Python Reader

Introduction

Interpreter

Variables

print

input

Math and expressions

Constants

random

Booleans

if

while

for

Back to Code in Place

Python Math

Numbers - int and float

Surprisingly, there are two distinct types of numbers for doing arithmetic in a computer - "int" for whole integer numbers like 6 and 42 and -3, and "float" for numbers like 3.14 with a decimal fraction.

Int Type

The Python "int" type represents whole integer values like 12 and -2. Addition, subtraction, and multiplication and division work the usual way with the operators: `+ - * /`. Division by zero is an error.

```
>>> 1 + 10 - 2
9
>>> 2 * 3 * 4
24
>>> 2 * 6 / 3
4.0
>>> 6 / 0
ZeroDivisionError: division by zero
```

Precedence

Just as in regular mathematics, multiplication and division have higher "precedence" than addition and subtraction, so they are evaluated first in an expression. After accounting for precedence, the arithmetic is done left-to-right.

e.g. here the multiplication happens first, then the addition:

```
>>> 1 + 2 * 3
7
>>> 1 + 3 * 3 + 1
11
```

Add parenthesis into the code to control which operations are evaluated first:

```
>>> (1 + 2) * 3
9
```

60 / 2 * 3

What is the value of `60 / 2 * 3'?

The evaluation proceeds left-to-right, applying each operator to a running result which is simple but can be unintuitive. For `60 / 2 * 3`, the steps are..

```
1. start with 60
2. 60 / 2 yielding 30.0
3. 30.0 * 3 yielding 90.0
```

The 2 is in the denominator, but the 3 is not. Add parenthesis to put both numbers in the denominator e.g. `60 / (2 * 3)`

```
>>> 60 / 2 * 3
90.0
>>> 60 / (2 * 3)
10.0
```

Division / Yields Float

One problem problem with / is that it does not produce an int, it produces a float. This is basically reasonable — 7 divided by 2 isn't an integer.

```
>>> 7 / 2
3.5        # a float, notice the "."
```

Adding subtracting or multiplying two ints always yields an int result, but division is different. The result of division is always a float value, even if the division comes out even.

```
>>> 9 / 2
4.5
>>> 8 / 2
4.0
>>> 101 / 8
12.625
```

// int Division

Many times an algorithm makes the most sense if all of the values are kept as ints, so we need an alternative to the / which produces floats. In Python the int-division operator // rounds down any fraction, always yielding an int result.

```
>>> 9 / 2    # "/" yields a float, not what we wanted
4.5
>>> 9 // 2   # "//" rounds down to int
4
>>> 8 // 2
4
>>> 87 // 8
10
```

njp Since the int division // rounds values down, it's important to do the division **last** .. could have example about this >>> 10 * 200 // 11 181 >>> 10 // 11 * 200 0 >>> # 10 bags of cookies, 200 cookies per back, divided among 11 cars

** Exponentiation

The ** operator does exponentiation, e.g. 3 ** 2 is 3^2

```
>>> 3 ** 2
9
>>> 2 ** 10
1024
```

Unlike most programming languages, Python int values do not have a maximum. Python allocates more and more bytes to store the int as it gets larger. The number of grains of sand making up the universe when I was in school was thought to be about 2^{100}, playing the role of handy very-large-number (I think it's bigger now as they keep finding more universe, but this number is handy). In Python, we can write an expression with that number and it just works.

```
>>> 2 ** 100
1267650600228229401496703205376
>>> 2 ** 100 + 1
1267650600228229401496703205377
```

Memory use approximation: int values of 256 or less are stored in a special way that uses very few bytes. Other ints take up about 24 bytes each in RAM.

Int Mod %

The "modulo" or "mod" operator % is essentially the remainder after division. So (23 % 10) yields 3 — divide 23 by 10 and 3 is the leftover remainder.

Python Reader

Introduction

Interpreter

Variables

print

input

Math and expressions

Constants

random

Booleans

if

while

for

Back to Code in Place

```
>>> 23 % 10
3
>>> 36 % 10
6
>>> 43 % 10
3
>>  40 % 10  # mod result 0 = divides evenly
0
>>> 17 % 5
2
>>> 15 % 5
0
```

If the modulo result is 0, it means the division can out evenly, e.g. `40 % 10` above. The best practice is to only use mod with non-negative numbers. Modding by 0 is an error, just like dividing by 0.

```
>>> 43 % 0
ZeroDivisionError: integer division or modulo by zero
```

Review Expressions

What is the value of each expression? Write the result as int (6) or float (6.0).

```
>>> 2 * 1 + 6
??
>>> 20 / 4 + 1
??
>>> 20 / (4 + 1)
??
>> 40 / 2 * 2
??
>>> 5 ** 2
??
>>> 7 / 2
??
>>> 7 // 2
??
>>> 13 % 10
??
>>> 20 % 10
??
>>> 42 % 20
??
>>> 31 % 20
??
```

Show

Float Type

Floating point numbers are used to do math with real quantities, such as a velocity or angle. The regular math operators `+ - * / **` all work with floats, producing a float result. If an expression mixes some int values and some float values, the math is converted to float - a one-way street called "promotion" to float.

```
>>> 1.0 + 2.0 * 3.0
7.0
>>>
>>> 1 + 2 * 3.0
7.0
>>>
>>> 2.1 ** 2
4.41
```

Float values can be written in scientific notation with the letter 'e' or 'E', like this:

```
>>> 1.2e23 * 10
1.2e+24
>>> 1.0e-4
0.0001
```

Float Error Term

Famously, floating point numbers have a tiny error term that builds up way off 15 digits or so to the right. Mostly this error is not shown when the float value is printed, as a few digits are not printed. However, the error digits are real, throwing the float value off a tiny amount. The error term will appear sometimes, just as a quirk of how many digits are printed (see below). This error term is an intrinsic limitation of floating point values in the computer. (Perhaps also why CS people are drawn to do their algorithms with int.)

```
>>> 0.1 + 0.1
0.2
>>> 0.1 + 0.1 + 0.1
0.30000000000000004
>>> 0.1 + 0.1 + 0.1 + 0.1 + 0.1 + 0.1 + 0.1
0.7
>>> 0.1 + 0.1 + 0.1 + 0.1 + 0.1 + 0.1 + 0.1 + 0.1
0.7999999999999999
```

Mostly, an error 15 or so digits off to the right does not invalidate your computation. However, it means that code should not use `==` with float values, since the comparison will be thrown off by the error term. To compare float values, subtract them and compare the absolute value of the difference to some small delta.

```
>>> # are float values a and b the same?
>>> diff = abs(a - b)   # abs() is absolute value
>>> if diff < 1.0e-9:   # if diff less than 1 billionth,
    ...
```

Memory use approximation: float values take up about 24 bytes apiece.