Berkeley
UNIVERSITY OF CALIFORNIA

M. Hohle:

Physics 77: Introduction to Computational Techniques in Physics

Introduction to Computational Techniques in Physics: syllabus

Week	Date	Topic
1	June 12th	Programming Environment & UIs for Python, Programming Fundamentals
2	***June 19th***	**Basic Types in Python**
3	June 26th	Parsing, Data Processing and File I/O, Visualization
4	July 3rd	Functions, Map & Lambda
5	July 10th	Random Numbers & Probability Distributions, Interpreting Measurements
6	July 17th	Numerical Integration and Differentiation
7	July 24th	Root finding, Interpolation
8	July 31st	Systems of Linear Equations, Ordinary Differential Equations (ODEs)
9	Aug 7th	Stability of ODEs, Examples
10	Aug 14th	Final Project Presentations

Basic Types
Modules
The `math` Library

The zoo of types

	numeric: `int`, `float`, `complex`	`5`, `5.55`, `(5+5j)`
iterable	strings: `str`	`'this is a string'`, `"this is a string"`
iterable	sequence: `list`, `tuple`, `range`	`my_tuple = (3, 'a', [2,3,4,5])` `range(10)`
iterable, mutable		`my_list = [1, 2, 'a']`
iterable, mutable	mapping: `dict`	`my_dict = {1: 'a', 2: 'b'}`
iterable, 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`

(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_number = 5

2*my_number
my_number + my_number
```

```
10
```

```
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'
```

(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_string.
```

- capitalize
- casefold
- center
- count
- encode
- endswith
- expandtabs
- find
- format
- format_map
- index

```
my_number = 5
```

```
my_number.
```

- as_integer_ratio
- bit_count
- bit_length
- conjugate
- denominator
- from_bytes
- imag
- numerator
- real
- to_bytes

when to use:

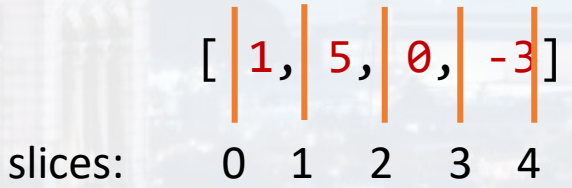
labels and titles of plots
paths and file names
error messages

```
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



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

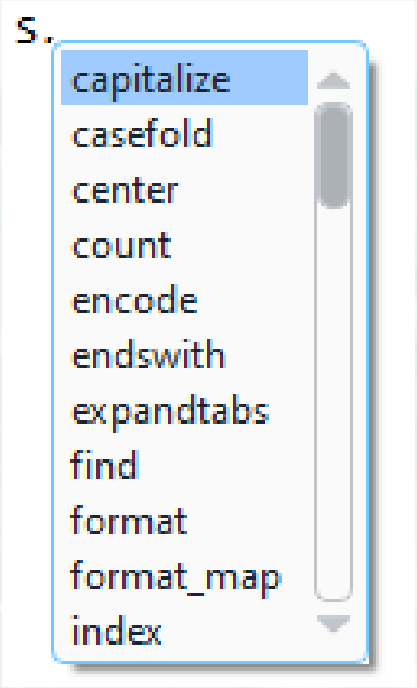
```
[ 1, 5, 0, -3]
slices:  0  1  2  3  4
```

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

```
index:   -4  -3  -2  -1
```

indexing

```
string12[-1]

string12[1:]

string12[:-1]
```



try some of the functions like

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

Basic Types
int
Modules
The math Library

```
my_int = 5
```

```
type(my_int)
Int
```

```
my_float = 5.0
```

```
type(my_float)
float
```

```
type(10/5)
float
```

check out:

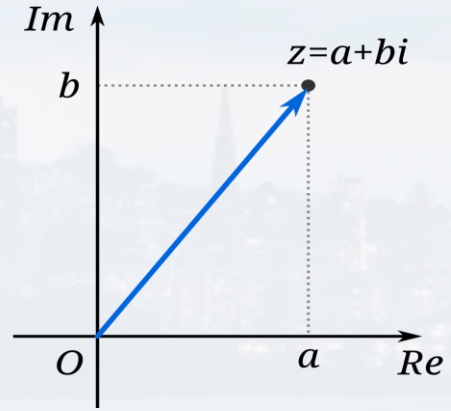
```
5**2
36**0.5
5//3
6//3
5%3
6%3
type(str(6))
```

```
my_int.
    as_integer_ratio
    bit_count
    bit_length
    conjugate
    denominator
    from_bytes
    imag
    numerator
    real
    to_bytes
```

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

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

machine epsilon: 2.2e-16

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




```
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

```
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

```
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

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)))
```

The zoo of types

numeric: `int, float, complex` — `5, 5.55, (5+5j)`

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)
```

mutable

```
my_list  = [1, 2, 'a']
```

mapping: `dict`

```
my_dict  = {1: 'a', 2: 'b'}
```

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
```

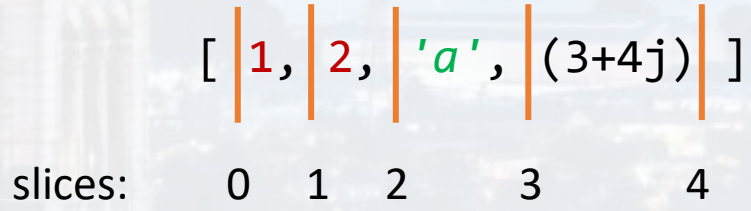
when to use:

"default" type in Python
storing variables of different types in one object
error messages

```
L1 = [1, 2, 'a', complex(3,4)]

type(L1)
list

print(2*L1)
[1, 2, 'a', (3+4j), 1, 2, 'a', (3+4j)]

print(L1 + L1)
[1, 2, 'a', (3+4j), 1, 2, 'a', (3+4j)]

L1[2:4]
```

recall: operator overload
(see *'strings'*)

slicing is identical too



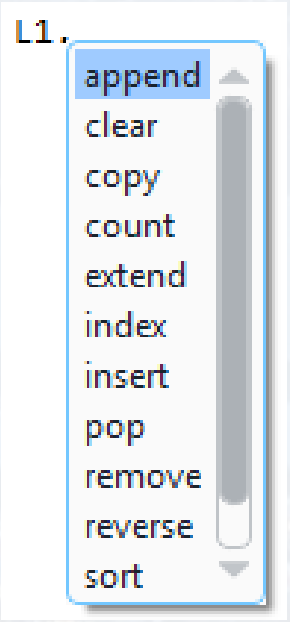
```
L1 = [1, 2, 'a', complex(3,4)]

L2 = L1


print(L1[2], L2[2])
a a

L1[2] = 'b'


print(L1[2], L2[2])
b b


L2[2] = 'a'


print(L1[2], L2[2])
a a
```

Even though we only changed L1, it affected L2 too!

Lists are **mutable**!

```
L1 = [1, 2, 'a', complex(3,4)]

L2 = L1.copy()


print(L1[2], L2[2])
a a

L1[2] = 'b'

print(L1[2], L2[2])
b a

L2[2] = 'c'

print(L1[2], L2[2])
b c
```

not passing mutability to L2



when to use:

keyword arguments (**kwargs) in functions
making code more compact (vs control structures)

```
D1 = dict(      name   = "John Doe",
                DoB    = 'March 2nd, 2005',
                grades = ["A+", 'B', 'A-'],
                year   = 2024               )
```

key **value**

creating a dictionary using the **constructor** `dict`

```
D2 =      {     'name':       "John Doe",
                'DoB':        'March 2nd, 2005',
                'grades':     ["A+", 'B', 'A-'],
                'year':       2024                }
```

creating a dictionary using { }

```
type(D2)
dict

print(D2)
{'name': 'John Doe', 'DoB': 'March 2nd, 2005', 'grades': ['A+', 'B', 'A-'],
'year': 2024}



D2['name']
'John Doe'



D2.keys()
dict_keys(['name', 'DoB', 'grades', 'year'])

D2.values()
dict_values(['John Doe', 'March 2nd, 2005', ['A+', 'B', 'A-'], 2024])
```

```
print(D2)
{'name': 'John Doe', 'DoB': 'March 2nd, 2005', 'grades': ['A+', 'B', 'A-'],
'year': 2024}
```

```
D2.update(Address = '134 Street, Home')
```

adding keys

```
D2
{'name':      'John Doe',
 'DoB':       'March 2nd, 2005',
 'grades':    ['A+', 'B', 'A-'],
 'year':      2024,
 'Address':   '134 Street, Home'}
```

```
print(D2)
{'name': 'John Doe', 'DoB': 'March 2nd, 2005', 'grades': ['A+', 'B', 'A-'],
'year': 2024}
```

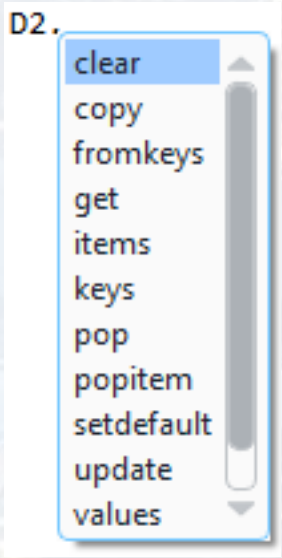
```
D2['name'] = 'Tony Clifton'
```

updating values

```
D2
{'name':      'Tony Clifton',
 'DoB':       'March 2nd, 2005',
 'grades':    ['A+', 'B', 'A-'],
 'year':      2024,
 'Address':   '134 Street, Home'}
```

```
print(D2)
{'name': 'John Doe', 'DoB': 'March 2nd, 2005', 'grades': ['A+', 'B', 'A-'],
'year': 2024}
```

```
D2.pop('DoB')

D2
{'name':        'Tony Clifton',
 'grades':      ['A+', 'B', 'A-'],
 'year':        2024,
 'Address':     '134 Street, Home'}
```



removing keys

dictionaries are **mutable** too!

```
print(D2)
{'name': 'John Doe', 'DoB': 'March 2nd, 2005', 'grades': ['A+', 'B', 'A-'],
'year': 2024}
```

```
D3 = D2

D2['year'] = 2025

D3['year']
2025
```

```
D3 = D2.copy()

D2['year'] = 2024

D3['year']
2025
```

when to use:

making code more compact (vs control structures)
comparing data entries/ removing duplicates

```
S1 = set(('Mike', 'Karen', 'Simon', 1))
S2 = set(['Mike', 'Karen', 'Simon', 1])
```

creating a set using the **constructor** set

```
type(S1)
type(S2)
set
set
```

```
S3 = {'Mike', 'Karen', 'Simon', 1}

type(S3)
set
```

creating a set using { }

when to use:

making code more compact (vs control structures)
comparing data entries/ removing duplicates

Note:

sets are **not subscriptable** → `S1[1]` prompts a type error!

duplicates are not permitted

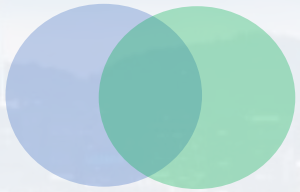
```
S2 = set(['Mike', 'Karen', 'Simon', 1, 1])

print(S2)
{1, 'Simon', 'Mike', 'Karen'}
```

sets are **unchangeable**

sets are **mutable**

```
S1 = set(('a', 'b', 'c'))
S2 = set(('a', 'b', 'd'))
```

```
S1.intersection(S2)
{'a', 'b'}
```



```
S1-S2
Out[34]: {'c'}
```

```
S2-S1
Out[35]: {'d'}
```

when to use:

later: shape of arrays (matrices, data frames)
convenient way to store different objects

```
T1 = tuple([1, 2, 'abc'])

type(T1)
tuple
```

creating a tuple using the **constructor** `tuple`

```
T2 = (1, 2, 'abc')
type(T2)
tuple
```

creating a tuple using `( )`

```
T1 = tuple([1, 2, 'abc'])
T2 = (1, 2, 'abc')
```

```
T1[2]
'abc'
```

indexing & slicing

```
T1[:2]
(1, 2)
```

```
type(T1[:2])
tuple
```

```
(t11, t12, t13) = T1
```

retrieving elements

```
print(t11,t12,t13)
1 2 abc
```

```
T1 = tuple([1, 2, 'abc'])
T2 = (1, 2, 'abc')
```

Note: tuples are **subscriptable** → `T1[1] = 2`

duplicates **are** permitted

tuples are **unchangeable**

summary data collection types

type	constructor	direct construction	mutable	changeable	indexing	slicing	duplicates
list	`list`	`[]`	yes	yes	yes	yes	yes
dictionary	`dict`	`{key: value }`	yes	yes	no	no	yes
set	`set`	`{}`	yes	no	no	no	no
tuple	`tuple`	`()`	no	no	yes	yes	yes

The zoo of types

✓	numeric: `int`, `float`, `complex`	`5`, `5.55`, `(5+5j)`
	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)`
✓		`my_list = [1, 2, 'a']`
	mapping: `dict`	`my_dict = {1: 'a', 2: 'b'}`
	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`

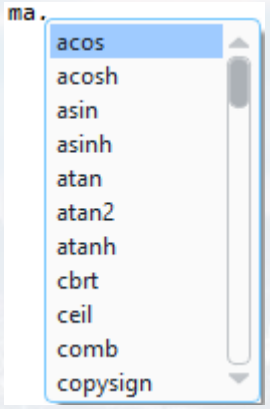
```
from my_module import my_method as my_alias
```

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	

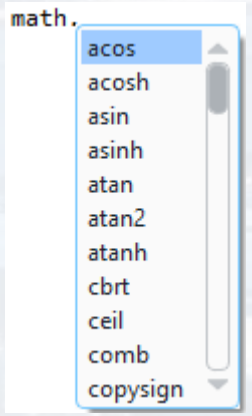
```
from my_module import my_method as my_alias
```

alias

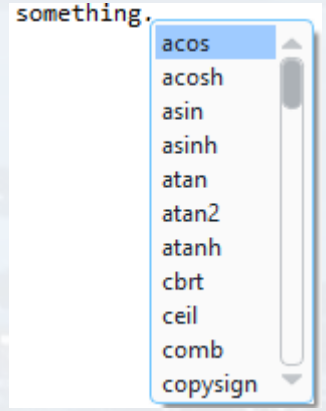
```
import math as ma
```




```
import math as math
```




```
import math as something
```



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

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
```

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
```

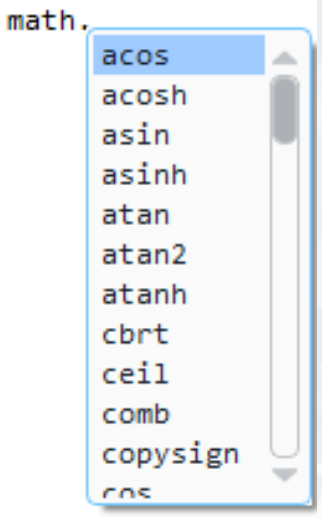
importing specific tools

```
from math import cos, sin

cos(3.14159)
-0.9999999999964793

sin(3.14159)
2.6535897933273e-06
```

importing all tools at once

```
from math import *

cos(3.14159)
-0.9999999999964793

sin(3.14159)
2.6535897933273e-06

tan(3.14159)
-2.6535897933620727e-06
```

```
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
```

```
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.
```

```
import math as math
```

Homework assignment!

Calculate the values for the following equations using `math`

$log_4(32)$ $\cos(60°)$ $\sqrt{2+5i}$

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

Introduction to Computational Techniques in Physics:

Thank you for your attention!