Numbers

WE'LL COVER THE FOLLOWING

Numbers are awesome. There are so many to choose from. Python supports both integers and floating point numbers. There's no type declaration to distinguish them; Python tells them apart by the presence or absence of a decimal point.

```
print (type(1))                 #①
#<class 'int'>

print (isinstance(1, int) )     #②
#True

print (1 + 1 )                  #③
#2

print (1 + 1.0 )                #④
#2.0

print (type(2.0))
#<class 'float'>
```

① You can use the `type()` function to check the type of any value or variable. As you might expect, `1` is an `int`.

② Similarly, you can use the `isinstance()` function to check whether a value

or variable is of a given type.

③ Adding an `int` to an `int` yields an `int`.

④ Adding an `int` to a `float` yields a `float`. Python coerces the `int` into a `float` to perform the addition, then returns a `float` as the result.

Coercing Integers To Floats And Vice-Versa

As you just saw, some operators (like addition) will coerce integers to floating point numbers as needed. You can also coerce them by yourself.

```
print (float(2) )               #①
#2.0

print (int(2.0))                #②
#2

print (int(2.5) )               #③
#2

print (int(-2.5) )              #④
#-2

print (1.12345678901234567890)  #⑤
#1.1234567890123457

print (type(1000000000000000))  #⑥
#<class 'int'>
```

① You can explicitly coerce an `int` to a `float` by calling the `float()` function.

② Unsurprisingly, you can also coerce a `float` to an `int` by calling `int()`.

③ The `int()` function will truncate, not round.

④ The `int()` function truncates negative numbers towards 0. It's a true truncate function, not a floor function.

⑤ Floating point numbers are accurate to 15 decimal places.

⑥ Integers can be arbitrarily large.

> *Python 2 had separate types for **int** and **long**. The **int** datatype was limited*

> *by **sys.maxint**, which varied by platform but was usually 232-1. Python 3 has just one **integer** type, which behaves mostly like the old **long** type from Python 2. See pep 237 for details.*

Common Numerical Operations

You can do all kinds of things with numbers.

```
print (11 / 2)      #①
#5.5

print (11 // 2)     #②
#5

x = -11
print ( x // 2)     #③
#-6

print (11.0 // 2)   #④
#5.0

print (11 ** 2)     #⑤
#121

print (11 % 2)      #⑥
#1
```

① The `/` operator performs floating point division. It returns a `float` even if both the numerator and denominator are `ints`.

② The `//` operator performs a quirky kind of integer division. When the result is positive, you can think of it as truncating (not rounding) to 0 decimal places, but be careful with that.

③ When integer-dividing negative numbers, the `//` operator rounds "up" to the nearest integer. Mathematically speaking, it's rounding "down" since `-6` is less than `-5`, but it could trip you up if you were expecting it to truncate to `-5`.

④ The `//` operator doesn't always return an integer. If either the numerator or denominator is a `float`, it will still round to the nearest integer, but the actual return value will be a `float`.

⑤ The `**` operator means “raised to the power of.” `11^2` is `121`.

⑥ The `%` operator gives the remainder after performing integer division. `11` divided by `2` is `5` with a remainder of `1`, so the result here is `1`.

> *In Python 2, the / operator usually meant integer division, but you could make it behave like floating point division by including a special directive in your code. In Python 3, the / operator always means floating point division. See PEP 238 for details.*

Fractions

Python isn’t limited to integers and floating point numbers. It can also do all the fancy math you learned in high school and promptly forgot about.

```
import fractions                        #①
x = fractions.Fraction(1, 3)            #②
print (x)
#Fraction(1, 3)

print (x * 2 )                          #③
#Fraction(2, 3)

print (fractions.Fraction(6, 4))        #④
#Fraction(3, 2)

print (fractions.Fraction(0, 0) )       #⑤
#Traceback (most recent call last):
# File "/usercode/__ed_file.py", line 12, in <module>
# print (fractions.Fraction(0, 0) ) #\u2464
# File "/usr/lib/python3.4/fractions.py", line 167, in __new__
# raise ZeroDivisionError('Fraction(%s, 0)' % numerator)
#ZeroDivisionError: Fraction(0, 0)
```

① To start using fractions, import the `fractions` module.

② To define a fraction, create a `Fraction` object and pass in the numerator and denominator.

③ You can perform all the usual mathematical operations with fractions. Operations return a new `Fraction` object. `2 * (1/3) = (2/3)`

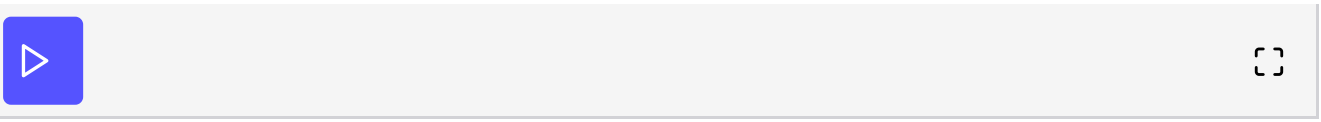
④ The `Fraction` object will automatically reduce fractions. `(6/4) = (3/2)`

⑤ Python has the good sense not to create a fraction with a zero denominator.

Trigonometry

You can also do basic trigonometry in Python.

```
import math
print (math.pi )                #①
#3.141592653589793

print (math.sin(math.pi / 2))   #②
#1.0

print (math.tan(math.pi / 4))   #③
#0.99999999999999989
```



① The `math` module has a constant for `π`, the ratio of a circle's circumference to its diameter.

② The `math` module has all the basic trigonometric functions, including `sin()`, `cos()`, `tan()`, and variants like `asin()`.

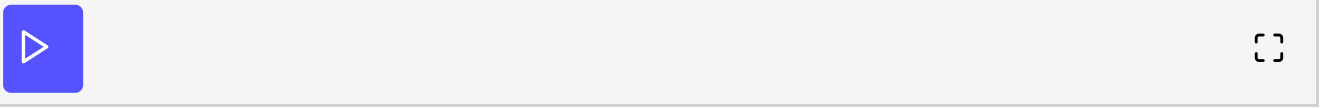
③ Note, however, that Python does not have infinite precision. `tan(π / 4)` should return `1.0`, not `0.99999999999999989`.

Numbers In A Boolean Context

Zero values are false, and non-zero values are true. You can use numbers in a boolean context, such as an `if` statement. Zero values are false, and non-zero values are true.

```
def is_it_true(anything):             #①
   if anything:
     print("yes, it's true")
   else:
     print("no, it's false")


print (is_it_true(1) )                #②
#yes, it's true
#None
```

```
print (is_it_true(-1))
#yes, it's true
#None


print (is_it_true(0))
#no, it's false
#None

print (is_it_true(0.1) )                        #③
#yes, it's true
#None

print (is_it_true(0.0))
#no, it's false
#None

import fractions
print (is_it_true(fractions.Fraction(1, 2)))  #④
#yes, it's true
#None

print (is_it_true(fractions.Fraction(0, 1)))
#no, it's false
#None
```



① Did you know you can define your own functions in the Python interactive shell? Just press `ENTER` at the end of each line, and `ENTER` on a blank line to finish.

② In a boolean context, non-zero integers are true; 0 is false.

③ Non-zero floating point numbers are true; `0.0` is false. Be careful with this one! If there's the slightest rounding error (not impossible, as you saw in the previous section) then Python will be testing `0.0000000000001` instead of `0` and will return `True`.

④ Fractions can also be used in a boolean context. `Fraction(0, n)` is false for all values of `n`. All other fractions are true.