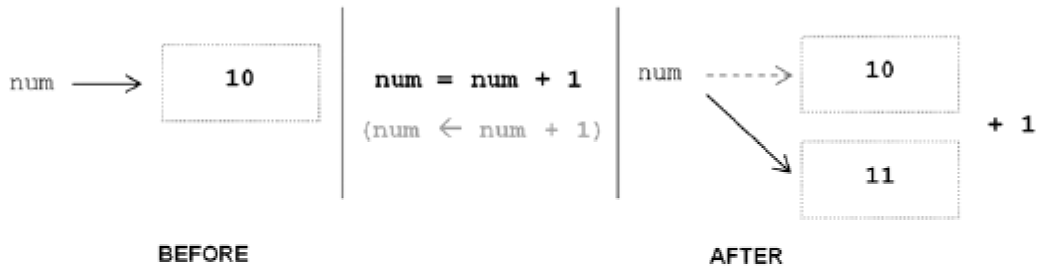
A variable can be assigned different values during a program's execution—hence, the name "variable." Wherever a variable appears in a program (except on the left-hand side of an assignment statement), *it is the value associated with the variable that is used*, and not the variable's name,

```
num 1 1 → 10 1 1 → 11
```

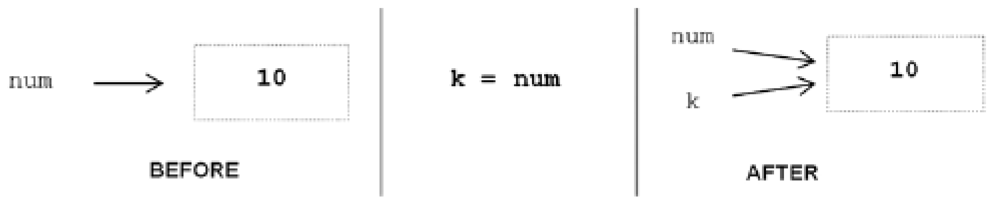
Variables are assigned values by use of the **assignment operator**, =,

```
num = 10   num = num + 1
```

Assignment statements often look wrong to novice programmers. Mathematically, `num = num + 1` does not make sense. In computing, however, it is used to increment the value of a given variable by one. It is more appropriate, therefore, to think of the = symbol as an arrow symbol, as shown below:



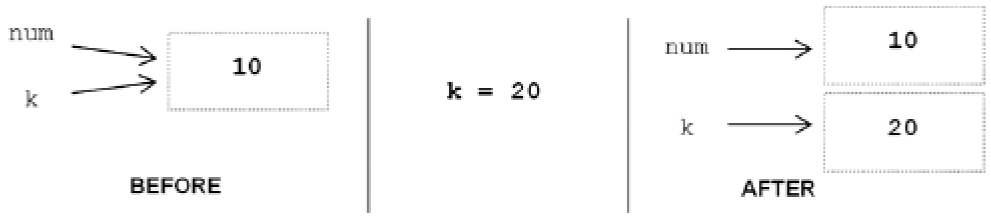
When thought of this way, it makes clear that *the right side of an assignment is evaluated first, then the result is assigned to the variable on the left*. An arrow symbol is not used simply because there is no such character on a standard computer keyboard. Variables may also be assigned to the value of another variable (or expression, discussed below) as shown below:



Variables num and k are both associated with the same literal value 10 in memory. One way to see this is by use of built-in function id,

```
>>> id(num)          >>> id(k)
505494040            505494040
```

The id function produces a unique number identifying a specific value (object) in memory. Since variables are meant to be distinct, it would appear that this sharing of values would cause problems. Specifically, if the value of num changed, would variable k change along with it? This cannot happen in this case because the variables refer to integer values, and integer values are *immutable*. An **immutable value** is a value that cannot be changed. Thus, both will continue to refer to the same value until one (or both) of them is reassigned, as seen below:



If no other variable references the memory location of the original value, the memory location is **deallocated** (that is, it is made available for reuse).

Finally, in Python the same variable can be associated with values of different type during program execution, as shown below.

```
var = 12             integer
var = 12.45          float
var = 'Hello'        string
```

**Your Turn**

**From the Python Shell, enter the following and observe the results.**

```
>>>num = 10                 >>> k = 25
>>> num                     >>> k
???                         ???
>>>id(num)                  >>> num
???                         ???
                            >>> id(k)
                            ???
                            >>> id(num)
                            ???

>>>num = 50
>>>num
???
>>> id(num)
???

>>>k = num                  >>> k = k + 1
>>> k                       >>> k
???                         ???
>>>id(k)                    >>> id(num)
???                         ???
>>> id(num)                 >>> id(k)
???                         ???
```

## Part II - Variable Assignment and Keyboard Input

The value that is assigned to a given variable does not have to be specified in the program, as demonstrated in previous examples. The value can come from the user by use of the input function introduced in Chapter 1,

```
>>> name = input('What is your first name?')
What is your first name? John
```

In this case, the variable `name` is assigned the string `'John'`. If the user hit return without entering any value, `name` would be assigned to the empty string (`''`).

All input is returned by the `input` function as a string type. *For the input of numeric values, the response must be converted to the appropriate type.* Python provides built-in **type conversion functions `int`**`()` and **`float`**`()` for this purpose, as shown below for a gpa calculation program,

```
line = input('How many credits do you have?')
num_credits = int(line)
line = input('What is your grade point average?')
gpa = float(line)
```

Here, the entered number of credits, say `'24'`, is converted to the equivalent integer value, `24`, before being assigned to variable `num_credits`. For input of the gpa, the entered value,

say `'3.2'`, is converted to the equivalent floating-point value, `3.2`. Note that the program lines above could be combined as follows,

```
num_credits 5 int(input('How many credits do you have? '))
gpa 5 float(input('What is your grade point average? '))
```

**Your Turn**

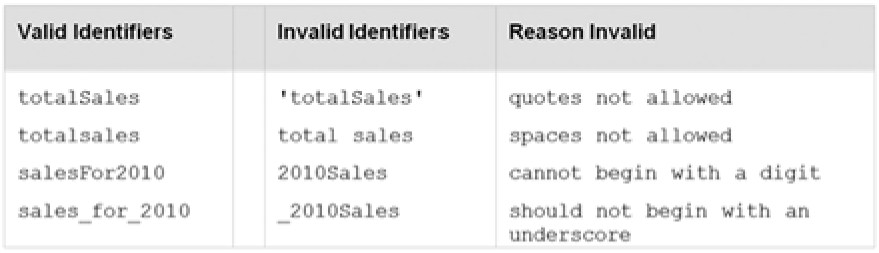
From the Python Shell, enter the following and observe the results.

```
>>> num = input('Enter number: ')   >>> num = input('Enter name: ')

Enter number: 5                     Enter name: John

???                                 ???

>>> num = int(input('Enter number: '))  >>> num = int(input('Enter name: '))

Enter number: 5                     Enter name: John

???                                 ???
```

## Part III - Identifiers

An **identifier** is a sequence of one or more characters used to provide a name for a given program element. Variable names `line, num_credits,` and `gpa` are each identifiers. Python is *case sensitive*, thus, `Line` is different from `line`. Identifiers may contain letters and digits, but cannot begin with a digit. The underscore character, _, is also allowed to aid in the readability of long identifier names. It should not be used as the *first* character, however, as identifiers beginning with an underscore have special meaning in Python.

Spaces are not allowed as part of an identifier. This is a common error since some operating systems allow spaces within file names. In programming languages, however, spaces are used to delimit (separate) distinct syntactic entities. Thus, any identifier containing a space character would be considered two separate identifiers. Examples of valid and invalid identifiers in Python are shown below:



| Valid Identifiers | | Invalid Identifiers | Reason Invalid |
|---|---|---|---|
| totalSales | | 'totalSales' | quotes not allowed |
| totalsales | | total sales | spaces not allowed |
| salesFor2010 | | 2010Sales | cannot begin with a digit |
| sales_for_2010 | | _2010Sales | should not begin with an underscore |

**Your Turn**

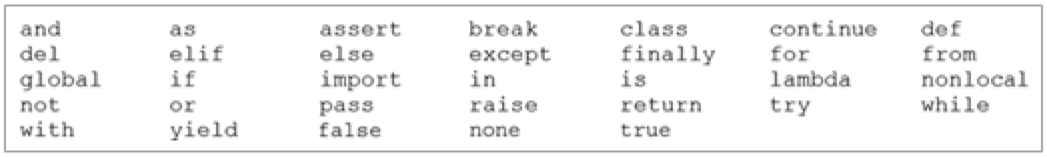
From the Python Shell, enter the following and observe the results.

```
>>>spring2014SemCredits = 15          >>> spring2014-sem-credits = 15
???                                   ???

>>>spring2014_sem_credits = 15        >>> 2014SpringSemesterCredits = 15
???                                   ???
```

## Keywords and Other Predefined Identifiers in Python

A **keyword** is an identifier that has predefined meaning in a programming language. Therefore, keywords cannot be used as “regular” identifiers. Doing so will result in a syntax error, as demonstrated in the attempted assignment to keyword `and` below,

```
>>> and 5 10
SyntaxError: invalid syntax
```



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

The keywords in Python are listed above. To display the keywords, type `help()` in the Python shell, and then type `keywords` (type `'q'` to quit).

There are other predefined identifiers that *can* be used as regular identifiers, but should not be. This includes `float, int, print, exit,` and `quit,` for example. A simple way to check whether a given identifier is a keyword in Python is given below,

```
>>> 'exit' in dir( builtins )
True

>>> 'exit_program' in dir( builtins )
False
```

## Your Turn

From the Python Shell, enter the following and observe the results.

```
>>>yield = 1000            >>> print('Hello')

???                           ???

>>>Yield = 1000                >>> print = 10
???                            >>> print (‘Hello’)
                               ???
```

## Concepts

**1.** Which of the following are valid assignment statements, in which only variable *k* has already been assigned a value?

**(a)** `n = k + 1` **(b)** `n = n + 1` **(c)** `n + k = 10` **(d)** `n + 1 = 1`

**2.** What is the value of variable `num` after the following assignment statements are executed?

```
num = 0
num = num + 1
num = num + 5
```

**3.** Do variables `num` and `k` reference the same memory location after the following instructions are executed? (YES / NO)

```
num = 10
k = num
num = num + 1
```

**4.** Which of the following are valid identifiers in Python?

**(a)** `errors` **(b)** `error_count` **(c)** `error-count`

**5.** Which of the following are keywords in Python?

**(a)** `and` **(b)** `As` **(c)** `while` **(d)** `until` **(e)** `NOT`

**6.** Which one of the following is correct for reading and storing an integer value from the user?

**(a)** `n = int_input('Enter: ')` **(b)** `n = int(input('Enter: '))`

## Problem Solving

**1.** Regarding variable assignment,

**(a)** What is the value of variables `num1` and `num2` after the following instructions are executed?

```
num = 0
k = 5
num1 = num + k * 2
num2 = num + k * 2
```

**(b)** Are the values `id(num1)` and `id(num2)` equal after the last statement is executed?

**2.** Regarding the `input` function in Python,

**(a)** Give an instruction that prompts the user for their last name and stores it in a variable named `last_ name`.

**(b)** Give an instruction that prompts the user for their age and stores it as an integer value named `age`.

**(c)** Give an instruction that prompts the user for their temperature and stores it as a float named `current_temperature`.

**3.** Regarding keywords and other predefined identifiers in Python, give the result for each of the following,

**(a)** `'int' in dir(  builtins )`

**(b)** `'import' in dir(  builtins )`