Berkeley MSSE
MOLECULAR SCIENCE & SOFTWARE ENGINEERING

Lecture 03:

Data Structures and Error Handling

Markus Hohle

University California, Berkeley

Python for Molecular Sciences

MSSE 272, 3 Units

Outline

- More Types:
  - lists
  - dictionaries
  - sets
  - tuples

- Error Messages and Debugging

Outline

- More Types:

- lists
- dictionaries
- sets
- tuples

- Error Messages and Debugging

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`

The zoo of types

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'}
```

when to use:

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

list:	`list`
dictionaries:	`dict`
sets	`set`
tuples	`tuple`

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

type(L1)
list
```

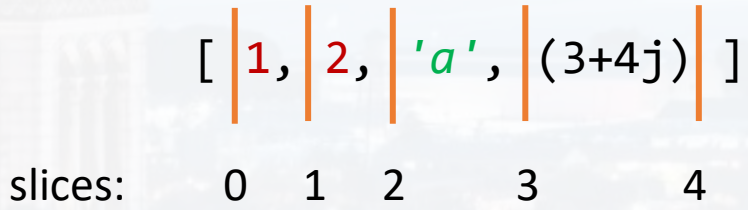
```
print(2*L1)
[1, 2, 'a', (3+4j), 1, 2, 'a', (3+4j)]
```

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

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

```
L1[2:4]
```

slicing is identical too



list:	`list`
dictionaries:	`dict`
sets	`set`
tuples	`tuple`

```
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
```

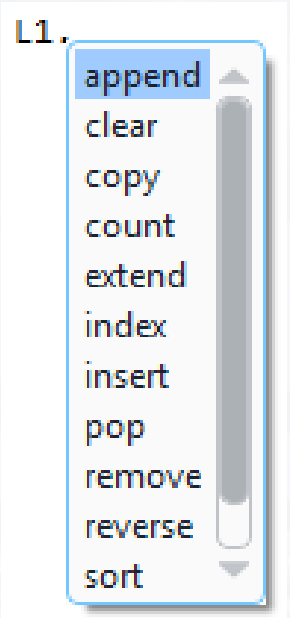
Even though we only changed L1, it affected L2 too!

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
```

list:	**list**
dictionaries:	dict
sets	set
tuples	tuple

not passing mutability to L2



when to use:

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

list:	`list`
dictionaries:	**`dict`**
sets	`set`
tuples	`tuple`

```
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 `{ }`

list:	`list`
dictionaries:	**`dict`**
sets	`set`
tuples	`tuple`

```
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])
```

list:	`list`
dictionaries:	**`dict`**
sets	`set`
tuples	`tuple`

```
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'}
```

list:	`list`
dictionaries:	**`dict`**
sets	`set`
tuples	`tuple`

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

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

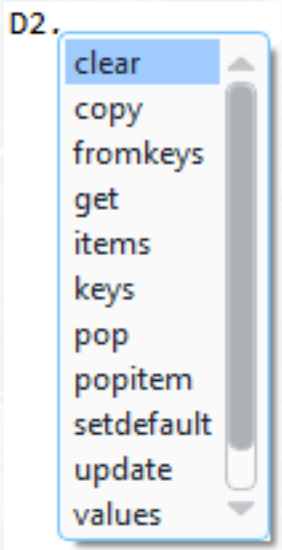
updating values

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

list:	`list`
dictionaries:	**`dict`**
sets	`set`
tuples	`tuple`

```
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!

list:	`list`
dictionaries:	**`dict`**
sets	`set`
tuples	`tuple`

```
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

list:	`list`
dictionaries:	`dict`
sets	**`set`**
tuples	`tuple`

```
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

list:	`list`
dictionaries:	`dict`
sets	**`set`**
tuples	`tuple`

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'}
```

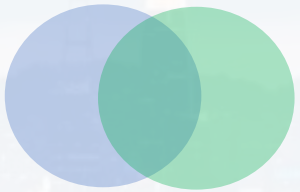
sets are **unchangeable**

sets are **mutable**

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

list:	`list`
dictionaries:	`dict`
sets	**`set`**
tuples	`tuple`

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



Lecture Exercise!

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

`03_Lecture_Exercise.ipynb`

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

when to use:

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

list:	`list`
dictionaries:	`dict`
sets	`set`
tuples	**`tuple`**

```
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')
```

list:	`list`
dictionaries:	`dict`
sets	`set`
tuples	`tuple`

```
T1[2]
'abc'

T1[:2]
(1, 2)

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

indexing & slicing

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

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

retrieving elements

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

list:	`list`
dictionaries:	`dict`
sets	`set`
tuples	**`tuple`**

Note:

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

duplicates **are** permitted

tuples are **unchangeable**

summary data collection types

list: `list`
dictionaries: `dict`
sets `set`
tuples `tuple`

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

Outline

- More Types:
  - lists
  - dictionaries
  - sets
  - tuples

- Error Messages and Debugging

syntax errors

- mostly **typos**
- spelling commands, constructer, variable names etc wrong
- missing symbols where needed (such as `:`)
- missing indentations (loops, functions → see next lectures)

prevention:
- most IDEs (VS Code, Spyder) highlight **syntax errors**
- most IDEs provide **auto complete**

type errors

- **syntax is *in principle* correct**
- performing an operation which is not permitted for current type
- mostly **typos** too

runtime errors

- **syntax is correct**
- function, method or variable is not defined or has not been compiled yet
- mostly **typos** too

logical errors

- valid code syntax, but code does not produce desired results
- **hardest to find and fix**

syntax errors

- **mostly typos**
- spelling commands, constructer, variable names etc wrong
- missing symbols where needed (such as :)
- missing indentations (loops, functions → see next lectures)

prevention:
- most IDEs (VS Code, Spyder) highlight **syntax errors**
- most IDEs provide **auto complete**

For beginners, these are 90 – 95% of all bugs!

type errors

- **syntax is *in principle* correct**
- performing an operation which is not permitted for current type
- **mostly typos too**

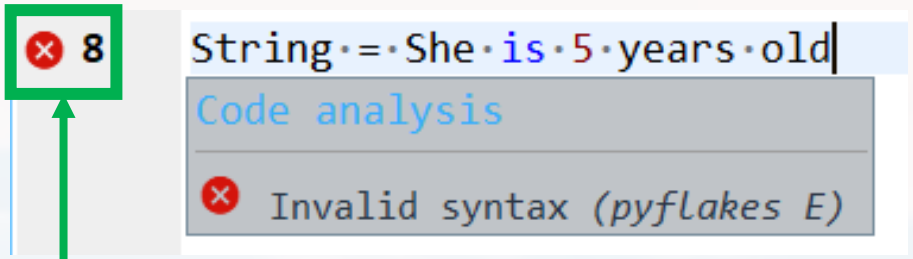
runtime errors

- **syntax is correct**
- function, method or variable is not defined or has not been compiled yet
- **mostly typos too**

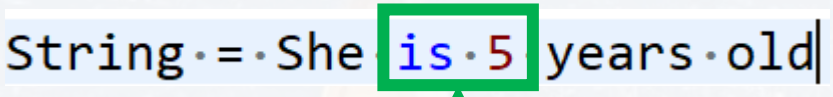
logical errors

- valid code syntax, but code does not produce desired results
- **hardest to find and fix**

syntax errors
type errors
runtime errors
logic errors



most IDEs (VS Code, Spyder) highlight **syntax errors**



most IDEs (VS Code, Spyder) indicate different types by different colors (**syntax highlighting**)

```
In [70]: String = She is 5 years old
  Cell In[70], line 1
    String = She is 5 years old
                     ^
SyntaxError: invalid syntax
```

Variable line is not recognized as string!

syntax errors
type errors
runtime errors
logic errors

Variable line is not recognized as string!

```
String = She is 5 years old
```

turning statement into a string using quotes

```
String = 'She is 5 years old'
```

color indicates recognition as string (**syntax highlighting**) and error message in IDE disappears!

```
 8  String = 'She is 5 years old'
 9  prnit(String)
10
```

Code analysis

Undefined name 'prnit' *(pyflakes E)*

typo raises another error message

syntax errors
type errors
runtime errors
logic errors

```
 8  String = 'She is 5 years old'
 9  prnit(String)
10
```

Code analysis

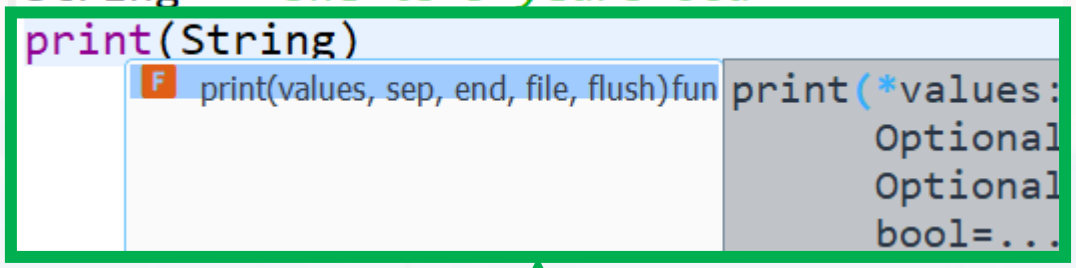
Undefined name 'prnit' (pyflakes E)

```
In [72]: prnit(String)
Traceback (most recent call last):

  Cell In[72], line 1
    prnit(String)

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

because of the typo: `'prnit'` is interpreted as yet undefined function/variable

syntax errors
type errors
runtime errors
logic errors

```
String = 'She is 5 years old'
print(String)
```



now `'print'` is recognized as `print` (color changed!)
and the IDE opens a short description

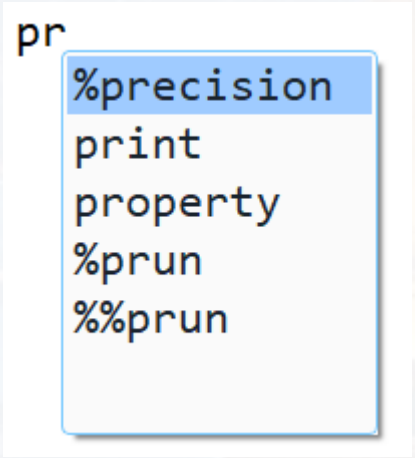
```
In [74]: print(String)
She is 5 years old
```

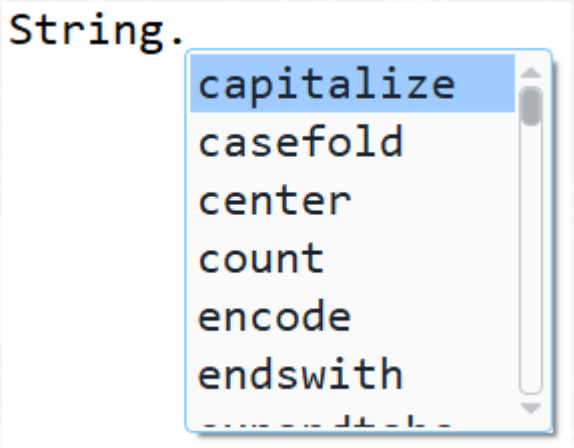
Tip: use an IDE that has **syntax highlighting**!

Tip: use an IDE that has **syntax highlighting**!

syntax errors
type errors
runtime errors
logic errors

Tip: use an IDE that has **auto completion**!





```
String = 'She is 5 years old'
print(String)


String(5)
```

```
In [77]: String(5)
Traceback (most recent call last):

  Cell In[77], line 1
    String(5)

TypeError: 'str' object is not callable
```

syntax errors

type errors

runtime errors

logic errors

Using ( ) instead of [ ] prompts a type error. This operation is not valid for type string!

```
String = 'She is 5 years old'
print(String)
```

```
String[5]
```

```
String[5]
's'
```

```
"5"**2
```

vs

```
5**2
```

```
Traceback (most recent call last):

  Cell In[81], line 1
    "5"**2

TypeError: unsupported operand type(s) for **
or pow(): 'str' and 'int'
```

25

syntax errors

type errors

runtime errors

logic errors

```
String = 'She is 5 years old'
print(String + String1)
```

- syntax errors
- type errors
- **runtime errors**
- logic errors

correct syntax

```
8  String = 'She is 5 years old'
9  print(String + String1)
10 Code analysis
11
12  Undefined name 'String1' (pyflakes E)
```

again, the IDE prompts an error

```
    print(String + String1)

NameError: name 'String1' is not defined
```

```
String = 'She is 5 years old'
print(String)
```

```
In [89]: print(String)
She is 5 years old
```

syntax errors
type errors
runtime errors

logic errors

correct syntax!

```
String = 'She is 5 years old'
print('String')
```

```
In [91]: print('String')
String
```

the code runs smoothly, but does not produce the desired output!

```
70  from draw_umap import draw_umapp
71  Code analysis
72
73  'draw_umap.draw_umapp' imported but unused (pyflakes E)
```

syntax errors
type errors
runtime errors

logic errors

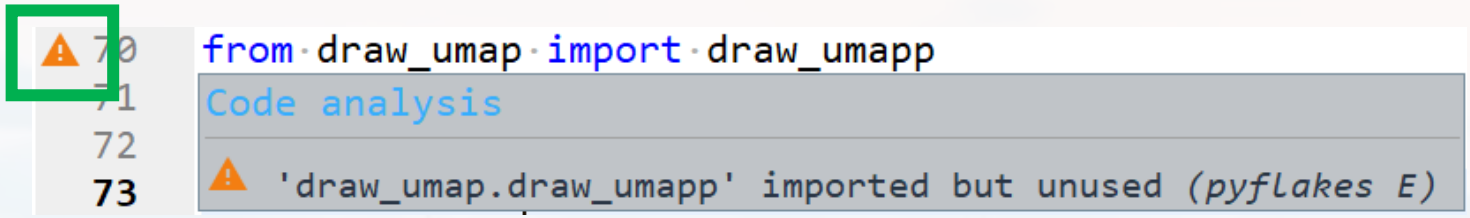
Again, correct syntax!
A typo is causing a warning!
→ imported library will not be used

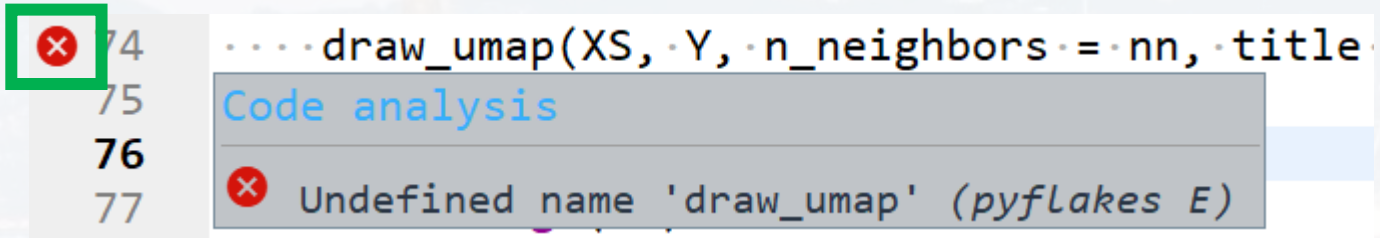
```
74      draw_umap(XS, Y, n_neighbors = nn, title
75  Code analysis
76
77  Undefined name 'draw_umap' (pyflakes E)
```

sometimes, a logic error causes a runtime error later in the code

syntax errors
type errors
runtime errors

logic errors





Highlighting errors or inconsistencies by the IDE while coding, is called **linting**.

If debugging is tricky (i. e. logic errors)

→ Python has an internal **debugger**

```
String = 'She is 5 years old'

breakpoint()

print(String + String1)
```

```
      8 String = 'She is 5 years old'
      9 breakpoint()
---> 10 print(String + String1)
     11
     12


IPdb [1]:
```

runs code to breakpoint

Python debugger

If debugging is tricky (i. e. logic errors)

→ Python has an internal **debugger**

```
String = 'She is 5 years old'

breakpoint()

print(String + String1)
```

```
IPdb [1]: type(String1)
*** NameError: name 'String1' is not defined
```

```
IPdb [1]: type(String)
Out  [1]: <class 'str'>
```

Tip 1: use an IDE that has **syntax highlighting**!

Tip 2: use an IDE that has **auto completion**!

Tip 3: use an IDE that has **linting**!

Tip 4: use `breakpoint()` in order to run the code to that line where it stops producing correct results

→ check `type` of objects!

Thank you very much!