Python Bootcamp & Masterclass

float

gk nxt

A float (Python's name for floating-point number) is a real number written with a decimal point dividing the integer and fractional parts.

Floats may also be in scientific notation, with E or e indicating the power of `10`

(for example, `2.5e+2` = $2.5 \times 10^{2} = 250$ and `2.5e-2` = $2.5 \times 10^{-2} = 0.025$)

Examples:

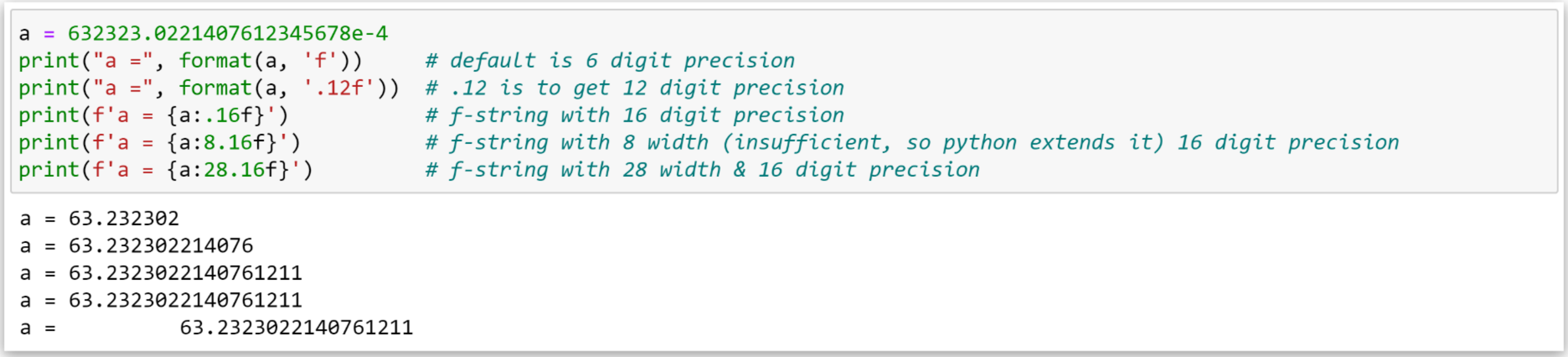
- - 66.67
- 89.29
- `6. 02214076e+23`

Floats are immutable (they can't be modified after they've been created)

gk nxt

- To convert scientific notation to regular float notation, `format ( )` option can be used.

- Formatting with `'f'` displays the number as a fixed-point number. The default precision is 6 digits. The precision can be changed by specifying the required precision (for example, `'.12f'` for 12 digit precision)

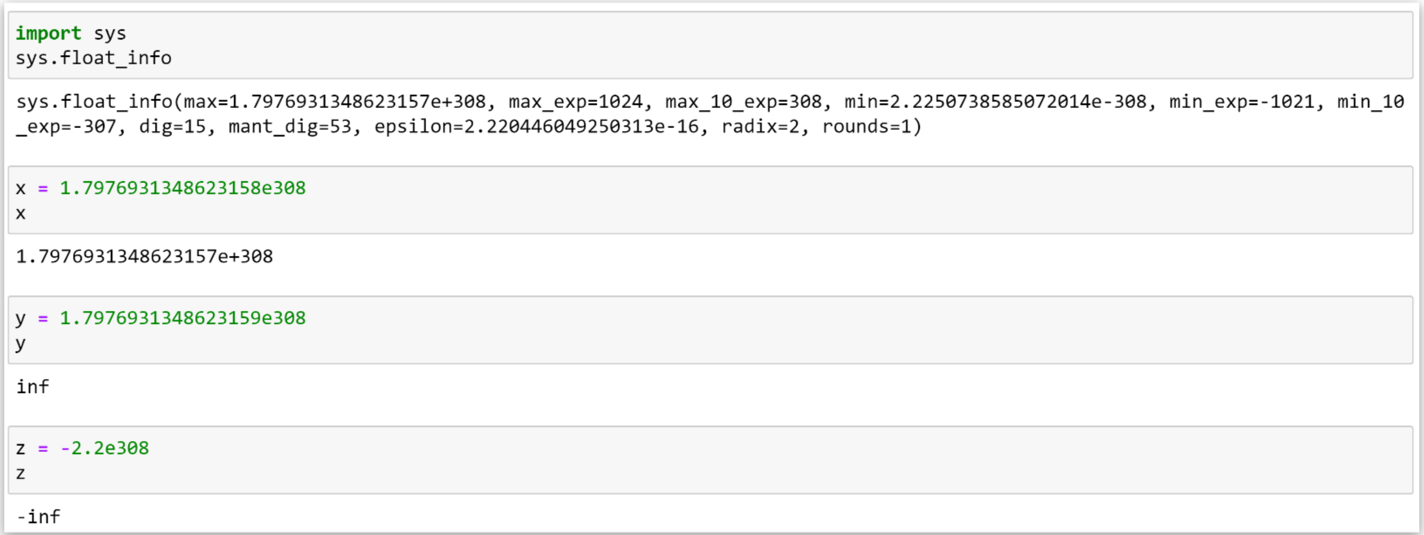


```
a = 632323.02214076123456 78e-4
print("a =", format(a, 'f'))     # default is 6 digit precision
print("a =", format(a, '.12f'))  # .12 is to get 12 digit precision
print(f'a = {a:.16f}')           # f-string with 16 digit precision
print(f'a = {a:8.16f}')          # f-string with 8 width (insufficient, so python extends it) 16 digit precision
print(f'a = {a:28.16f}')         # f-string with 28 width & 16 digit precision
```

```
a = 63.232302
a = 63.232302214076
a = 63.2323022140761211
a = 63.2323022140761211
a =          63.2323022140761211
```

gknxt

- The maximum value a floating-point number can have is OS/hardware-dependent.
- Python will indicate a float greater than the maximum possible floating-point number for the system by `inf` (to indicate infinity)
- Python uses `-inf` (negative infinity) for a negative floating-point number that is beyond the minimum possible floating-point number for the system)



```
import sys
sys.float_info
```

```
sys.float_info(max=1.7976931348623157e+308, max_exp=1024, max_10_exp=308, min=2.2250738585072014e-308, min_exp=-1021, min_10
_exp=-307, dig=15, mant_dig=53, epsilon=2.220446049250313e-16, radix=2, rounds=1)
```

```
x = 1.7976931348623158e308
x
```

```
1.7976931348623157e+308
```

```
y = 1.7976931348623159e308
y
```

```
inf
```

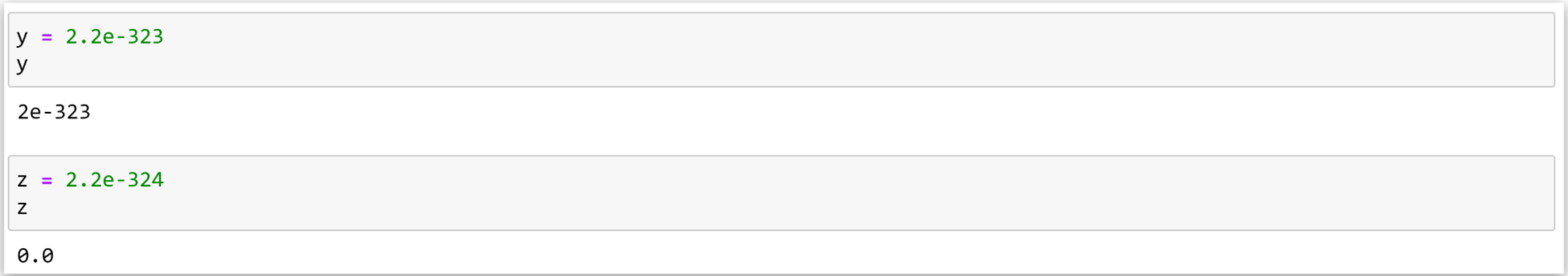
```
z = -2.2e308
z
```

```
-inf
```

gk nxt

The closest nonzero number that will be approximated to zero is OS/hardware-dependent.

Any floating-point number smaller than that will be rounded to zero.



```
y = 2.2e-323
y
```

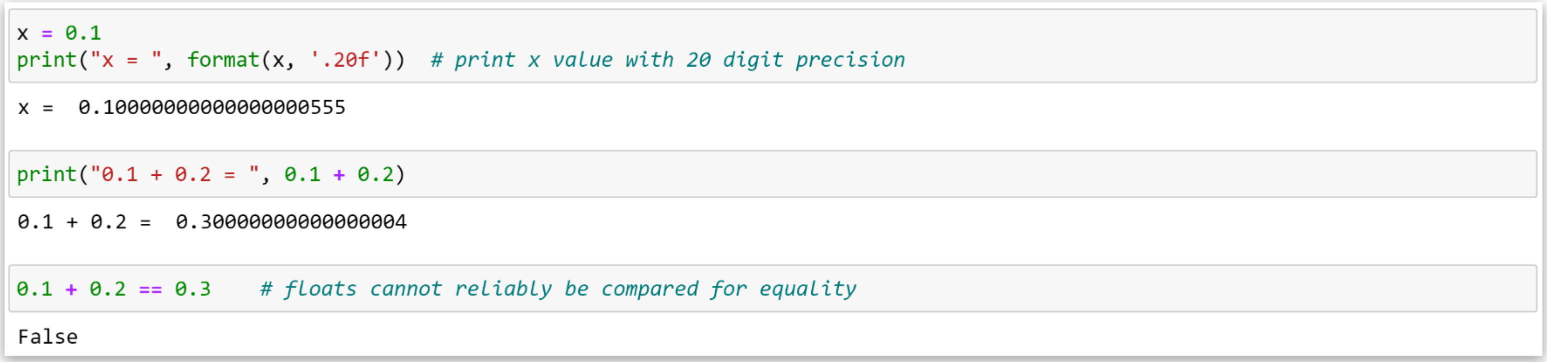
```
2e-323
```

```
z = 2.2e-324
z
```

```
0.0
```

Computers natively represent floats using base 2 So some decimals can be represented exactly (`0.5, 0.75` etc.), but many others only approximately (`0.1, 0.2` etc.)

A consequence is that, in general, the decimal floating-point numbers are approximated by the binary floating-point numbers actually stored in the machine. So, some rounding issues should be anticipated when floating-point arithmetic is used. Python floats provide reliable accuracy for up to 17 significant digits on a typical implementation.



```python
x = 0.1
print("x = ", format(x, '.20f'))  # print x value with 20 digit precision
```

```
x =  0.10000000000000000555
```

```python
print("0.1 + 0.2 = ", 0.1 + 0.2)
```

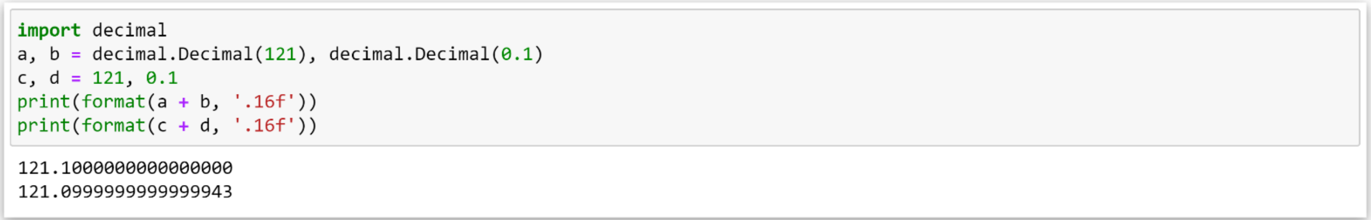
```
0.1 + 0.2 =  0.30000000000000004
```

```python
0.1 + 0.2 == 0.3    # floats cannot reliably be compared for equality
```

```
False
```

Computers effectively store many floating-point numbers as approximations. So, this is not a problem specific to Python - every programming language have this problem with floating-point numbers. Starting from version 3.1, Python is using David Gay's algorithm wherever possible to find the fewest possible digits preserving as much accuracy as possible.

Though for most domains this is not a problem, some scientific and finance applications (e.g. currency conversion) need high precision. Python's decimal.Decimal numbers from the decimal module ensures calculations that are accurate to the level of precision specified (by default, to 28 decimal places) and can represent periodic numbers like `0.1` exactly. Downside of using decimal.Decimal numbers is that the processing is a lot slower than with floats.



```
import decimal
a, b = decimal.Decimal(121), decimal.Decimal(0.1)
c, d = 121, 0.1
print(format(a + b, '.16f'))
print(format(c + d, '.16f'))
```

```
121.1000000000000000
121.0999999999999943
```

Division operations involving integers and/or floats will always result in a float

```
4/2
```

```
2.0
```

```
4.0/2
```

```
2.0
```

```
4.0/2.0
```

```
2.0
```

```
4/2.0
```

```
2.0
```

Online Resources

For best python resources, please visit:

gknxt.com/python/

gk nxt

Python
Bootcamp
& Masterclass

Thank You
for your Rating & Review

gknxt