Operations

- None, bool
- basic operations
- strings
- += and friends

NoneType

- The type None has only one value: `None`

- Used when context requires a value, but none is really available

- **Example**: All functions must return a value. The function `print` has the *side-effect* of printing something to the standard output, but returns `None`

- **Example**: Initialize a variable with no value, e.g. list entries `mylist = [None, None, None]`

Python shell

```
> x = print(42)
| 42
> print(x)
| None
```

Type bool

- The type `bool` only has two values: `True` and `False`

- Logic truth tables:

x **or** *y*	True	False
True	True	True
False	True	False

x **and** *y*	True	False
True	True	False
False	False	False

x	**not** *x*
True	False
False	True

Scalar vs Non-scalar Types

- **Scalar types** (atomic/indivisible): `int`, `float`, `bool`, `None`
- **Non-scalar**: Examples `strings` and `lists`

```
"string"[3] = "i"
[2, 5, 6, 7][2] = 6
```

Questions – What is `[7,3,5][[1,2,3][1]]` ?

a) 1
b) 2
c) 3
d) 5
e) 7
f) Don’t know

Operations on int and float

Result is float if and only if at least one argument is float, except ** with negative exponent always gives a float

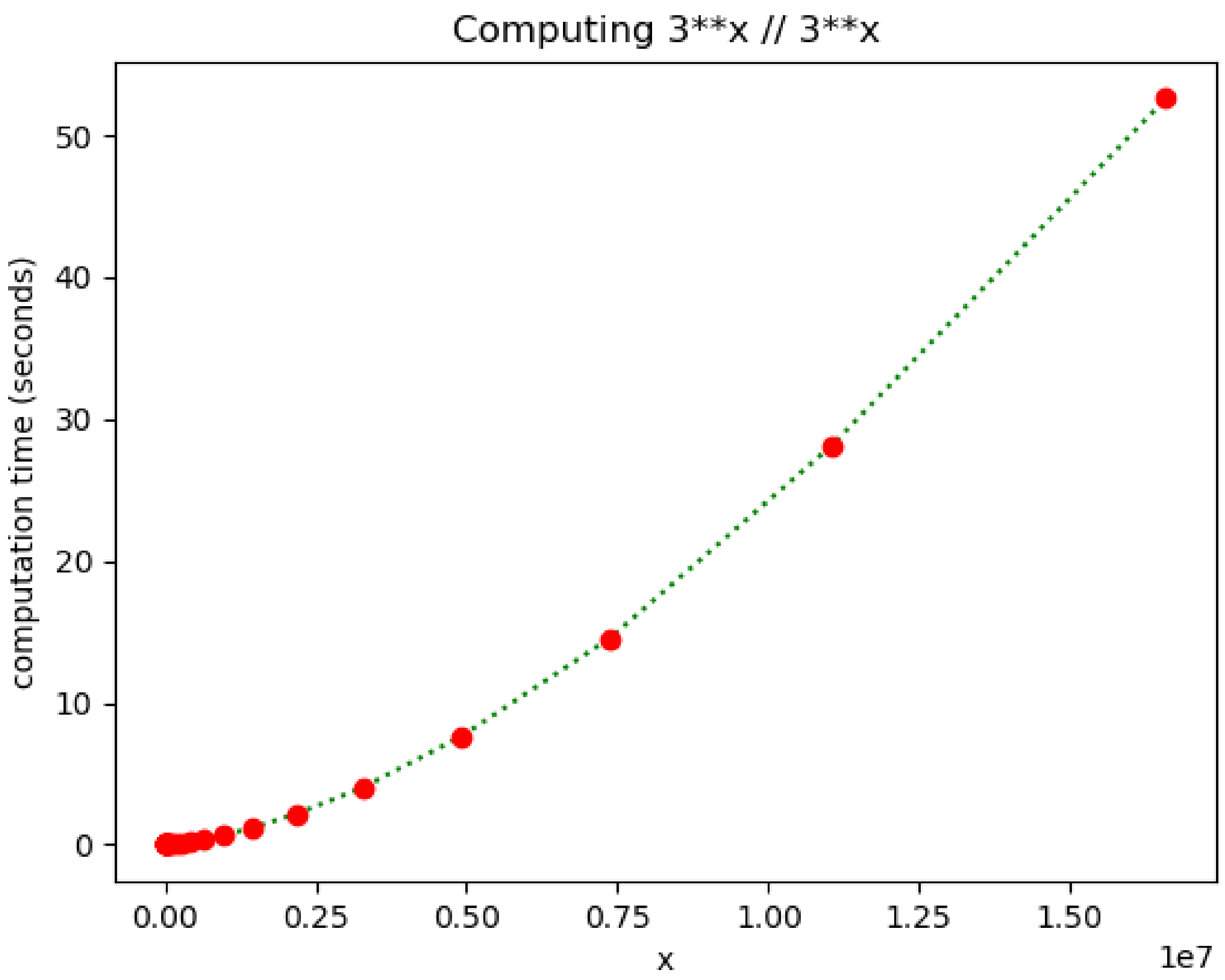
- `+, -, *` addition multiplication, e.g. `3.0*2` = `6.0`
- `**` and `pow(x, y)` power, e.g. `2**3=pow(2,3)=8`, `2**-2=0.25`
- `//` integer division = ⌊x / y⌋
  e.g. `15.0//4= 3.0`. Note: `-8//3=-3`
- `/` division returns float, `6/3=2.0`
- `abs(x)` absolute value
- `%` integer division remainder (modulo)
  `11%3 =2`
  `4.7%0.6=0.5000000000000003`

Python shell

```
> 0.4 // 0.1
| 4.0
> 0.4 / 0.1
| 4.0
> 0.3 // 0.1
| 2.0
> 0.3 / 0.1
| 2.9999999999999996
> 10**1000 / 2
| OverflowError: integer division
  result too large for a float
```

Running time for 3**x // 3**x

Working with larger integers takes slightly more than linear time in the number of digits



integer-division-timing.py

```
from time import time
import matplotlib.pyplot as plt

bits, compute_time = [], []

for i in range(42):
    x = 3**i // 2**i
    start = time()
    result = 3**x // 3**x     # the computation we time
    end = time()
    t = end - start
    print("i =", i, "x =", x, "Result =", result, "time(sec) =", t)
    bits.append(x)
    compute_time.append(t)

plt.title('Computing 3**x // 3**x')
plt.xlabel('x')
plt.ylabel('computation time (seconds)')
plt.plot(bits, compute_time, "g:")
plt.plot(bits, compute_time, "ro")
plt.show()
```

module math

Many standard mathematical functions are available in the Python module “`math`”, e.g.

sqrt, sin, cos, tan, asin, acos, atan, log(natural), log10, exp, ceil, floor, ...

- To use all the functions from the math module use `import math`
  Functions are now available as e.g. `math.sqrt(10)` and `math.ceil(7.2)`
- To import selected functions you instead write `from math import sqrt, ceil`
- The library also contains some constants, e.g.
  `math.pi` = 3.141592... and `math.e` = 2.718281...
- Note: `x ** 0.5` significantly faster than `sqrt(x)`

Python shell

```
> (0.1 + 0.2) * 10
| 3.0000000000000004
> math.ceil((0.1 + 0.2) * 10)
| 4
```

docs.python.org/3/library/math.html

Python shell

```
> import math
> math.sqrt(8)
| 2.8284271247461903
> from math import pi, sqrt
> pi
| 3.141592653589793
> sqrt(5)
| 2.23606797749979
> from math import sqrt as kvadratrod
> kvadratrod(3)
| 1.7320508075688772

> import timeit
> timeit.timeit("1e10**0.5")
| 0.021124736888936863
> timeit.timeit("sqrt(1e10)", "from math import sqrt")
| 0.13663140528657789
> timeit.timeit("math.sqrt(1e10)", "import math")
| 0.19466608416345582
```

docs.python.org/3.6/library/timeit.html

Rounding up integer fractions

- Python: $\lceil x/y \rceil = -(-x//y)$

-(-13/3)		
Python	**Java**	**C**
-(-13//3) = 5	-(-13/3) = 4 ⚠	-(-13/3) = 4 ⚠

⚠ The intermediate result `x/y` in `math.ceil(x/y)` is a float with limited precision

- Alternative computation:

  $\lceil x/y \rceil = (x+(y-1))//y$

Python shell

```
> from math import ceil
> from timeit import timeit
> 13 / 3
| 4.333333333333333
> 13 // 3
| 4
> -13 // 3
| -5
> -(-13 // 3)
| 5
> ceil(13 / 3)
| 5
> -(-22222222222222222223 // 2)
| 11111111111111111112
> ceil(22222222222222222223 / 2)
| 11111111111111110656
> timeit('ceil(13 / 3)', 'from math import ceil')
| 0.27746671276099731
> timeit('-(-13 // 3)')  # negation trick is fast
| 0.05231945830200857
```

floats : Overflow, inf, -inf, nan

- There exists special float values
  `inf`, `-inf`, `nan`
  representing “+infinity”, “-infinity” and “not a number”
- Can be created using e.g.
  `float('inf')`
  or imported from the `math` module
- Some overflow operations generate an `OverflowError`, other return `inf` and allow calculations to continue !
- Read the IEEE 754 standard if you want to know more details...

Python shell

```
> 1e250 ** 2
| OverflowError:
  (34, 'Result too large')
> 1e250 * 1e250
| inf
> -1e250 * 1e250
| -inf
> import math
> math.inf
| inf
> type(math.inf)
| <class 'float'>
> math.inf / math.inf
| nan
> type(math.nan)
| <class 'float'>
> math.nan == math.nan
| false
> float('inf') - float('inf')
| nan
```

Operations on bool

- The operations `and`, `or`, and `not` behave as expected when the arguments are `False`/`True`.
- The three operators also accept other types, where the following values are considered *false*:

    `False, None, 0, 0.0, "", [],…`

  (see The Python Standard Library > 4.1. True Value Testing for more *false* values)
- *Short-circuit evaluation*: The rightmost argument of `and` and `or` is only evaluated if the result cannot be determined from the leftmost argument alone. The result is either the leftmost or rightmost argument (see truth tables), i.e. the result is not necessarily `False`/`True`.

    `True or 7/0` is completely valid since 7/0 will never be evaluated (which otherwise would throw a ZeroDivisionError exception)

x	x or y
false	y
otherwise	x

x	x and y
false	x
otherwise	y

x	not x
false	True
otherwise	False

Questions – What is `"abc" and 42` ?

a) `False`

b) `True`

c) `"abc"`

d) `42`

e) TypeError

f) Don’t know

Comparison operators (e.g. int, float, str)

==	test if two objects are equal, returns bool not to be confused with the assignment operator (=)
!=	not equal
>	
>=	
<	
<=	

Python shell

```
> 3 == 7
| False
> 3 == 3.0
| True
> "-1" != -1
| True
> "abc" == "ab" + "c"
| True
> 2 <= 5
| True
> -5 > 5
| False
> 1 == 1.0
| True
> 1 == 1.0000000000000001
| True
> 1 == 1.000000000000001
| False
```

Chained comparisons

- A recurring condition is often

```
x < y and y < z
```

- If `y` is a more complex expression, we would like to avoid computing `y` twice, i.e. we often would write

```
tmp = complex expression
x < tmp and tmp < z
```

- In Python this can be written as a chained comparisons (which is shorthand for the above)

```
x < y < z
```

- Note: Chained comparisons do not exist in C, C++, Java, ...

Questions – What is `1 < 0 < 6/0` ?

a) `True`
b) `False`
c) `0`
d) `1`
e) `6`
f) ZeroDivisionError
g) Don’t know

Binary numbers and operations

- Binary number = integer written in base 2: $101010_2 = 42_{10}$
- Python constant prefix 0b: `0b101010` → `42`
- `bin(x)` converts integer to string: `bin(49)` → `"0b110001"`
- `int(x,2)` converts binary string value to integer: `int("0b110001",2)` → `49`
- Bitwise operations
  - `|` Bitwise OR
  - `&` Bitwise AND
  - `~` Bitwise NOT (`~ x` equals to `-x - 1`)
  - `^` Bitwise XOR
- Example: `bin(0b1010 | 0b1100)` → `"0b1110"`
- Hexadecimal = base 16, Python prefix 0x: `0x30` → `48`, `0xA0` → `160`, `0xFF` → `255`
- `<<` and `>>` integer bit shifting left and right, e.g. `12 >> 2` → `3`, and `1 << 4` → `16`

Operations on strings

- `len(`*str*`)` returns length of *str*
- *str*`[`*index*`]` returns *index*+1'th symbol in *str*
- str_1 + str_2 returns concatenation of two strings
- *int* `*` *str* concatenates *str* with itself *int* times
- Formatting: % operator or .format() function
  old Python 2 way — since Python 3.0
  or formatted string literals (f-strings) with prefix
  since Python 3.6
  letter `f` and Python expressions in `{}`

  (see pyformat.info for an introduction)

From "What's New In Python 3.0", 2009: *A new system for built-in string formatting operations replaces the % string formatting operator. (However, the % operator is still supported; it will be deprecated in Python 3.1 and removed from the language at some later time.) Read PEP 3101 for the full scoop.*

Python shell

```
> len("abcde")
| 5
> "abcde"[2]
| 'c'
> x = 2; y = 3
> "x = %s, y = %s" % (x, y)
| 'x = 2, y = 3'
> "x = {}, y = {}".format(x,y)
| 'x = 2, y = 3'
> f'x + y = {x + y}'
| 'x + y = 5'
> f'{x + y = }'   # >= Python 3.8
| 'x + y = 5'
> f'{x} / {y} = {x / y:.3}'
| '2 / 3 = 0.667'
> "abc" + "def"
| 'abcdef'
> 3 * "x--"
| 'x--x--x--'
> 0 * "abc"
| ''
```

% formatting (inherited from C's sprint() function) was supposed to be on the way out - but is still going strong in Python 3.8

… more string functions

- *str* [−*index*] returns the symbol i positions from the right, the rightmost *str* [−1]
- *str* [*from* : *to*] substring starting at index *from* and ending at index *to*-1
- *str* [*from* : −*to*] substring starting at *form* and last at index `len` (*str*) − *to* −1
- *str* [*from* : *to* : *step*] only take every *step*'th symbol in *str*[*from*:*to*]
  - *from* or/and *to* can be omitted and defaults to the beginning/end of string
- `chr` (*x*) returns a string of length 1 containing the *x*'th Unicode character
- `ord` (*str*) for a string of length 1, returns the Unicode number of the symbol
- *str*.`lower` ( ) returns string in lower case
- *str* . `split` ( ) split string into list of words, e.g.

```
"we love python".split() = ['we', 'love', 'python']
```

docs.python.org/3/library/string.html

Questions – What is `s[2:42:3]`?

```
      0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44
s = 'abwwdexy___lwtopavghevt_xypxxyattx_hxwoadnxxx'
```

a) `'wwdexy___lwtopavghevt_xypxxyattx_hxwoadn'`

b) `'we_love_python'`

c) `'we_love_java'`

d) Don’t know

Strings are immutable

- Strings are non-scalar, i.e. for `s = "abcdef"`, `s[3]` will return `"d"`
- Strings are immutable and cannot be changed once created. I.e. the following natural update is not possible (but is e.g. allowed in C)

```
s[3] = "x"
```

- To replace the `"d"` with `"x"` in `s`, instead do the following update

```
s = s[:3] + "x" + s[4:]
```

Operators
Precedence rules & Associativity

Example: * has higher precedence than +

2 + 3 * 4 ≡ 2 + (3 * 4) → 14 and (2 + 3) * 4 → 20

All operators in same group are evaluated left-to-right

2 + 3 - 4 - 5 ≡ ((2 + 3) - 4) - 5 → -4

except for **, that is evaluated right-to-left

2**2**3 ≡ 2**(2**3) → 256

Rule: Use parenthesis whenever in doubt of precedence!

docs.python.org/3/reference/expressions.html

Precedence (low to high)
`or`
`and`
`not x`
`in` `not in` `is` `is not` `==` `<` `<=` `!=` `>` `>=`
`|`
`^`
`&`
`<<` `>>`
`+` `-`
`*` `@` `/` `//` `%`
`+x` `-x` `~x`
`**`

Long expressions

- Long expressions can be broken over several lines by putting parenthesis around it
- The PEP8 guidelines recommend to limit **all** lines to a maximum of 79 characters

Python shell

```
> (1
    + 2 +
          3)
| 6
```

https://www.python.org/dev/peps/pep-0008/#maximum-line-length

+= and friends

- Recurring statement is

```
x = x + value
```

- In Python (and many other languages) this can be written as

```
x += value
```

- This also applies to other operators like

```
+=   -=   *=   /=   //=   **=
|=   &=   ^=   <<=   >>=
```

Python shell

```
> x = 5
> x *= 3
> x
| 15
> a = 'abc'
> a *= 3
> a
| 'abcabcabc'
```

:= assignment expressions (the “Walrus Operator”)

New in Python 3.8

- Syntax

  `name := expression`

- Evaluates to the value of `expression`, with the side effect of assigning result to `name`
- Useful for naming intermediate results/repeating subexpressions for later reusage
- See PEP 572 for further details and restrictions of usage

Python shell

```
> (x := 2 * 3) + 2 * x
| 18
> print(1 + (x := 2 * 3), 2 + x)
| 7 8
> x := 7
| SyntaxError
> (x := 7)   # valid, but not recommended
> while line := input():
      print(line.upper())
> abc
| ABC
```

- In some languages, e.g. Java, C and C++, “`=`“ also plays the role of “`:=`“, implying “`if (x=y)`” and “`if (x==y)`” mean quite different things (common typo)