# Scalar Types

**Austin Bingham**
COFOUNDER - SIXTY NORTH

@austin_bingham

**Robert Smallshire**
COFOUNDER - SIXTY NORTH

@robsmallshire

## Overview

Python's fundamental scalar types

Basic use of relational operators

Basic flow-control mechanisms

## Scalar Types

# Scalar Types

| | |
|---|---|
| int<br>42 | arbitrary precision integer |
| float<br>4.2 | 64-bit floating point numbers |
| NoneType<br>None | the null object |
| bool<br>True   bool<br>False | boolean logical values |

`int`

unlimited precision signed integer

## Int

```
>>> 10
10
>>> 0b10
2
>>> 0o10
8
>>> 0x10
16
>>> int(3.5)
3
>>> int(-3.5)
-3
>>> int("496")
496
>>> int("10000", 3)
81
>>>
```

## `float`

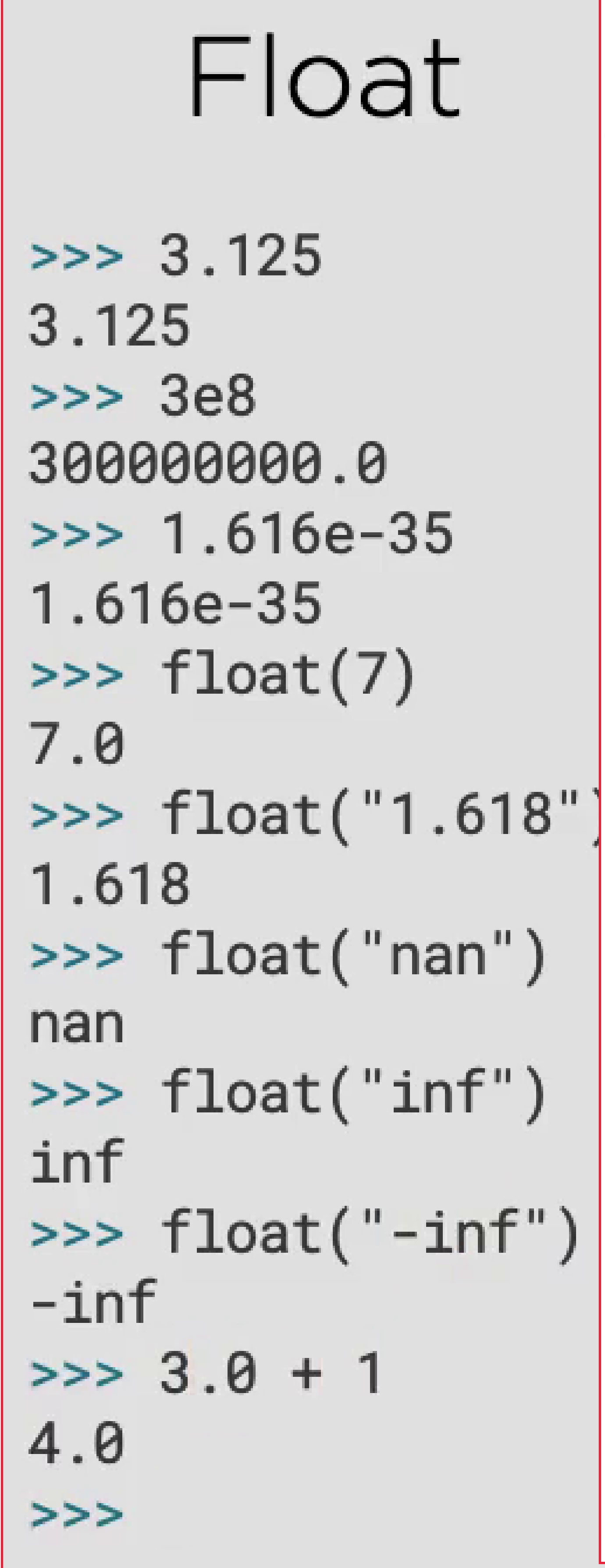
IEEE-754 double-precision with 53-bits of binary precision

15-16 significant digits in decimal

Python Float:
1) Floating-point numbers are supported in python by the float type. Python floats are implemented as IEEE 754 double-precision floating-point numbers with 53 bits of binary precision. This is equivalent to between 15 and 16 significant digits and decimals.

2) Any literal number containing a decimal point is interpreted by python as a float.

3) Scientific notation can be used so for large numbers such as the approximate speed of light we could use 3e8 , and for small numbers like plank, constant 1.616 times 10 to the negative 35th We can enter 1.616e-35. Python automatically switches the display representation. That is the format it prints to the REPL to the most readable form.



## Float

```
>>> 3.125
3.125
>>> 3e8
300000000.0
>>> 1.616e-35
1.616e-35
>>> float(7)
7.0
>>> float("1.618"
1.618
>>> float("nan")
nan
>>> float("inf")
inf
>>> float("-inf")
-inf
>>> 3.0 + 1
4.0
>>>
```

4) As with integers, we can convert to floats from other numerical string types. Using the float constructor, we can pass into values to the float constructor, and we can pass strings.
his is also how we create the special floating point values nan or not a number as well as positive infinity and negative infinity.Þxamples: float(7), float(1.618), float("nan"),float("inf"),float("-inf")

5) The result of any calculation involving into in float is promoted to a float.

Python has a special no value called None. Spelled with a capital N .

None is frequently used to represent the absence of a value. The python REPL never prints None results. So typing none into the REPL has no effect.

None could be bounded to variable names just like any other object, and we can test  whether an object is None by using pythons is operator.

The object None is of the type NoneType.

## None

Null value

Often represents the absence of a value

## None

```
>>> None
>>> a = None
>>> a is None
True
>>>
```

`bool`

Boolean logical values

1)The Bool type represents logical states and plays an important role in several of pythons control flow structures as well.

2)There are two Bool values: True in False, both spelled with initial capitals. There is also a Bool constructor which could be used to convert from other types to Bool.

3)There is also a Bool constructor which could be used to convert from other types to Bool. Let's look at how it works. For integer zero is considered falsy and all other values truthy. We see the same behavior with floats where only zero is considered falsy.

4)When converting from collections such as strings or lists, empty collections are treated as falsy. The empty list is falsy While any non empty list is truthy. Similarly empty strings are falsy while any other strings are truthy.

5) It's worth noting that the Bool constructor may not behave as you expect when passing in the strings "True" and "False", since both are non empty strings. Both result in True

6) These conversions to Bool are important because they're widely used in python if statements and while loops which expect Bool values into the condition.

## Bool

```
False
>>> bool(42)
True
>>> bool(-1)
True
>>> bool(0.0)
False
>>> bool(0.207)
True
>>> bool(-1.117)
True
>>> bool([])
False
>>> bool([1, 5, 9])
True
>>> bool("")
False
>>> bool("Spam")
True
>>> bool("False")
True
>>> bool("True")
True
>>>
```

## Relational Operators

## Relational Operators

| | |
|---|---|
| == | value equality / equivalence |
| != | value inequality / inequivalence |
| < | less-than |
| > | greater-than |
| <= | less-than or equal |
| >= | greater-than or equal |

## Value Equality

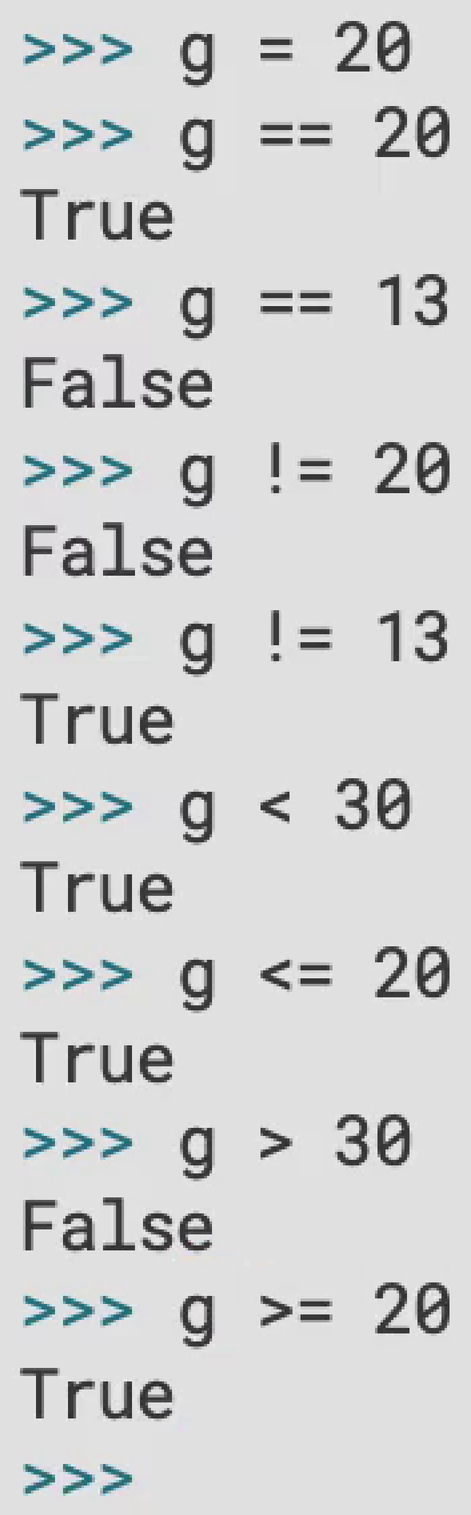
==

True

# Relational Operators

1) Bool values are commonly produced by pythons relational operators, which could be used for comparing objects. These include value equality or equivalents, value inequality or inequivalence, less than greater than less than or equal to and greater than or equal to.

2) We test for equality with the double equals operator(==), and we test for inequality using the not equals operator an exclamation point followed by an equal sign(!=).



```
>>> g = 20
>>> g == 20
True
>>> g == 13
False
>>> g != 20
False
>>> g != 13
True
>>> g < 30
True
>>> g <= 20
True
>>> g > 30
False
>>> g >= 20
True
>>>
```

## Control Flow

Two important control flow structures which depend on the conversions to the Bool type:

1) If statements and
2) while loops

## Conditional statement

Branch execution based on the value of an expression

The if statements are also known as conditional statements.
Conditional statements allow us to branch execution based on the value of an expression.
The form of the statement is the if keyword, followed by an expression terminated by a colon to introduce a new block.

## If-statement Syntax

```
if expression:
    block
```

# If-statement

```
>>> if True:
...     print("It's true!")
...
It's true!
>>> if False:
...     print("It's true!")
...
>>> if bool("eggs"):
...     print("Yes please!")
...
Yes please!
>>> if "eggs":
...     print("Yes please!")
...
Yes please!
>>>
```

The expression used with the if Statement will be converted to a Bool, just as if the Bool constructor had been used.

So explicitly Constructing the Bool in the if statement is exactly equivalent to using a bare string.

Thanks to this useful shorthand explicit conversion to Bool using the Bool constructor is rarely used in python.

# Else-clause

```
>>> if h > 50:
...     print("Greater than 50")
... else:
...     print("50 or smaller")
...
50 or smaller
>>> if h > 50:
...     print("Greater than 50")
... else:
...     if h < 20:
...         print("Less than 20")
...     else:
...         print("Between 20 and 50")
...
Between 20 and 50
>>> if h > 50:
...     print("Greater than 50")
... elif h < 20:
...     print("Less than 20")
... else:
...     print("Between 20 and 50")
...
Between 20 and 50
>>>
```

The if statement supports the optional else clause that goes in a block introduced by the else keyword, followed by colon, which is indented to the same level as the if keyword.

Whenever you find yourself tempted to nest if statements inside else blocks, you should consider using pythons elif key word, which is a combined else - if

## While-loops

## While-loops

```
while expression:
    block
```

converted to boolean

The while loops in  python are introduced by the while keyword, which is followed by a Boolean expression. As with the condition for if statements, **the expression is implicitly converted to a Boolean value,** as if it had been passed to the Bool constructor. The while statement is terminated by a colon because it introduces a new block.

# While-loops

Another new language feature here is the use of the **augmented assignment operator**, a minus sign followed by an equal sign to subtract one from the value of C on each iteration. Similar augmented assignment operators exist for other basic math operators, such as plus and multiply

```
>>> c = 5
>>> while c != 0:
...     print(c)
...     c -= 1
...
5
4
3
2
1
>>> c = 5
>>> while c:
...     print(c)
...     c -= 1
...
5
4
3
2
1
>>>
```

Because the condition, also called the predicate, will be implicitly converted to Bool. Just as if a call to the Bool constructor present, we could replace the above code with the following version. This works because the conversion of the integer value of C two Bool results in True, until we get to zero, which converts to False.

## Int Truthiness

```
bool(5) == True
bool(4) == True
...
bool(0) == False
```

*Explicit is better than implicit*

## Relational Operators

```
>>> while True:
...     pass
...
Traceback (most recent call last):
  File "<stdin>", line 2, in <module>
KeyboardInterrupt
>>>
```

While loops are often used in python, where an infinite loop is required. We achieve this by simply passing true as the predicate expression to the while construct.

To get out of the loop, we press control+C. Python intercepts this to raise a special exception which terminates the loop.

## break

Many programming languages support a loop construct which places the predicate test at the end of the loop rather than at the beginning. For example, C, C++, C Sharp and Java support the do while construct other languages have repeat until loops.
This is not the case in Python, where the *idiom is to use, while true, together with an early exit facilitated by the break statement* . The break statement jumps out of the loop and only the innermost loop. If several loops have invested and then continues execution immediately after the loop body

Many languages support a loop ending in a predicate test

C, C++, C#, and Java have do-while

Python requires you to use `while True` and `break`

`break` jumps out of the inner-most executing loop to the line immediately after it

## Break

```
>>> while True:
...         response = input()
...         if int(response) % 7 == 0:
...                 break
...
12
67
34
28
>>>
```

On the first statement of the while block, we used the **built-in input function** to request a string from the user. We assigned that string to a variable called response. We now use an if statement to test whether the value provided is divisible by seven. We convert response to an integer, using the int constructor and then use the modulo operator to divide by seven and give the remainder.

If the remainder is equal to zero. The response was divisible by seven, and we enter the body of the if block Within the if block, we use the break keyword.

Break terminates the innermost loop, in this case the while loop and causes execution to jump to the first statement after the loop. In our case, this is the end of the program.

## Summary

`int`, `float`, `None`, and `bool`

Relational operators for equivalence and ordering

Conditional code with `if-elif-else`

While-loops

While-loop expressions converted to bool

## Summary

Interrupt loops with Control-C

Control-C generates a `KeyboardInterrupt` exception

Break out of loops with `break`

- Exits the inner-most executing loop
- Takes execution to the first statement following the loop

Augmented assignment operators like `+=`

Request text input from the user with `input()`