

Chapter 2

Algebra with Variables

Python Interpreter working as a basic calculator was explained in the opening chapter. Working in the calculator mode can be done with variables as well. Variables, their types, and different basic operations with them are discussed here.

2.1 Variables

One can define variables, assign values to them, and do algebra. Consider the sequence in Fig. 2.1. [1] has a variable with assigned name '**a**'. It has been assigned the integral value '**3**'. There is no need to assign a variable '**tag**' or assign a type to it. From the statement in [1] Python understands all these. In [2] we are putting a query to Python 'What is '**a**'?' Python interpreter returns the value assigned to '**a**'. In [3] we are passing on the query 'What type of object is this entity '**a**'?'. The interpreter returns with the clarification that '**a**' belongs to the class of objects termed '**int**' (integer). Such type queries can be made whenever desired to understand the class (identity) of any object.

In [4] a variable has been given the name **Aa** and it is assigned the integral value '**4**'. In general a variable can be given a name—called 'Identifier'—as a sequence of ASCII characters excluding the two—'\$' and '?'. The preferred practice is to use Identifiers for variables as well as other entities that we use in a language like Python such that the variable/entity can be easily identified from it. The constraints in the selection here are

- The first character has to be a small or a capital letter or '_' (underscore).
- The characters '\$' and '?' cannot be used in an Identifier.
- The Identifier should not begin or end with a pair of underscores. In fact these are reserved for specific use (described later).
- A specific set of combinations of letters is used as 'keywords' in Python (van Rossum and Drake 2014a). These are to be avoided as Identifiers. Table 2.1

>>> a = 3	[1]	>>> d1, e1 = 4.2, 5.3	[14]
>>> a	[2]	>>> d1, e1 = e1, d1	
3		>>> d1	
>>> type(a)	[3]	5.3	
<class 'int'>		>>> e1	
>>> Aa = 4	[4]	4.2	
>>> Aa*2		>>> d1, e1 = e1, d1-2	[15]
8		>>> d1	
>>> b = 4.1	[5]	4.2	
>>> type(b)		>>> e1	
<class 'float'>		3.3	
>>> c = a+b	[6]	>>> d3, e3 = d2*e2, d2/e2	[16]
>>> c		>>> d3	
7.1		22.8484	
>>> type(c)	[7]	>>> e3	
<class 'float'>		1.0	
>>> d = c*c-a**2-b**2	[8]	>>> d2 = e2 = 4.78	[17]
>>> d		>>> d2	
24.599999999999998		4.78	
>>> a1=4	[9]	>>> e2	
>>> _b = 5		4.78	
>>> c_ = (a1 - _b)**3		>>> d3+=1	[18]
>>> c_-1		>>> d3	
>>> type(c)		23.8484	
<class 'float'>		>>> e3-=2	[19]
>>> d, e = 5.1, 9	[10]	>>> e3	
>>> d		-1.0	
5.1		>>> d3/=2	[20]
>>> type(d)	[11]	>>> d3	
<class 'float'>		11.9242	
>>> e		>>> d3*=2	[21]
9		>>> d3	
>>> type(e)	[12]	23.8484	
<class 'int'>		>>> d3 *= e3	[22]
>>> d_e = d/e	[13]	>>> d3	
>>> d_e		-23.8484	
0.5666666666666667		>>>	
>>> type(d_e)			
<class 'float'>			

Fig. 2.1 A Python Interpreter sequence involving variables and assignments

is the set of all the keywords in Python. Avoiding their use directly or in combinations is healthy programming practice. Same holds good of ‘built-in’ function names such as abs, repr, chr, divmod, float, and so on.

b in [5], **c** in [6], and **d_e** in [13] are other examples of such Identifiers. Identifiers are case sensitive; **a** and **A** are different variables. [5] defines a variable **b** and assigns a value 4.1 to it. Python automatically takes b as a floating point variable and assigns the value 4.1 to it. The same is clarified by the ‘type (**b**)’ query and the clarification offered by Python in the two lines following. Algebra with variables can be carried out as with integers. In [6] the values of **a** and **b** are

Table 2.1 The set of keywords in Python

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

added and assigned to a new variable **c**. Once again there is no need for a separate declaration, type clarification, and so on. [7] and [8] in the following lines clarify this. ‘Chain’ algebra can be carried out and assigned to variables—if necessary new ones—as can be seen from [8] and the following lines. [9] and the following lines are further examples of this. [10] has two variables assigned values in a sequence. Such sequential assignments can be done for any number of variables. Python will decide the type of variable and assign values to them conforming to the sequence specified. The type queries [11] and [12] and the Python responses in the lines that follow clarify this. In [13] **d_e** is assigned the value (**d/e**). ‘**d**’ being a floating point variable—with value 5.1 as can be seen from [10]—**d_e** is automatically taken as a floating point variable and assigned the result. The query and response that follow confirm this. **d1** and **e1** are assigned values 4.2 and 5.3 in [14]. The following lines reassign values to them. Note that the assignments to **d1** and **e1** have been interchanged without the use of an intermediate temporary storage. This is not limited/restricted to numerical assignments alone. In [15] the new value of **e1** is **d1-2** with **d1** having the value prior to the present assignment. [16] is another example of multiple assignments done concurrently. The Python execution sequence following confirms this. In [17] **d2** and **e2** are assigned the same value of -4.78 . Such sequence of assignments is also possible. The combination operator ‘+=’ in [18] assigns a new value to **d3** as **d3** = **d3** + 1. Same holds good of the combination operators ‘-=’, ‘*=’, and ‘/=’ as can be seen from [19], [20], [21], and [22] and the query-response sequences following these.

Table 2.2 Operators in Python: Algebraic operators are listed in order of ascending priorities

Algebraic operators	Symbol	Operation performed	Logical/bit operators	Symbol	Operation performed
	+	Addition		~	Complement
	−	Subtraction		&	Logical AND
	*	Multiplication			Logical OR
	/	Division		^	Logical XOR
	//	Floored quotient		>>	Right shift (bits)
	%	Remainder		<<	Left shift (bits)
	**	Exponentiation			

In addition the combination operators—+=, -=, *=, /=, //=, %=, **=, &=, |=, ~=, >>=, and <<= are also available

>>> a=3.2		4.78539444560216
>>> b = _	[1]	>>> hc = complex(3.1, 4.2)
>>> b		[15]
3.2		>>> hc
>>> c = b*2		(3.1+4.2j)
>>> c		>>> complex(a,aa) [16]
6.4		(-3.2+3.2j)
>>> d = 9-_	[2]	>>> he = complex(a) [17]
>>> d		>>> he
2.5999999999999996		(-3.2+0j)
>>> e = 4+3j	[3]	>>> type(he) [18]
>>> f = e*2	[4]	<class 'complex'>
>>> f		>>> hk = hc.conjugate()
(8+6j)		[19]
>>> e**2	[5]	>>> hk
(7+24j)		(3.1-4.2j)
>>> g=e	[6]	>>> hr = hk.real [20]
(3+21j)		>>> hr
>>> h = 2.1-4.3j	[7]	3.1
>>> h*3	[8]	>>> hi = hk.imag [21]
(6.300000000000001-		>>> hi
12.899999999999999j)		-4.2
>>> i = h**2	[9]	>>> type(hi) [22]
>>> i		<class 'float'>
(-14.079999999999998-18.06j)		>>> a, b = 3, 4
>>> i**0.5	[10]	>>> pow(b,a) [23]
(2.0999999999999996-4.3j)		64
>>> ii = _*2	[11]	>>> pow(b,3.0) [24]
>>> ii		64.0
(4.199999999999999-8.6j)		>>> pow(b,-a) [25]
>>> (i**0.5)*2	[12]	0.015625
(4.199999999999999-8.6j)		>>> pow(b,a,5) [26]
>>> a = -3.2		4
>>> aa = abs(a)	[13]	>>> g = pow(13,11) [27]
>>> aa		>>> g
3.2		1792160394037
>>> ha = abs(h)	[14]	>>> g%17
>>> ha		4
		>>> pow(13,11,17) [28]
		4

Fig. 2.2 A Python Interpreter sequence with algebraic operations and simple functions with numbers

The operators used in algebra and the operations they signify are given in Table 2.2 (van Rossum and Drake 2014b). The combination operators are also given in the table.

The underscore symbol ‘_’ plays a useful role in interactive sessions. It is assigned the last ‘printed’ expression. Referring to the sequence in Fig. 2.2, in [1] it is the numerical value—that is 3.2—in the preceding line. **b** is assigned this value of 3.2. **b** carries this value for subsequent algebraic steps. [2] is an instance of using ‘_’ where it is used in an algebraic expression.

2.2 Complex Quantities

Python has the provision to handle complex numbers and variables. [3] assigns the value $4 + 3j$ to **e**. Here $3j$ signifies the imaginary component of the number. Algebra can be carried out with complex numbers with equal ease—in the same manner as real numbers. [4], [5], and [6] represent such algebra where the real imaginary parts of the variables/numbers and the expected results—are all integers. [7], [8], [9], and [10] show cases where the complex numbers involved have the real and imaginary parts in floating point form. The results too have the real and imaginary parts in floating point form. [11] is another example of the use of ‘_’—to use the last result in the current line without the need to retype. [12] confirms the correctness of the computation with [11].

2.3 Common Functions with Numbers

Python has a number of built-in functions (van Rossum and Drake 2014b); each function accepts the specified arguments, executes the routines concerned and returns the result (if and as desired). Functions are discussed in detail later. Here we introduce a few of the built-in functions useful directly in the calculator type of work. `abs(a)` returns the absolute value of a specified as argument. [13] is an instance of the absolute value of -3.2 returned as 3.2 . [14] returns the absolute value of the complex number **h** with assigned value in [7] as $2.1 - 4.3j$ that is 4.78539444560216 . `complex()` is another built-in function. It takes two arguments—**x** and **y**—in the same order and returns the complex quantity $\mathbf{x} + \mathbf{y}j$. [15] is an example of the direct use of `complex()` function to form the complex number $3.1 + 4.2j$ taking 3.1 and 4.2 as the arguments. [16] is another example confirming this. If only one argument is specified in the `complex()` function it is implicitly taken as the real component and the imaginary part is automatically taken as zero. [17] is an illustration of this usage as can be seen from the lines following. The conjugate of a complex quantity is obtained as in [19]—**he** representing the complex conjugate of **hc**. In [20] and [21] `hk.real` and `hk.imag` return the real and imaginary components of **hk** and assign them to the variables **hr** and **hi** respectively. The following line confirms that **hr** is a floating point number (the same is true of **hi** also). The function `pow(a, b)` returns $\mathbf{a}^{\mathbf{b}}$ —the same as `a ** b`. Here **a** and **b** can be integers or floating point numbers. $\mathbf{a}^{\mathbf{b}}$ is an integer if and only if **a** and **b** are integers and **b** is positive. These can be seen from [22] to [25]. The function `pow(a, b, c)` returns $(\mathbf{a} ** \mathbf{b}) \% \mathbf{c}$ as can be seen from [26]. [27] is another illustration of this at a slightly longer integer level. The sequence computes $(13^{11}) \% 17$ in two steps—a longer route. [28] achieves the same in a single step. Figure 2.3 shows the possibilities and constraints in the use of `pow()` in a compact form.

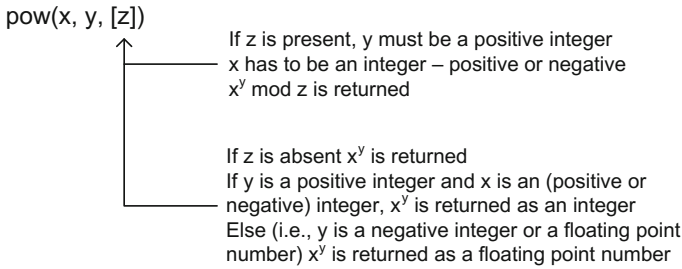


Fig. 2.3 Different possibilities and constraints of `pow()` function execution

When doing numerical work sometimes it becomes necessary to convert an integer into floating point mode. Similarly a floating point number may have to be approximated to an integer. The relevant functions and their use are illustrated through the Python Interpreted sequence in Fig. 2.4. **x** is a floating point number with value 4.3 and **y** an integer with value 3 as assigned in [1]. As can be seen from [2] $z = y^x$. **y**—an integer is raised to the power of a floating point number; the result seen from [3] is a floating point number. `int(x)` is assigned the value of the integral part of **x**—namely 4. Hence $z = y^{\text{int}(x)}$ is an (positive) integral power of an integer; the result is an integer as can be seen from [4]. The integer **y** is converted into floating point mode by [5] returning the assigned value 3.0; note that the numerical value remains unaltered. This floating point number is raised to the powers of **x** and `int(x)` respectively in [6]. Both the results—in [7] and [8] are in floating point mode. These may be compared with the corresponding values in [3] and [4] obtained earlier. The function $z = \text{int}(x)$ retains the integral part of **x** and ignores the fractional part. Even if the fractional part exceeds 0.5, it is ignored; the

```

>>> x, y = 4.3, 3                                [1]
>>> z1, z2 = y**x, y**int(x)                     [2]
>>> z1                                             [3]
112.62152279558863
>>> z2                                             [4]
81
>>> float(y)                                       [5]
3.0
>>> z3, z4 = float(y)**x, float(y)**int(x)         [6]
>>> z3                                             [7]
112.62152279558863
>>> z4                                             [8]
81.0
>>> int(4.7)                                       [9]
4
>>>
  
```

Fig. 2.4 Illustration of conversions between floating point numbers and integers

Table 2.3 Common functions with numbers

Function—form	Result
<code>abs(x)</code>	Returns the absolute value of x
<code>complex(a, b)</code>	Returns the complex number a + b j
<code>x.conjugate</code>	Returns the complex conjugate of x
<code>x.real</code>	Returns the real part of the complex number x
<code>x.imag</code>	Returns the imaginary part of the complex number x
<code>pow(x, y, z)</code>	Returns x ** y if z is absent; returns (x ** y) % z if z in present
<code>int(x)</code>	Returns the integral part of x
<code>float(x)</code>	Converts the integer x into a floating point number with the same value

operation done by `int()` is not a rounding off. [9] confirms this. Table 2.3 summarizes the functions with numbers discussed here. Additional functions are introduced later.

The basic operators in Python for doing algebra as well as for forming algebraic expressions are given in Table 2.2. They are listed in the table in the order of their priorities (specifically the operators in descending order of priorities are—`**`, `%`, `//`, `/`, `*`, `-`, and `+`). Thus in any algebraic chain `***`—if present—will be evaluated first; then `%` and so on. `+` operation is the last one to be carried out. The Python Interpreter sequence in Fig. 2.5 illustrates these. `3 * 4` in [1] is fairly clear; the integer 3 is multiplied by the positive integer +4 to yield the integer 12 as the result. A clearer way of specifying this is shown to the right (after the `#` symbol) as `3 * (+4)`. Similarly `12/-4` in [2] is interpreted as division of integer 12 by the negative integer -4 with -3.0 as the result. Once again `12/(-4)` shown at the right is clearer. With `12 + 5 * 8` in [3] the `*` operation gets priority over the `+` operation; hence `5 * 8` is done first and the result (40) added to 12 subsequently to yield 52 as the result. `12 + (5 * 8)` shown at the right avoids any ambiguity. Note that

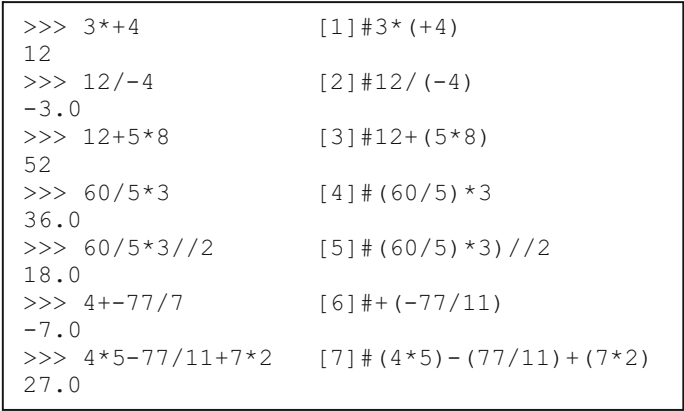


Fig. 2.5 Representative algebra involving multiple operations and their priorities

$(12 + 5) * 8$ is different from this. With $60/5 * 3$ the division operation ($/$) gets priority over the multiplication operation ($*$); hence the expression is evaluated as $(60/5) * 3$ ($=36.0$) as clarified at the right. Similarly $60/5 * 3//2$ in [5] is evaluated as $((60/5) * 3)//2$ to yield 18.0. With $4 * 5 - 77/11 + 7 * 2$ in [7] division ($77/11 = 7.0$), multiplications— $4 * 5$ (20) and $7 * 2$ ($=14$), subtraction ($20 - 7.0 = 13.0$), and addition— $13.0 + 14$ ($=27.0$) are carried out in that order. The expression evaluated is $(4 * 5) - (77/11) + (7 * 2)$ as shown in the right. Since the division $77/11$ always returns a floating point number the final result (algebra involving a mixture of integers and floating point numbers) yields a floating number.

Algebraic expressions can be made compact by conforming to the priorities of operators. However as a practice, it is better to use parentheses (though superfluous) and clarify the desired sequence and avoid room for ambiguity.

2.4 Logical Operators

The logical operators in Table 2.2 operate bit-wise on integers. The Python Interpreter sequence in Fig. 2.6 illustrates their use. **a** in [1] and **b** in [2] are 10110_2 (binary equivalent of the decimal number 22) and 10101_2 (binary equivalent of the decimal number 21) respectively. **a|b** in [3] is 10111_2 which is 23 in decimal form. The other operations too can be verified similarly.

Fig. 2.6 Illustration of use of logical operators

```

>>> a = 22                [1]
>>> b = 21                [2]
>>> a|b                   [3]
23
>>> a & b                 [4]
20
>>> a^b                   [5]
3
>>> (~a) & (~b)          [6]
-24
>>> a << 2               [7]
88
>>> b >> 2               [8]
5
>>>

```


2.5 Strings and Printing

A string is a type of ‘object’ in Python. Any character sequence can form a string. Strings can be useful in taking output as printouts and in presenting any entity in/from Python. Figure 2.7 is a Python interpreted sequence to demonstrate basic operations with strings (van Rossum and Drake 2014c). String ‘great’ is assigned to the identifier **s1**. [2] and the line following it confirm this. Similarly **s2** in [3] is a string. Strings can be combined conveniently using the addition operator—‘+’. **s3** defined in [4] is such a combination as can be seen from [5] and the output

```

>>> s1 = 'great' [1]
>>> type(s1) [2]
<class 'str'>
>>> s2 = 'day' [3]
>>> s3 = s1+s2 [4]
>>> s3 [5]
'greatday'
>>> s4 = s1 + ' ' + s2 [6]
>>> s4
'great day'
>>> s5 = s4*3 [7]
>>> s5
'great daygreat daygreat day'
>>> s5 = s4 + '! ' [8]
>>> s6 = s5*3
>>> s6
'great day! great day! great day! '
>>> print(s1) [9]
great
>>> a = 35 [10]
>>> print(a) [11]
35
>>> a, b, c = 10, 21.3, True [12]
>>> print(a, b, c) [13]
10 21.3 True
>>> b = 11
>>> c = a*b [14]
>>> repr(c) [15]
'385'
>>> type(repr(c)) [16]
<class 'str'>
>>> print('Product of a and b is' + '=' + repr(c)) [17]
Product of a and b is=385
>>> print('Product of a and b is' + '=' + repr(a*b)) [18]
Product of a and b is=385
>>> print('Product of a and b is' + ' = ' + repr(a*b)) [19]
Product of a and b is = 385

```

Fig. 2.7 Illustration of basic operations with strings

following. In Python the operator '+' is used in the sense of combining two entities but not restricted to mean the addition of two numbers alone. Such an extended concept is true of other operators as well as many functions as well. These will be explained duly.

s4 in [6] is a more elegant 'good day' than **s3**. Here the white space—' '—a string of single character has been interposed between 'good' and 'day'; the three strings—**s1**('good'), the white space string ' ', and **s2**('day') have been combined using the '+' operator to form the string **s4** ('good day'). The '*' operator can be used with a string to repeat a desired sequence (imposition!). **s5** in [7] is an example. It has been refined in [8] and the repeated sequence reproduced as **s6** in a more elegant manner as can be seen from [8].

Entities in Python (like strings) can be output/displayed by invoking the `print()` function. The `print(s1)` in [9] is possibly the simplest form of use of `print()` function. Any string can be directly output in this manner. Objects which can be output directly too can be printed in this manner. **a** has been assigned the integer value 35 in [10] (hence **a** is of Type `int`); it is output in [11] through `print(a)`. **a**, **b**, and **c** have been assigned values 10, 21.3 and `True` respectively in [12]; in turn they are of type `int`, `float`, and `Boolean`. They are output directly in the same sequence with the `print` command `print(a, b, c)` in [13]. Numbers, values of variables, and the like have to be converted into string form before they can be output through the `print` function. The function `repr()` achieves such a conversion into a string form which can be directly used as input to the `print()` function. **c** as specified in [14] forms the product of **a**(=35) and **b**(=11). To get it (value of 385) printed out it is converted into string form through `repr(c)` in [15]. [16] confirms this. Its value is printed out in [17]. **a * b** or any other such algebraic sequence—its value—can be directly converted into a string—avoiding the use of the intermediate temporary variable **c**—for output as done in [18] and [19].

2.6 Exercises

1. Evaluate the following:

- (a) $0.2 + 2.3 - 9.7 + 11.2$
- (b) $4.2 - 2.3 + 9.7 - 11.2$
- (c) $4.2 - 2.3 + 9.7 * 11.2$
- (d) $4.2 - 2.3 + 9.7/11.2$
- (e) $(4.2/2.3) * (9.7/11.2)$
- (f) $(4.2/2.3) ** (9.7/11.2)$
- (g) $97 \% 3, \text{pow}(97, 3, 23), \text{pow}(3, 97, 23)$

2. **AA** = 'Mary had a little lamb'. '**AA**' is a string here. Assign different names to each word in **AA** (**b1**, **b2**, ...), combine them in different combinations, and print out.

3. Evaluate the following:

- (h) $(((((3 ** 2) ** 2) ** 2) ** 0.5) ** 0.5) ** 0.5$: these types of expressions are evaluated to ascertain the accuracy/speed achieved/possible with numerical methods/computers.
- (i) $(((((0.3 ** 0.5) ** 0.5) ** 0.5) ** 0.5) ** 0.5) ** 0.5$
- (j) $(((((3 ** 0.5) ** 0.5) ** 0.5) ** 0.5) ** 0.5) ** 0.5$
- (k) $\frac{1}{1 + \frac{1}{1 + \frac{1}{1 + \frac{1}{4}}}}$
- (l) $\frac{1}{1 - \frac{1}{1 - \frac{1}{1 - \frac{1}{4}}}}$

References

van Rossum G, Drake FL Jr (2014a) The Python language reference. Python software foundation
 van Rossum G, Drake FL Jr (2014b) The Python library reference. Python software foundation
 van Rossum G, Drake FL Jr (2014c) Python tutorial. Python software foundation



<http://www.springer.com/978-981-10-3276-9>

Programming with Python

Padmanabhan, T.R.

2016, XIII, 344 p. 207 illus., Hardcover

ISBN: 978-981-10-3276-9