Python 2

Operators

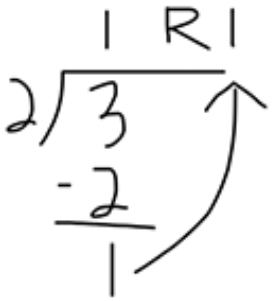
An operator in a programming language is a symbol that tells the compiler or interpreter to perform specific mathematical, relational or logical operation and produce a result. Here we explain the concept of operators.

Arithmetic Operators

In Python we can write statements that perform mathematical calculations. To do this we need to use operators that are specific for this purpose. Here are arithmetic operators:

Operator	Description	Example	Result
`+`	Addition	`3+2`	5
`-`	Subtraction	`3-2`	1
`*`	Multiplication	`3*2`	6
`/`	Division	`3/2`	1.5
`%`	Modulus (divides left operand by right operand and returns the remainder)	`3%2`	1
`**`	Exponent	`3**2`	9
`//`	Floor Division (result is the quotient with digits after the decimal point removed. If one of the operands is negative, the result is floored, i.e., rounded away from zero	`3//2` `-11//3`	1 -4

Modulus



Floor examples

```
>>> 3/2
1.5
>>> 3//2
1
>>> -11/3
-3.6666666666666665
>>> -11//3
-4
>>> 11/3
3.6666666666666665
>>> 11//3
3
```

Assignment Operators

We use assignment operators to assign values to variables. You have been using the `=` assignment operator. Here are others:

Operator	Equivalent to	Example	result evaluates to
`=`	`a = 3`	`result = 3`	3
`+=`	`result = result + 2`	`result = 3 ; result += 2`	5
`-=`	`result = result - 2`	`result = 3 ; result -= 2`	1
`*=`	`result = result * 2`	`result = 3 ; result *= 2`	6
`/=`	`result = result / 2`	`result = 3 ; result /= 2`	1.5
`%=`	`result = result % 2`	`result = 3 ; result %= 2`	1
`**=`	`result = result ** 2`	`result = 3 ; result **= 2`	9
`//=`	`result = result // 2`	`result = 3 ; result //= 3`	1

Comparison Operators

These operators compare two values and returns true or false.

Operator	Description	Example	Result
`==`	equal to	`3 == 2`	False
`!=`	not equal	`3 != 2`	True
`>`	greater than	`3 > 2`	True
`<`	less than	`3 < 2`	False
`>=`	greater than or equal	`3 >= 2`	True
`<=`	less than or equal	`3 <= 2`	False

Logical Operators

Logical operators allow you to combine two or more sets of comparisons. You can combine the results in different ways. For example you can 1) demand that all the statements are true, 2) that only one statement needs to be true, or 3) that the statement needs to be false.

Operator	Description	Example	Result
`and`	True if left operand is True and right operand is True	`3>=2 and 2<3`	True
`or`	True if left operand is True or right operand is True	`3==2 or 2<3`	True
`not`	Reverses the logical status	`not False`	True

Membership Operators

You can test to see if a value is included in a string, tuple, or list. You can also test that the value is not included in the string, tuple, or list.

Operator	Description
`in`	True if a value is included in a list, tuple, or string
`not in`	True if a value is absent in a list, tuple, or string

For Example:

```
>>> dna = 'GTACCTTGATTTCGTATTCTGAGAGGCTGCTGCTTAGCGGTAGCCCCTTGGTTTCCGTGGCAACGGAAAA'
>>> 'TCT' in dna
True
>>>
>>> 'ATG' in dna
False
>>> 'ATG' not in dna
True
>>> codons = [ 'atg' , 'aaa' , 'agg' ]
>>> 'atg' in codons
True
>>> 'ttt' in codons
False
```

Operator Precedence

Operators are listed in order of precedence. Highest listed first. Not all the operators listed here are mentioned above.

Operator	Description
`**`	Exponentiation (raise to the power)
`~` `+` `-`	Complement, unary plus and minus (method names for the last two are +@ and -@)
`*` `/` `%` `//`	Multiply, divide, modulo and floor division
`+` `-`	Addition and subtraction
`>>` `<<`	Right and left bitwise shift
`&`	Bitwise 'AND'
`^` `|`	Bitwise exclusive 'OR' and regular 'OR'
`<=` `<` `>` `>=`	Comparison operators
`<>` `==` `!=`	Equality operators
`=` `%=` `/=` `//=` `-=` `+=` `*=` `**=`	Assignment operators
`is`	Identity operator
`is not`	Non-identity operator
`in`	Membership operator
`not in`	Negative membership operator
`not` `or` `and`	logical operators

Note: Find out more about bitwise operators.

Truth

Lets take a step back... What is truth?

Everything is true, except for:

expression	TRUE/FALSE
`0`	FALSE
`None`	FALSE
`False`	FALSE
`''` (empty string)	FALSE
`[]` (empty list)	FALSE
`()` (empty tuple)	FALSE
`{}` (empty dictionary)	FALSE

Which means that these are True:

expression	TRUE/FALSE
`'0'`	TRUE
`'None'`	TRUE
`'False'`	TRUE
`'True'`	TRUE
`' '` (string of one blank space)	TRUE

Use `bool()` to test for truth

`bool()` is a function that will test if a value is true.

```
>>> bool(True)
True
>>> bool('True')
True
>>>
>>>
>>> bool(False)
False
>>> bool('False')
True
>>>
>>>
```

```
>>> bool(0)
False
>>> bool('0')
True
>>>
>>>
>>> bool('')
False
>>> bool(' ')
True
>>>
>>>
>>> bool(())
False
>>> bool([])
False
>>> bool({})
False
```

Logic: Control Statements

Control Statements are used to direct the flow of your code and create the opportunity for decision making. The foundation of control statements is building on truth.

If Statement

- Use the `if` Statement to test for truth and to execute lines of code if true.
- When the expression evaluates to true each of the statements indented below the `if` statment, also known as the nested statement block, will be executed.

if

```
if expression :
  statement
  statement
```

For Example:

```
dna = 'GTACCTTGATTTCGTATTCTGAGAGGCTGCTGCTTAGCGGTAGCCCCTTGGTTTCCGTGGCAACGGAAAA'
if 'AGC' in dna:
  print('found AGC in your dna sequence')
```

Returns:

```
found AGC in your dna sequence
```

else

- The `if` portion of the if/else statement behaves as before.
- The first indented block is executed if the condition is true.
- If the condition is false, the second indented else block is executed.

```
dna = 'GTACCTTGATTTCGTATTCTGAGAGGCTGCTGCTTAGCGGTAGCCCCTTGGTTTCCGTGGCAACGGAAAA'
if 'ATG' in dna:
  print('found ATG in your dna sequence')
else:
  print('did not find ATG in your dna sequence')
```

Returns:

```
did not find ATG in your dna sequence
```

if/elif

- The `if` condition is tested as before, and the indented block is executed if the condition is true.
- If it's false, the indented block following the `elif` is executed if the first `elif` condition is true.
- Any remaining `elif` conditions will be tested in order until one is found to be true. If none is true, the `else` indented block is executed.

```
count = 60
if count < 0:
  message = "is less than 0"
  print(count, message)
elif count < 50:
  message = "is less than 50"
  print (count, message)
elif count > 50:
  message = "is greater than 50"
  print (count, message)
else:
  message = "must be 50"
  print(count, message)
```

Returns:

```
60 is greater than 50
```

Let's change count to 20, which statement block gets executed?

```
count = 20
if count < 0:
  message = "is less than 0"
  print(count, message)
elif count < 50:
  message = "is less than 50"
  print (count, message)
elif count > 50:
  message = "is greater than 50"
  print (count, message)
else:
  message = "must be 50"
  print(count, message)
```

Returns:

```
20 is less than 50
```

What happens when count is 50?

```
count = 50
if count < 0:
  message = "is less than 0"
  print(count, message)
elif count < 50:
  message = "is less than 50"
  print (count, message)
elif count > 50:
  message = "is greater than 50"
  print (count, message)
else:
  message = "must be 50"
  print(count, message)
```

Returns:

```
50 must be 50
```

Numbers

Python recognizes 3 types of numbers: integers, floating point numbers, and complex numbers.

integer

- known as an int
- an int can be positive or negative
- and **does not** contain a decimal point or exponent.

floating point number

- known as a float
- a floating point number can be positive or negative
- and **does** contain a decimal point (`4.875`) or exponent (`4.2e-12`)

complex number

- known as complex
- is in the form of a+bi where bi is the imaginary part.

Conversion functions

Sometimes one type of number needs to be changed to another for a function to be able to do work on it. Here are a list of functions for converting number types:

function	Description
`int(x)`	to convert x to a plain integer
`float(x)`	to convert x to a floating-point number
`complex(x)`	to convert x to a complex number with real part x and imaginary part zero
`complex(x, y)`	to convert x and y to a complex number with real part x and imaginary part y

```
>>> int(2.3)
2
>>> float(2)
2.0
>>> complex(2.3)
(2.3+0j)
>>> complex(2.3,2)
(2.3+2j)
```

Numeric Functions

Here is a list of functions that take numbers as arguments. These do useful things like rounding.

function	Description
`abs(x)`	The absolute value of x: the (positive) distance between x and zero.
`round(x[,n])`	x rounded to n digits from the decimal point. round() rounds to an even integer if the value is exactly between two integers, so round(0.5) is 0 and round(-0.5) is 0. round(1.5) is 2. **Rounding to a fixed number of decimal places can give unpredictable results.**
`max(x1, x2,...)`	The largest argument is returned
`min(x1, x2,...)`	The smallest argument is returned

```
>>> abs(2.3)
2.3
>>> abs(-2.9)
2.9
>>> round(2.3)
2
>>> round(2.5)
2
>>> round(2.9)
3
>>> round(-2.9)
-3
>>> round(-2.3)
-2
>>> round(-2.009,2)
-2.01
>>> round(2.675, 2)  # note this rounds down
2.67
>>> max(4,-5,5,1,11)
11
>>> min(4,-5,5,1,11)
-5
```

Many numeric functions are not built into the Python core and need to be imported into our script if we want to use them. To include them, at the top of the script type:
`import math`

These next functions are found in the math module and need to be imported. To use these functions, prepend the function with the module name, i.e, `math.ceil(15.5)`

math.function	Description
`math.ceil(x)`	return the smallest integer greater than or equal to x is returned
`math.floor(x)`	return the largest integer less than or equal to x.
`math.exp(x)`	The exponential of x: e^x is returned
`math.log(x)`	the natural logarithm of x, for x > 0 is returned
`math.log10(x)`	The base-10 logarithm of x for x > 0 is returned
`math.modf(x)`	The fractional and integer parts of x are returned in a two-item tuple.
`math.pow(x, y)`	The value of x raised to the power y is returned
`math.sqrt(x)`	Return the square root of x for x >= 0

```
>>> import math
>>>
>>> math.ceil(2.3)
3
>>> math.ceil(2.9)
3
>>> math.ceil(-2.9)
-2
>>> math.floor(2.3)
2
>>> math.floor(2.9)
2
>>> math.floor(-2.9)
-3
>>> math.exp(2.3)
9.974182454814718
>>> math.exp(2.9)
18.17414536944306
>>> math.exp(-2.9)
0.05502322005640723
>>>
>>> math.log(2.3)
0.8329091229351039
>>> math.log(2.9)
1.0647107369924282
>>> math.log(-2.9)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: math domain error
>>>
```

```
>>> math.log10(2.3)
0.36172783601759284
>>> math.log10(2.9)
0.4623979978989561
>>>
>>> math.modf(2.3)
(0.2999999999999998, 2.0)
>>>
>>> math.pow(2.3,1)
2.3
>>> math.pow(2.3,2)
5.289999999999999
>>> math.pow(-2.3,2)
5.289999999999999
>>> math.pow(2.3,-2)
0.18903591682419663
>>>
>>> math.sqrt(25)
5.0
>>> math.sqrt(2.3)
1.51657508881031
>>> math.sqrt(2.9)
1.70293863659264
```

Comparing two numbers

Oftentimes, it is necessary to compare two numbers and find out if the first number is less than, equal to, or greater than the second.

The simple function `cmp(x,y)` is not available in Python 3.

Use this idiom instead:

```
cmp = (x>y)-(x<y)
```

It returns three different values depending on x and y

- if x<y, then -1 is returned
- if x>y, then 1 is returned
- x == y, then 0 is returned

Link to Python 2 Problem Set
