Berkeley MSSE
MOLECULAR SCIENCE & SOFTWARE ENGINEERING

Lecture 02:

Python Programming Fundamentals

Markus Hohle

University California, Berkeley

Python for Molecular Sciences

MSSE 272, 3 Units

Outline

- Python in a Nutshell
- Bits & Bytes and Encoding
- Basic Types
- Modules
- The `math` Library

Structure of this Course

- 1st Watch the Lecture Videos
- 2nd Watch the Supporting Material for each chapter
- 3rd Solve the Lecture Exercises
- 4th Solve the Problem Sets

Outline

- Python in a Nutshell

- Bits & Bytes and Encoding

- Basic Types

- Modules

- The `math` Library

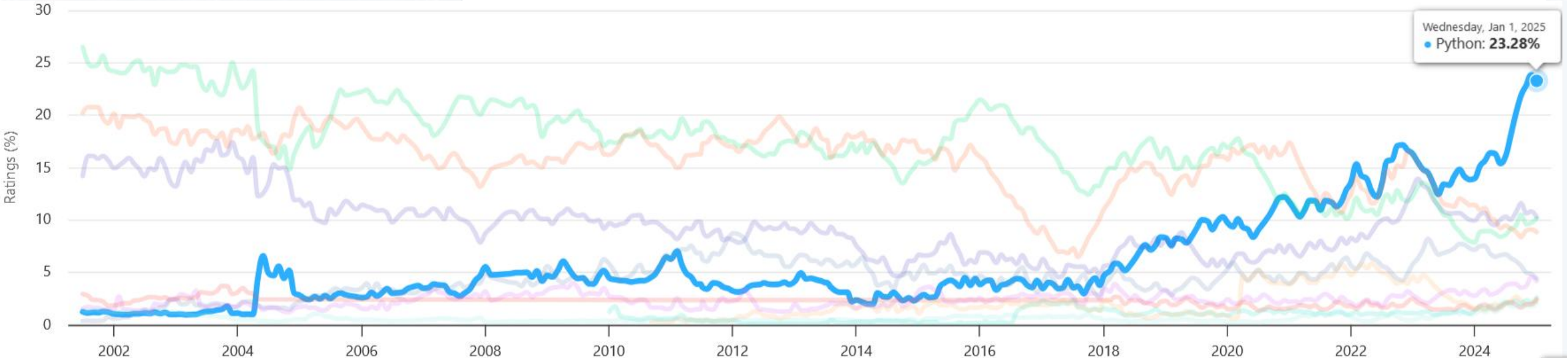
created 1990, Guido van Rossum

named after **“Monty Python”**, not the serpent

idea: flexible, simple and compact syntax, extendable

TIOBE index Jan 2025

Jan 2025	Jan 2024	Change	Programming Language	Ratings	Change
1	1		Python		23.28%
2	3				
3	4	▲	Java	10.15%	+2.28%
4	2	▼	C	8.86%	-2.59%
5	5		C#	4.45%	-2.71%
6	6		JavaScript	4.20%	+1.43%
7	11	⇈	Go	2.61%	+1.24%
8	9	▲	SQL	2.41%	+0.95%



Programming Languages: translate human instructions to a form understandable by a computer

procedural: functions/ routines that call each other
(Fortran, ALGOL, COBOL, BASIC, Pascal, C)

object oriented (OOP): creating objects/types of different properties (see later)
C++, Fortran 2003, Java, MATLAB, **Python**, Ruby, …

the *"style"* of programming

compiled language: close to the resulting machine code, **fast**
Fortran, C, C++, Java, Cobol, Pascal

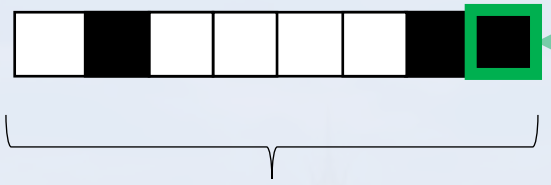
interpreted language: an interpreter translates between source code and machine code. **Slower, but simpler syntax**
Perl, Raku, **Python**, MATLAB

how a programming language *"talks"* to your CPU or GPU

Outline

- Python in a Nutshell
- **Bits & Bytes and Encoding**
- Basic Types
- Modules
- The `math` Library

We need only **two** states: **on** and **off**



$n = 8$

smallest memory cell: **8bits = 1byte**

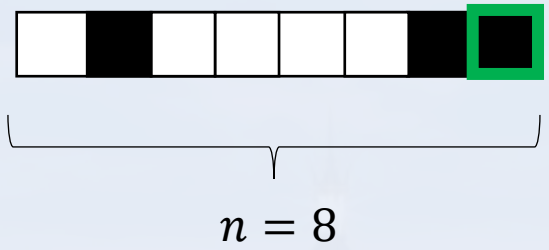
one bit (**bi**nary dig**it**)

yes/no
on/off
up/down

How many different states can I create with 8 switches? 2 x 2 x 2 … = 2^n

8 bit :	2^8 =	256
16 bit:	2^{16} =	65.536
32 bit:	2^{32} =	4.294.967.296
64 bit:	2^{64} =	18.446.744.073.709.551.616

We need only **two** states: **on** and **off**



smallest memory cell: **8bits = 1byte**

8 bit :	2^8 =	256
16 bit:	2^{16} =	65.536
32 bit:	2^{32} =	4.294.967.296
64 bit:	2^{64} =	18.446.744.073.709.551.616

0	00000000
1	00000001
2	00000010
3	00000011
4	00000100
5	00000101
6	00000110
7	00000111
8	00001000
9	00001001
10	00001010
11	00001011
12	00001100
13	00001101
14	00001110
15	00001111
16	00010000
…	

We need only **two** states: **on** and **off**

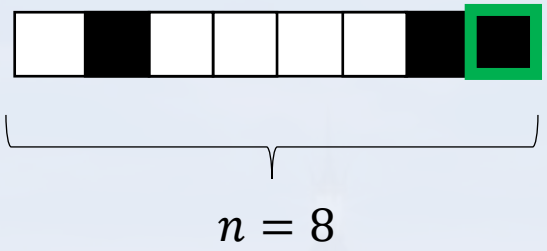
$$0 = 0 \cdot 2^7 + 0 \cdot 2^6 + 0 \cdot 2^5 + 0 \cdot 2^4 + 0 \cdot 2^3 + 0 \cdot 2^2 + 0 \cdot 2^1 + 0 \cdot 2^0$$
$$1 = 0 \cdot 2^7 + 0 \cdot 2^6 + 0 \cdot 2^5 + 0 \cdot 2^4 + 0 \cdot 2^3 + 0 \cdot 2^2 + 0 \cdot 2^1 + 1 \cdot 2^0$$
$$2 = 0 \cdot 2^7 + 0 \cdot 2^6 + 0 \cdot 2^5 + 0 \cdot 2^4 + 0 \cdot 2^3 + 0 \cdot 2^2 + 1 \cdot 2^1 + 0 \cdot 2^0$$

$$7 = 0 \cdot 2^7 + 0 \cdot 2^6 + 0 \cdot 2^5 + 0 \cdot 2^4 + 0 \cdot 2^3 + 1 \cdot 2^2 + 1 \cdot 2^1 + 1 \cdot 2^0$$

0	00000000
1	00000001
2	00000010
3	00000011
4	00000100
5	00000101
6	00000110
7	00000111
8	00001000
9	00001001
10	00001010
11	00001011
12	00001100
13	00001101
14	00001110
15	00001111
16	00010000
…	

8 bit :	2^8 =	256
16 bit:	2^{16} =	65.536
32 bit:	2^{32} =	4.294.967.296
64 bit:	2^{64} =	18.446.744.073.709.551.616

We need only **two** states: **on** and **off**



$n = 8$

smallest memory cell: **8bits = 1byte**

8 bit :	2^8 =	256
16 bit:	2^{16} =	65.536
32 bit:	2^{32} =	4.294.967.296
64 bit:	2^{64} =	18.446.744.073.709.551.616

8 bits	= 1 byte (B)
1 kB	= 1024 B
1 MB	= 1024 kB etc

accuracy of numerical operations:

```
import sys
sys.float_info.epsilon
```

```
2.220446049250313e-16
```

machine epsilon

Outline

- Python in a Nutshell
- Bits & Bytes and Encoding
- **Basic Types**
- Modules
- The `math` Library

(data) types: data values, with specific **set of possible values** and set of **allowed operations**

(data) types: data values, with specific **set of possible values** and set of **allowed operations**

```
my_string = 'this is a string'
```

A variable called `my_string`, that contains the *string* `'this is a string'`

```
type(my_string)
str
```

The `type` of `my_string` is `str` (= “string”)

```
my_number = 5
```

A variable called `my_number`, that contains an *integer* `5`

```
type(my_number)
int
```

The `type` of `my_number` is `int` (= “integer”)

(data) types: data values, with specific **set of possible values** and set of **allowed operations**

```
my_string = 'this is a string'
```

```
my_string = '5'

type(my_string)
str
```

```
my_string = 5

type(my_string)
int
```

```
my_number = 5
```

```
my_number = Ab
```

```
In [49]: my_number = Ab
Traceback (most recent call last):

  Cell In[49], line 1
    my_number = Ab

NameError: name 'Ab' is not defined
```

```
my_number = 'Ab'

type(my_number)
str
```

(data) types: data values, with specific **set of possible values** and set of **allowed operations**

```
my_string = 'this is a string'

2*my_string
my_string + my_string

'this is a stringthis is a string'


my_number = 5
```

```
my_string/3
```

```
Traceback (most recent call last):

  Cell In[56], line 1
    my_string/3

TypeError: unsupported operand type(s) for /: 'str' and 'int'
```

type error: operation is invalid for this specific type!

(data) types: data values, with specific **set of possible values** and set of **allowed operations**

```
my_string = 'this is a string'

2*my_string
my_string + my_string
```

```
'this is a stringthis is a string'
```

```
my_string/3
```

```
Traceback (most recent call last):

  Cell In[56], line 1
    my_string/3

TypeError: unsupported operand type(s) for /: 'str' and 'int'
```

```
my_number = 5

2*my_number
my_number + my_number
```

```
10
```

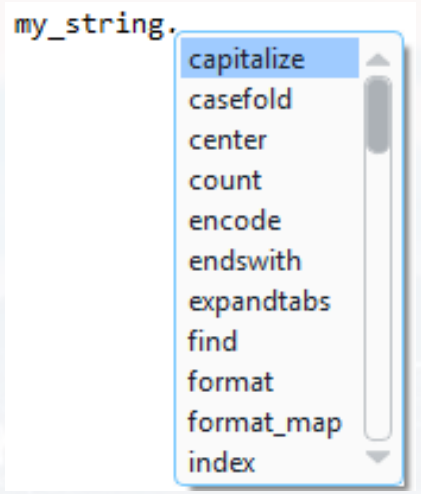
(data) types: data values, with specific **set of possible values** and set of **allowed operations**

```
my_string = 'this is a string'

2*my_string
my_string + my_string

'this is a stringthis is a string'


my_number = 5

2*my_number
my_number + my_number

10
```

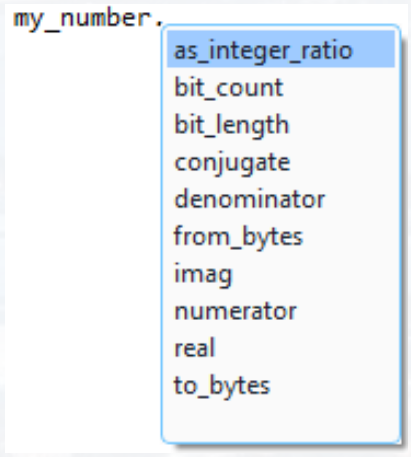
The fact that we can use the same operator (here +) for different types is called **operator overload**

(data) types: data values, with specific **set of possible values** and set of **allowed operations**

```
my_string = 'this is a string'
```




```
my_number = 5
```



The zoo of types

	numeric: `int`, `float`, `complex`	`5`, `5.55`, `(5+5j)`
iteratable	strings: `str`	`'this is a string'`, `"this is a string"`
iteratable	sequence: `list`, `tuple`, `range`	`my_tuple = (3, 'a', [2,3,4,5])` `range(10)`
iteratable, mutable		`my_list = [1, 2, 'a']`
iteratable, mutable	mapping: `dict`	`my_dict = {1: 'a', 2: 'b'}`
iteratable, mutable	mapping: `set`	`my_set = {1, 2, 'a'}`
	boolean:	`True` `False`
	none type:	`None`
	callable: functions, methods, classes	`def`, `class`, `map`, `lambda`
	modules:	`from my_module import my_method as my_alias`

The zoo of types

This lecture:

strings: `str`	`'this is a string', "this is a string"`
numeric: `int, float, complex`	`5, 5.55, (5+5j)`
boolean:	`True False`
modules:	`from my_module import my_method as my_alias`

when to use:

labels and titles of plots
paths and file names
error messages

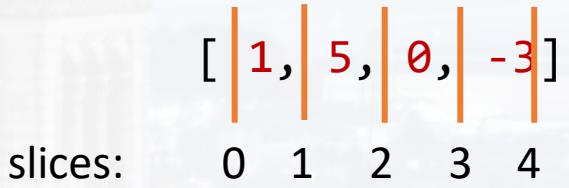
strings: `str`
numeric: `int`, `float`, `complex`
boolean
modules

```
string1 = 'Hello Students'
string2 = ', how are you'

string12 = string1 + string2
'Hello Students, how are you'

S = 'abc'

3*S
'abcabcabc'
```

concatenating is incredibly easy!

```
string12[2:6]
```

slicing



strings: `str`
numeric: `int`, `float`, `complex`
boolean
modules

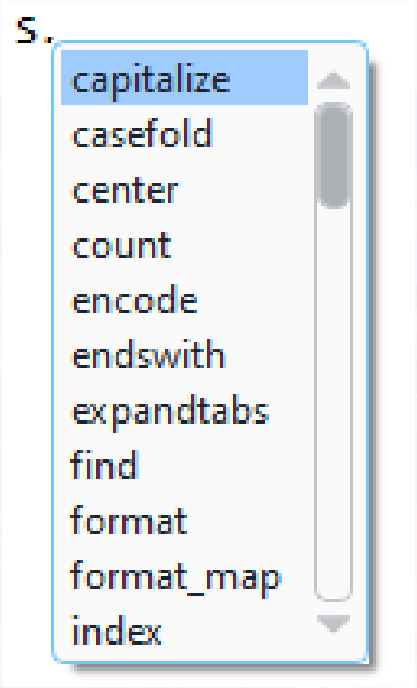
```
string12[2:6]
```

```
[ 1, 5, 0, -3]
slices:   0  1  2  3  4
```

```
[1, 5, 0, -3]
index:   0  1  2   3

index:   -4 -3 -2  -1

string12[-1]

string12[1:]

string12[:-1]
```

indexing

strings:	`str`
numeric:	`int`, `float`, `complex`
boolean	
modules	



try some of the functions like

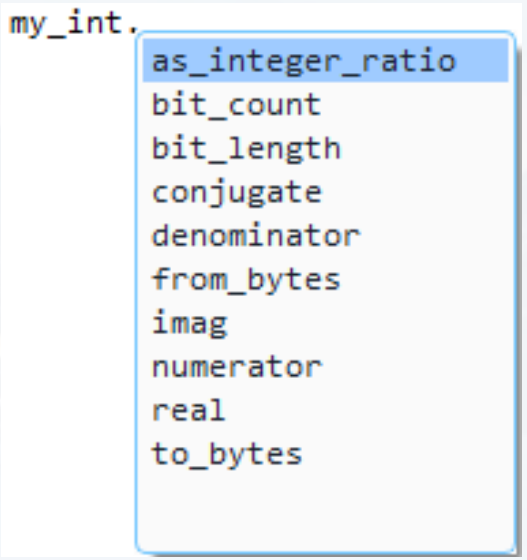
```
S.count()
S.find()
```

strings: str
numeric: int, float, complex
boolean
modules

```
my_int = 5

type(my_int)
Int

my_float = 5.0

type(my_float)
float

type(10/5)
float
```

check out:

```
5**2

36**0.5

5//3

6//3

5%3

6%3

type(str(6))
```



```
c1 = (-1)**0.5

print(c1)
(6.123233995736766e-17+1j)
```

machine epsilon: 2.2e-16

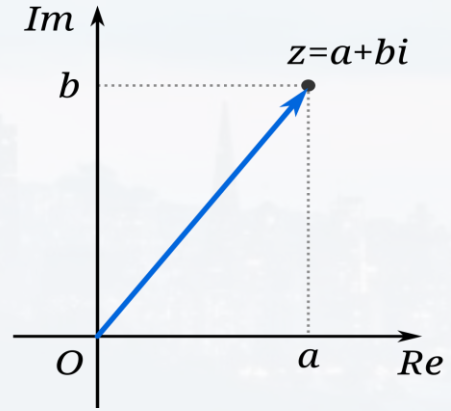
complex unit ***i***, in Python: ***j***

strings: `str`

numeric: **`int`, `float`, `complex`**

boolean

modules




```
c2 = complex(real = 5, imag = -4)

print(c2)
(5-4j)
```

check out:

```
c2**2

c2**0.25

c2.imag

c2.real
```

when to use:

checking if statement is true or false
checking if variable has a value/ is of a type
catching up error messages

strings: `str`
numeric: `int`, `float`, `complex`
boolean
modules

```
my_number = 5
```

assigning a value to a variable

```
my_number == 5
True

my_number > 5
False

my_number != 5
False
```

logical operations return ***boolean*** statement

value of a variable equal to another value

```
type(my_number > 5)
bool
```

when to use:

checking if statement is true or false
checking if variable has a value/ is of a type
catching up error messages

strings: `str`
numeric: `int, float, complex`
boolean
modules

```
my_number = 5
string    = 'Python is great!'
```

```
isinstance(my_number, int)
True

isinstance(my_number, float)
False

isinstance(string, str)
True

isinstance(True, bool)
True
```

checking if variable has a value
(What do you expect the output is?)

```
bool(None)

bool(0)

bool("")

bool(())

bool([])

bool({})

bool(False)

bool(True)
```

when to use:

checking if statement is true or false
checking if variable has a value/ is of a type
catching up error messages

strings:	`str`
numeric:	`int`, `float`, `complex`
boolean	
modules	

Let's run the following code together and try to understand, what it does:

```
def CheckTimes2(var = None):
    if not bool(var):
        print('You need an input!')
    elif isinstance(var, float):
        return 2*var
    elif isinstance(var, int):
        return 2*var
    elif isinstance(var, str):
        return 2*var
    else:
        print('not possible to multiply ' + str(type(var)))
```

Outline

- Python in a Nutshell
- Bits & Bytes and Encoding
- Basic Types
- **Modules**
- The `math` Library

```
from my_module import my_method as my_alias
```

strings: `str`
numeric: `int`, `float`, `complex`
boolean
modules

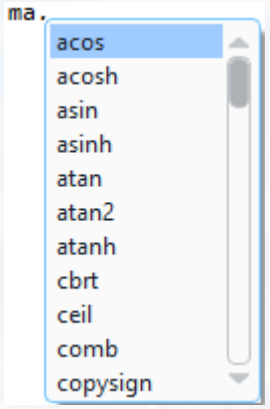
1) **reading files (.xlsx, .xls, .csv, .txt, …)** — `pandas` (standard), `dask`, `polars`
2) **plotting** — `matplotlib`, `seaborn`
3) **numerical methods** — `math`, `numpy`, `scipy`
4) **machine learning** — `scikitlearn`
5) **ANN/AI/DeepLearning**

```
from my_module import my_method as my_alias
```

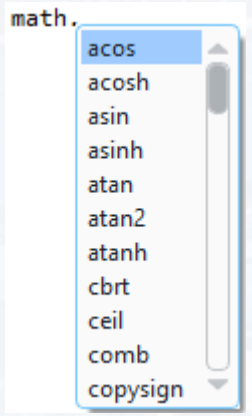
strings: `str`
numeric: `int`, `float`, `complex`
boolean
modules

alias

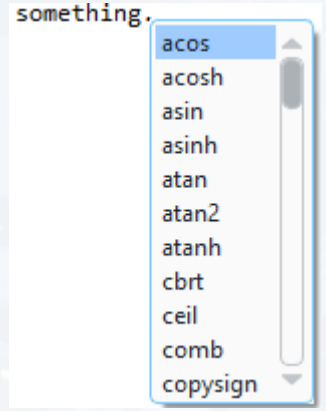
```
import math as ma
```




```
import math as math
```




```
import math as something
```



```
from my_module import my_method as my_alias
```

strings: `str`
numeric: `int`, `float`, `complex`
boolean
modules

importing specific tools

```
from math import cos as cosine
```

```
cosine(3.14159)
-0.9999999999964793
```

```
sin(3.14159)
```

```
Traceback (most recent call last):

  Cell In[11], line 1
    sin(3.14159)

NameError: name 'sin' is not defined
```

the method `sin` has not been imported yet

```
from my_module import my_method as my_alias
```

strings: `str`
numeric: `int`, `float`, `complex`
boolean
modules

importing specific tools

```
from math import cos, sin
```

```
cos(3.14159)
-0.9999999999964793
```

```
sin(3.14159)
2.65358979335273e-06
```

```
from my_module import my_method as my_alias
```

strings: str
numeric: int, float, complex
boolean
modules

importing specific tools

```
from math import cos, sin

cos(3.14159)
-0.9999999999964793

sin(3.14159)
2.65358979335273e-06
```

importing all tools at once

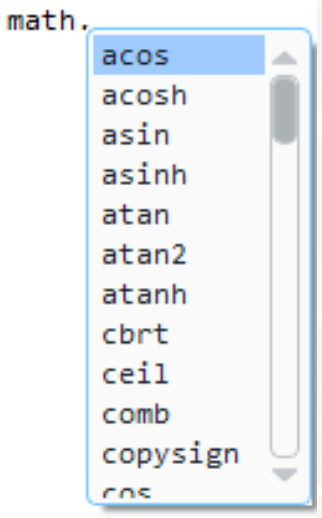
```
from math import *

cos(3.14159)
-0.9999999999964793

sin(3.14159)
2.65358979335273e-06

tan(3.14159)
-2.653589793620727e-06
```

Outline

- Python in a Nutshell
- Bits & Bytes and Encoding
- Basic Types
- Modules
- **The `math` Library**

strings: `str`
numeric: `int, float, complex`
boolean
modules

```
import math as math
```




```
math.acos(0) * 2
3.141592653589793
```

importing a module as alias → calling a method via `my_alias.method`, i.e.

```
math.acos
math.ceil
```

or

```
from math import *
```

```
acos(0) * 2
3.141592653589793
```

importing all methods (*) from a module

```
acos
ceil
```

strings: str
numeric: int, float, complex
boolean
modules

```
import math as math
```

math contains a vast set of mathematical operations

```
dir(math)
```

```
['acos',
'acosh',
'asin',
'asinh',
'atan',
'atan2',
'atanh',
'cbrt',
'ceil',
'comb',
'copysign',
'cos',
'cosh',
'degrees',
'dist',
'e',
'erf',
'erfc',
'exp',
'exp2',
'expm1',
'fabs',
'factorial',
'floor',
'fmod',
'frexp',
'fsum',
'gamma',
'gcd',
'hypot',
'inf',
'isclose',
'isfinite',
'isinf',
'isnan',
'isqrt',
'lcm',
'ldexp',
'lgamma',
'log',
'log10',
'log1p',
'log2',
'modf',
'nan',
'nextafter',
'perm',
'pi',
'pow',
'prod',
'radians',
'remainder',
'sin',
'sinh',
'sqrt',
'tan',
'tanh',
'tau',
'trunc',
'ulp']
```

each method includes a documentation

```
math.log(
        log(x, [base=math.e])

        Return the logarithm of x to the given base.

        If the base not specified, returns the natural logarithm
        (base e) of x.
```

strings: `str`
numeric: `int`, `float`, `complex`
boolean
modules

```
import math as math
```

Lecture Exercise!

Calculate the values for the following equations using `math`

`see 02_Lecture_Exercise.ipynb`

$log_4(32)$ $\cos(60°)$ $\sqrt{2+5i}$

$e^{i\pi}$ $\frac{e^5}{6!}$ $\binom{10}{5}$

Thank you very much for your attention!