APA: Advanced Programming, Algorithms and Data Structures

Emre Güney, PhD

Master in Bioinformatics for Health Sciences
Universitat Pompeu Fabra

Python Overview

November 5-12, 2019

Institut Hospital del Mar d'Investigacions Mèdiques

RESEARCH PROGRAMME ON BIOMEDICAL INFORMATICS

UNIVERSITAT POMPEU FABRA

Python 3 Cheat Sheet

Latest version on :
https://perso.limsi.fr/pointal/python:memento

Base Types

integer, float, boolean, string, bytes

```
int 783  0  -192   0b010   0o642   0xF3
         zero      binary  octal   hexa
float 9.23 0.0   -1.7e-6
                     ×10⁻⁶
bool True  False
str "One\nTwo"          Multiline string:
     escaped new line     """X\tY\tZ
    'I\'m'                1\t2\t3"""
     escaped '            escaped tab
bytes b"toto\xfe\775"
           hexadecimal octal
```

☝ *immutables*

Container Types

▪ **ordered sequences**, fast index access, repeatable values

```
list [1,5,9]     ["x",11,8.9]     ["mot"]     []
tuple (1,5,9)    11,"y",7.4       ("mot",)    ()
```

Non modifiable values (immutables)   ☝ *expression with only comas →* `tuple`

```
str bytes   (ordered sequences of chars / bytes)    ""
                                                    b""
```

▪ **key containers**, no *a priori* order, fast key access, each key is unique

dictionary `dict {"key":"value"}`   `dict(a=3,b=4,k="v")`   `{}`

(key/value associations) `{1:"one",3:"three",2:"two",3.14:"π"}`

collection `set {"key1","key2"}`   `{1,9,3,0}`   `set()`

☝ *keys=hashable values (base types, immutables…)*   `frozenset` *immutable set*   *empty*

Identifiers

for variables, functions, modules, classes… names

`a…zA…Z_` followed by `a…zA…Z_0…9`

- diacritics allowed but should be avoided
- language keywords forbidden
- lower/UPPER case discrimination

☺ `a toto x7 y_max BigOne`
☹ ~~`8y`~~ ~~`and`~~ ~~`for`~~

Variables assignment

=

☝ assignment ⇔ **binding** of a *name* with a *value*

1) evaluation of right side expression value
2) assignment in order with left side names

`x=1.2+8+sin(y)`

`a=b=c=0` *assignment to same value*

`y,z,r=9.2,-7.6,0` *multiple assignments*

`a,b=b,a` *values swap*

`a,*b=seq` / `*a,b=seq` } *unpacking of sequence in item and list*

`x+=3` *increment ⇔* `x=x+3`   and `*=`

`x-=2` *decrement ⇔* `x=x-2`   `/=`

`x=None` *« undefined » constant value*   `%=`

`del x` *remove name* `x`   …

Conversions

`type(`*expression*`)`

`int("15")` → `15`

`int("3f",16)` → `63`   can specify integer number base in 2nd parameter

`int(15.56)` → `15`   truncate decimal part

`float("-11.24e8")` → `-1124000000.0`

`round(15.56,1)` → `15.6`   rounding to 1 decimal (0 decimal → integer number)

`bool(x)` `False` for null `x`, empty container `x` , `None` or `False` `x` ; `True` for other `x`

`str(x)` → `"…"`   representation string of `x` for display *(cf. formatting on the back)*

`chr(64)` → `'@'`   `ord('@')` → `64`   code ↔ char

`repr(x)` → `"…"`   *literal* representation string of `x`

`bytes([72,9,64])` → `b'H\t@'`

`list("abc")` → `['a','b','c']`

`dict([(3,"three"),(1,"one")])` → `{1:'one',3:'three'}`

`set(["one","two"])` → `{'one','two'}`

separator `str` and *sequence of* `str` → *assembled* `str`

`':'.join(['toto','12','pswd'])` → `'toto:12:pswd'`

`str` *splitted on whitespaces* → `list` of `str`

`"words with  spaces".split()` → `['words','with','spaces']`

`str` *splitted on separator* `str` → `list` *of* `str`

`"1,4,8,2".split(",")` → `['1','4','8','2']`

sequence of one type → `list` of another type *(via list comprehension)*

`[int(x) for x in ('1','29','-3')]` → `[1,29,-3]`

Source: Laurent Pointal

Sequence Containers Indexing

for lists, tuples, strings, bytes…

negative index	-5	-4	-3	-2	-1	
positive index	0	1	2	3	4	

```
lst=[10, 20, 30, 40, 50]
```

positive slice	0	1	2	3	4	5
negative slice	-5	-4	-3	-2	-1	

Items count

```
len(lst)→5
```

☝ **index from 0**
(here from 0 to 4)

Individual access to **items** via `lst[`*index*`]`

```
lst[0]→10    ⇒ first one     lst[1]→20
lst[-1]→50   ⇒ last one      lst[-2]→40
```

On mutable sequences (`list`), remove with `del lst[3]` *and modify with assignment* `lst[4]=25`

Access to **sub-sequences** via `lst[`*start slice*`:`*end slice*`:`*step*`]`

```
lst[:-1]→[10,20,30,40]   lst[::-1]→[50,40,30,20,10]   lst[1:3]→[20,30]    lst[:3]→[10,20,30]
lst[1:-1]→[20,30,40]     lst[::-2]→[50,30,10]         lst[-3:-1]→[30,40]  lst[3:]→[40,50]
lst[::2]→[10,30,50]      lst[:]→[10,20,30,40,50] shallow copy of sequence
```

Missing slice indication → from start / up to end.
On mutable sequences (`list`), remove with `del lst[3:5]` *and modify with assignment* `lst[1:4]=[15,25]`

Boolean Logic

Comparisons : `<  >  <=  >=  ==  !=`
(boolean results) ≤ ≥ = ≠

`a and b` logical and — *both simulta- -neously*

`a or b` logical or — *one or other or both*

☝ pitfall : `and` *and* `or` *return value of* `a` *or of* `b` *(under shortcut evaluation).*
⇒ *ensure that* `a` *and* `b` *are booleans.*

`not a` logical not

`True`, `False` } True and False constants

Statements Blocks

```
parent statement:
    statement block 1…
        ⁝
    parent statement:
        statement block2…
            ⁝
next statement after block 1
```

indentation !

☝ *configure editor to insert 4 spaces in place of an indentation tab.*

Modules/Names Imports

module `truc`⇔*file* `truc.py`

`from monmod import nom1,nom2 as fct`
→*direct access to names, renaming with* `as`

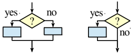
`import monmod` →*access via* `monmod.nom1` …

☝ modules and packages searched in *python path* (cf `sys.path`)

Conditional Statement

statement block executed only **if** *a condition is true*

```
if logical condition:
    statements block
```



Can go with several *elif*, *elif*... and only one final *else*. Only the block of first true condition is executed.

```
if age<=18:
    state="Kid"
elif age>65:
    state="Retired"
else:
    state="Active"
```

☝ *with a var* `x`:

```
if bool(x)==True:  ⇔ if x:
if bool(x)==False: ⇔ if not x:
```

Maths

☝ *floating numbers… approximated values*

Operators: `+ - * / // % **`
× ÷ integer ÷ ÷ remainder a^b

Priority `(…)`

`@` → matrix × *python3.5+`numpy`*

```
(1+5.3)*2→12.6
abs(-3.2)→3.2
round(3.57,1)→3.6
pow(4,3)→64.0
```

☝ *usual order of operations*

angles in radians

```
from math import sin,pi…
sin(pi/4)→0.707…
cos(2*pi/3)→-0.4999…
sqrt(81)→9.0     √
log(e**2)→2.0
ceil(12.5)→13
floor(12.5)→12
```

modules `math`, `statistics`, `random`, `decimal`, `fractions`, `numpy`, *etc. (cf. doc)*

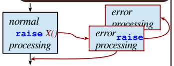
Exceptions on Errors

Signaling an error:

```
    raise ExcClass(…)
```

Errors processing:

```
try:
    normal procesising block
except Exception as e:
    error processing block
```



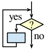
☝ `finally` *block for final processing in all cases.*

Source: Laurent Pointal

Conditional Loop Statement

*statements block executed **as long as** condition is true*

```
while logical condition:
    statements block
```



☝ beware of infinite loops!

```
s = 0
i = 1
while i <= 100:
    s = s + i**2
    i = i + 1
print("sum:",s)
```

*initializations **before** the loop*

*condition with a least one variable value (here **i**)*

☝ *make condition variable change !*

Algo: $s = \sum_{i=1}^{i=100} i^2$

Loop Control

break *immediate exit*

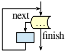
continue *next iteration*

☝ **else** *block for **normal** loop exit.*

Iterative Loop Statement

*statements block executed **for each** item of a container or iterator*

```
for var in sequence:
    statements block
```



Go over sequence's **values**

```
s = "Some text"
cnt = 0
for c in s:
    if c == "e":
        cnt = cnt + 1
print("found",cnt,"'e'")
```

*initializations **before** the loop*

*loop variable, **assignment managed by for statement***

Algo: count number of e *in the string.*

loop on dict/set ⇔ loop on keys sequences

use *slices* to loop on a subset of a sequence

Go over sequence's **index**

- modify item at index
- access items around index (before / after)

```
lst = [11,18,9,12,23,4,17]
lost = []
for idx in range(len(lst)):
    val = lst[idx]
    if val > 15:
        lost.append(val)
        lst[idx] = 15
print("modif:",lst,"-lost:",lost)
```

Algo: limit values greater than 15, memorizing of lost values.

☝ good habit : don't modify loop variable

Go simultaneously over sequence's **index** and **values**:

`for idx,val in enumerate(lst):`

Display

`print("v=",3,"cm :",x,",",y+4)`

items to display : literal values, variables, expressions

print options:

- `sep=" "` items separator, default space
- `end="\n"` end of print, default new line
- `file=sys.stdout` print to file, default standard output

Input

`s = input("Instructions:")`

☝ **input** always returns a **string**, convert it to required type (cf. boxed *Conversions* on the other side).

Generic Operations on Containers

`len(c)` → items count

`min(c)` `max(c)` `sum(c)`

`sorted(c)` → `list` sorted *copy*

*Note: For dictionaries and sets, these operations use **keys**.*

`val in c` → boolean, membership operator `in` (absence `not in`)

`enumerate(c)` → *iterator* on (index, value)

`zip(c1,c2…)` → *iterator* on tuples containing c_i items at same index

`all(c)` → `True` if **all** `c` items evaluated to true, else `False`

`any(c)` → `True` if **at least one** item of `c` evaluated true, else `False`

*Specific to **ordered sequences containers*** *(lists, tuples, strings, bytes…)*

`reversed(c)` → *inversed iterator* `c*5` → duplicate `c+c2` → concatenate

`c.index(`*val*`)` → *position* `c.count(`*val*`)` → *events count*

`import copy`

`copy.copy(c)` → shallow copy of container

`copy.deepcopy(c)` → deep copy of container

Integer Sequences

`range(`*[start,] end [,step]*`)`

☝ *start* default 0, *end* not included in sequence, *step* signed, default 1

`range(5)` → 0 1 2 3 4 `range(2,12,3)` → 2 5 8 11

`range(3,8)` → 3 4 5 6 7 `range(20,5,-5)` → 20 15 10

`range(len(`*seq*`))` → *sequence of index of values in seq*

☝ *range provides an immutable sequence of int constructed as needed*

Source: Laurent Pointal

Operations on Lists

☝ modify original list

- `lst.append(val)` add item at end
- `lst.extend(seq)` add sequence of items at end
- `lst.insert(idx, val)` insert item at index
- `lst.remove(val)` remove first item with value *val*
- `lst.pop([idx])` →*value* remove & return item at index *idx* (default last)
- `lst.sort()` `lst.reverse()` sort / reverse liste *in place*

Operations on Dictionaries

- `d[key]=value`
- `d[key]` → *value*
- `d.update(d2)` update/add associations
- `d.keys()`, `d.values()`, `d.items()` →*iterable views on keys/values/associations*
- `d.pop(key[,default])` → *value*
- `d.popitem()` → *(key,value)*
- `d.get(key[,default])` → *value*
- `d.setdefault(key[,default])` →*value*
- `d.clear()`
- `del d[key]`

Operations on Sets

Operators:
- `|` → union (vertical bar char)
- `&` → intersection
- `- ^` → difference/symmetric diff.
- `< <= > >=` → inclusion relations

Operators also exist as methods.

- `s.update(s2)` `s.copy()`
- `s.add(key)` `s.remove(key)`
- `s.discard(key)` `s.clear()`
- `s.pop()`

Files

storing data on disk, and reading it back

```
f = open("file.txt","w",encoding="utf8")
```

- file **variable** for operations
- **name** of file on disk (+path…)
- opening **mode**: □ `'r'` read, □ `'w'` write, □ `'a'` append, □ …`'+'` `'x'` `'b'` `'t'`
- **encoding** of chars for *text files*: `utf8` `ascii` `latin1` …

cf. modules `os`, `os.path` and `pathlib`

writing
- `f.write("coucou")`
- `f.writelines(list of lines)`

reading

☝ *read empty string if end of file*
- `f.read([n])` → *next chars* — *if n not specified, read up to end !*
- `f.readlines([n])` → *`list` of next lines*
- `f.readline()` → *next line*

☝ *text mode* `t` *by default (read/write* `str`*), possible binary mode* `b` *(read/write* `bytes`*). **Convert from/to required type !***

`f.close()` ☝ dont forget to **close the file** after use !

- `f.flush()` write cache
- `f.truncate([size])` resize

reading/writing progress sequentially in the file, modifiable with:
- `f.tell()` →*position*
- `f.seek(position[,origin])`

Very common: opening with a guarded block (automatic closing) and reading loop on lines of a text file:

```
with open(…) as f:
    for line in f :
        # processing of line
```

Function Definition

function name (identifier), named parameters

```
def fct(x,y,z):
    """documentation"""
    # statements block, res computation, etc.
    return res
```

`return res` ← result value of the call, if no computed result to return: `return None`

☝ parameters and all variables of this block exist only *in* the block and *during* the function call (think of a "black box")

Advanced: `def fct(x,y,z,*args,a=3,b=5,**kwargs):`
**args variable positional arguments (→`tuple`), default values,*
***kwargs variable named arguments (→`dict`)*

Function Call

`r = fct(3,i+2,2*i)`

storage/use of returned value — *one argument per parameter*

☝ this is the use of function name *with parentheses* which does the call

Advanced: **sequence* ***dict*

Operations on Strings

- `s.startswith(prefix[,start[,end]])`
- `s.endswith(suffix[,start[,end]])` `s.strip([chars])`
- `s.count(sub[,start[,end]])` `s.partition(sep)` → *(before,sep,after)*
- `s.index(sub[,start[,end]])` `s.find(sub[,start[,end]])`
- `s.is…()` *tests on chars categories (ex.* `s.isalpha()`*)*
- `s.upper()` `s.lower()` `s.title()` `s.swapcase()`
- `s.casefold()` `s.capitalize()` `s.center([width,fill])`
- `s.ljust([width,fill])` `s.rjust([width,fill])` `s.zfill([width])`
- `s.encode(encoding)` `s.split([sep])` `s.join(seq)`

Formatting

formating directives — values to format

`"modele{} {} {}".format(x,y,r)` → `str`

`"{selection:formatting!conversion}"`

□ **Selection** :
- `2`
- `nom`
- `0.nom`
- `4[key]`
- `0[2]`

Examples:
- `"{:+2.3f}".format(45.72793)` →`'+45.728'`
- `"{1:>10s}".format(8,"toto")` →`'      toto'`
- `"{x!r}".format(x="I'm")` →`'"I\'m"'`

□ **Formatting** :

fill char *alignment* *sign* *mini width*.*precision~maxwidth* *type*

`< > ^ =` `+ -` *space* `0` at start for filling with 0

integer: `b` binary, `c` char, `d` decimal (default), `o` octal, `x` or `X` hexa…
float: `e` or `E` exponential, `f` or `F` fixed point, `g` or `G` appropriate (default),
string: `s` … `%` percent

□ **Conversion** : `s` (readable text) or `r` (literal representation)

Source: Laurent Pointal

High-level, dynamically typed, easily readble

```
def quicksort(arr):
""" An example (not necessarily most efficient) implementation of quicksort algorithm"""

    if len(arr) <= 1:
        return arr

    pivot = arr[len(arr) // 2]

    left = [x for x in arr if x < pivot]
    middle = [x for x in arr if x == pivot]
    right = [x for x in arr if x > pivot]

    return quicksort(left) + middle + quicksort(right)

print(quicksort([3,6,8,10,1,2,1]))
# Output: [1, 1, 2, 3, 6, 8, 10]
```

Source: Justin Johnson (accessed on Nov 2019)

Understanding Reference Semantics

- **Assignment manipulates references**
  - —x = y **does not make a copy** of the object y references
  - —x = y makes x **reference** the object y references
- **Very useful; but beware!**
- **Example:**

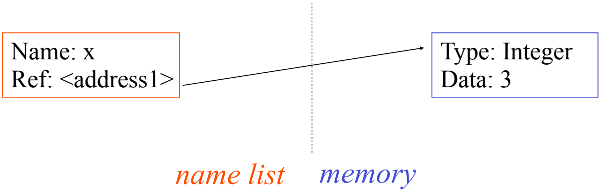
```
>>> a = [1, 2, 3]    # a now references the list [1, 2, 3]
>>> b = a            # b now references what a references
>>> a.append(4)      # this changes the list a references
>>> print b          # if we print what b references,
[1, 2, 3, 4]         # SURPRISE!  It has changed…
```

Why??

Source: SAO Telescope Data Center

Understanding Reference Semantics II

- **There is a lot going on when we type:**
  `x = 3`
- **First, an integer *3* is created and stored in memory**
- **A name *x* is created**
- **An *reference* to the memory location storing the *3* is then assigned to the name *x***
- **So: When we say that the value of *x* is *3***
- **we mean that *x* now refers to the integer *3***



Source: SAO Telescope Data Center

Understanding Reference Semantics III

- **The data 3 we created is of type integer. In Python, the datatypes integer, float, and string (and tuple) are "immutable."**
- **This doesn't mean we can't change the value of x, i.e. *change what x refers to* …**
- **For example, we could increment x:**

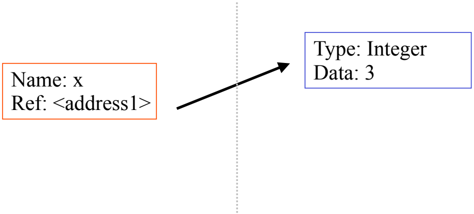
```
>>> x = 3
>>> x = x + 1
>>> print x
4
```


Source: SAO Telescope Data Center

Understanding Reference Semantics IV

- **If we increment x, then what's really happening is:**
  1. *The reference of name **X** is looked up.*
  2. *The value at that reference is retrieved.*

```
>>> x = x + 1
```

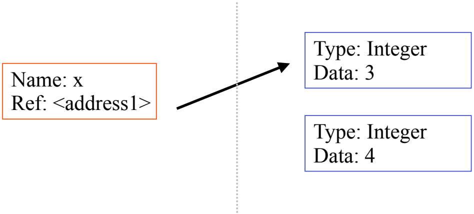


Source: SAO Telescope Data Center

Understanding Reference Semantics IV

- **If we increment x, then what's really happening is:**
  1. The reference of name ***x*** is looked up.
  2. The value at that reference is retrieved.
  3. *The 3+1 calculation occurs, producing a new data element* ***4*** *which is assigned to a fresh memory location with a new reference.*

```
>>> x = x + 1
```

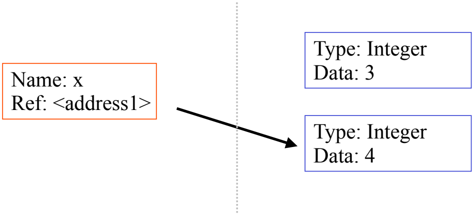


Source: SAO Telescope Data Center

Understanding Reference Semantics IV

- **If we increment x, then what's really happening is:**
  1. The reference of name ***x*** is looked up.
  2. The value at that reference is retrieved.
  3. The 3+1 calculation occurs, producing a new data element **4** which is assigned to a fresh memory location with a new reference.
  4. *The name **x** is changed to point to this new reference.*

```
>>> x = x + 1
```



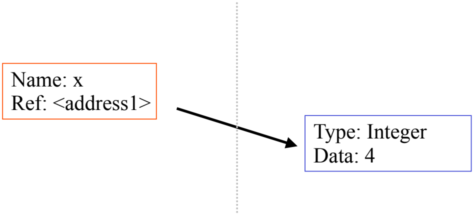
Source: SAO Telescope Data Center

Understanding Reference Semantics IV

- **If we increment x, then what's really happening is:**

```
>>> x = x + 1
```

1. The reference of name ***x*** is looked up.
2. The value at that reference is retrieved.
3. The 3+1 calculation occurs, producing a new data element **4** which is assigned to a fresh memory location with a new reference.
4. The name **x** is changed to point to this new reference.
5. *The old data **3** is garbage collected if no name still refers to it.*



Source: SAO Telescope Data Center

Sequence Types

1. Tuple
   - A simple ***immutable*** ordered sequence of items
   - Items can be of mixed types, including collection types

2. Strings
   - ***Immutable***
   - **Conceptually very much like a tuple**

3. List
   - ***Mutable*** ordered sequence of items of mixed types

Tuples: Immutable

```
>>> t = (23, 'abc', 4.56, (2,3), 'def')
>>> t[2] = 3.14

Traceback (most recent call last):
  File "<pyshell#75>", line 1, in -toplevel-
    tu[2] = 3.14
TypeError: object doesn't support item assignment
```

You can't change a tuple.

You can make a fresh tuple and assign its reference to a previously used name.

```
>>> t = (23, 'abc', 3.14, (2,3), 'def')
```

Lists: Mutable

```
>>> li = ['abc', 23, 4.34, 23]
>>> li[1] = 45
>>> li
  ['abc', 45, 4.34, 23]
```

- **We can change lists *in place.***
- **Name *li* still points to the same memory reference when we're done.**
- **The mutability of lists means that they aren't as fast as tuples.**

Tuples vs. Lists

- **Lists slower but more powerful than tuples.**
  - Lists can be modified, and they have lots of handy operations we can perform on them.
  - Tuples are immutable and have fewer features.

- **To convert between tuples and lists use the list() and tuple() functions:**

```
li = list(tu)
tu = tuple(li)
```

Source: SAO Telescope Data Center

Other containers – Python collections module

namedtuple	factory function for creating tuple subclasses with named fields
deque	list-like container with fast appends and pops on either end
ChainMap	dict-like class for creating a single view of multiple mappings
Counter	dict subclass for counting hashable objects
OrderedDict	dict subclass that remembers the order entries were added
defaultdict	dict subclass that calls a factory function to supply missing values

- book.pythontips.com/en/latest/collections.html
- docs.python.org/3.7/library/collections.html

Sources: Yasoob Khalid & Python Software Foundation (accessed on Nov 2019)

Generators, iterators and iterables

- Iterable (any object that returns an iterator or can take indices)
  - __iter__ or __get_item__
- Iterator
  - *next* or __next__
- Generator (one time iterators where items generated on demand)
  - *yield*

```
def generator_function(n):

    for i in range(n):
        yield i

    for item in generator_function():
        print(item)
```

Source: Yasoob Khalid (accessed on Nov 2019)

Function calls with extra arguments

```
def f(a, *b, **c):
    print(a)
    print(b)
    print(c)
```

```
> f(1, 2, c=5)
```

```
> f(a=1, b=2, c=5)
```

```
> f(1, 2, 4, 5)
```

```
> f(1, 2, 4, 5, b=3)
```

Function calls with extra arguments

```
def f(a, *b, **c):
    print(a)
    print(b)
    print(c)
```

```
> f(1, 2, c=5)
1
(2,)
{'c': 5}
> f(a=1, b=2, c=5)
1
()
{'b': 2, 'c': 5}
> f(1, 2, 4, 5)
1
(2, 4, 5)
{}
> f(1, 2, 4, 5, b=3)
1
(2, 4, 5)
{'b': 3}
```

Function calls with extra arguments

```
def f(a, *b, **c):
    print(a)
    print(b)
    print(c)
```

```
> f(1, 2, c=5)
1
(2,)
{'c': 5}
> f(a=1, b=2, c=5)
1
()
{'b': 2, 'c': 5}
> f(1, 2, 4, 5)
1
(2, 4, 5)
{}
> f(1, 2, 4, 5, b=3)
1
(2, 4, 5)
{'b': 3}
```

```
def   f(*args, **kwargs):
```

a, *b $\Rightarrow$ *args

**c $\Rightarrow$ **kwargs

Unpacking syntax & zip function

*: unpacks collection (to sequence of arguments)

**: unpacks dictionary (to sequence of named arguments)

zip: Makes an iterator aggregating elements from each of the input iterables

```
names = ["Joe", "James", "Zach"]
ages = [26, 39, 14]
tuples = list(zip(names, ages))
d = dict(zip(names,ages))

print(tuples) # Output: [('Joe', 26), ('James', 39), ('Zach', 14)]
print(list(zip(*tuples))) # Output: [('Joe', 'James', 'Zach'), (26, 39, 14)]
```

Unpacking syntax & zip function

*: unpacks collection (to sequence of arguments)

**: unpacks dictionary (to sequence of named arguments)

zip: Makes an iterator aggregating elements from each of the input iterables

```
names = ["Joe", "James", "Zach"]
ages = [26, 39, 14]
tuples = list(zip(names, ages))
d = dict(zip(names,ages))

print(tuples) # Output: [('Joe', 26), ('James', 39), ('Zach', 14)]
print(list(zip(*tuples))) # Output: [('Joe', 'James', 'Zach'), (26, 39, 14)]
```

Beware of the behaviour of functions with 'unusual inputs'

```
print(list(zip(range(5), range(3)))) # Output: [(0, 0), (1, 1), (2, 2)]
```

Functional programming & lambda function

lambda: Inline function without explicit name (anonymous)

```
add = lambda x, y: x + y

print(add(3, 5))
# Output: 8
```

```
a = [(1, 2), (4, 1), (9, 10), (13, -3)]
a.sort(key=lambda x: x[1])

print(a)
# Output: [(13, -3), (4, 1), (1, 2), (9, 10)]
```

Source: Yasoob Khalid (accessed on Nov 2019)

Functional programming & higher-order functions

Higher-order functions: Functions that accepts another function as input

- map

```
items = [1, 2, 3, 4, 5]
squared = list(map(lambda x: x**2, items))

print(squared) # Output: [1, 4, 9, 16, 25]
```

- filter

```
number_list = range(-5, 5)
less_than_zero = list(filter(lambda x: x < 0, number_list))

print(less_than_zero) # Output: [-5, -4, -3, -2, -1]
```

- reduce

```
from functools import reduce
product = reduce((lambda x, y: x * y), [1, 2, 3, 4])

print(product) # Output: 24
```

Source: Yasoob Khalid (accessed on Nov 2019)

For with else clause

General structure:

```
for item in container:
    if search_something(item): # Found it!
        process(item)
        break
else: # Didn’t find anything..
    not_found_in_container()
```

Example:

```
for n in range(2, 10):
    for x in range(2, n):
        if n % x == 0:
        print( n, ’equals’, x, ’*’, n/x)
        break
    else:
        # loop fell through without finding a factor
        print(n, ’is a prime number’)
```

Source: Yasoob Khalid (accessed on Nov 2019)

File I/O, Strings, Exceptions...

```
>>> try:
...     1 / 0
... except:
...     print('That was silly!')
... finally:
...     print('This gets executed no matter what')
...
That was silly!
This gets executed no matter what
```

with open('filename') as fileptr:

```
    somestring = fileptr.read()
    for line in fileptr:
       print line
```

fileptr.closed # True

(Context management through __enter__() and __exit__() methods)

```
>>> a = 1
>>> b = 2.4
>>> c = 'Tom'
>>> '%s has %d coins worth a total of $%.02f' % (c, a, b)
'Tom has 1 coins worth a total of $2.40'
```

(OR: '{} has {} coins worth a total of ${:.02f}'.format(c, a, b))

Source: SAO Telescope Data Center

Why Use Modules?

- **Code reuse**
  - Routines can be called multiple times within a program
  - Routines can be used from multiple programs
- **Namespace partitioning**
  - Group data together with functions used for that data
- **Implementing shared services or data**
  - Can provide global data structure that is accessed by multiple subprograms

Modules

- **Modules are functions and variables defined in separate files**
- **Items are imported using from or import**

```
from module import function
function()

import module
module.function()
```

- **Modules are namespaces**
  - Can be used to organize variable names, i.e.

```
atom.position = atom.position - molecule.position
```

Source: SAO Telescope Data Center

What is an Object?

- **A software item that contains variables and methods**
- **Object Oriented Design focuses on**
  - Encapsulation:
    - —dividing the code into a public interface, and a private implementation of that interface
  - Polymorphism:
    - —the ability to overload standard operators so that they have appropriate behavior based on their context
  - Inheritance:
    - —the ability to create subclasses that contain specializations of their parents

Source: SAO Telescope Data Center

Example

```
class atom(object):
  def __init__(self,atno,x,y,z):
       self.atno = atno
       self.position = (x,y,z)
  def symbol(self):   # a class method
       return Atno_to_Symbol[atno]
  def __repr__(self): # overloads printing
       return '%d %10.4f %10.4f %10.4f' %
              (self.atno, self.position[0],
               self.position[1],self.position[2])

>>> at = atom(6,0.0,1.0,2.0)
>>> print at
6  0.0000  1.0000 2.0000
>>> at.symbol()
'C'
```

Source: SAO Telescope Data Center

Atom Class

- **Overloaded the default constructor**
- **Defined class variables (atno,position) that are persistent and local to the atom object**
- **Good way to manage shared memory:**
  - instead of passing long lists of arguments, encapsulate some of this data into an object, and pass the object.
  - much cleaner programs result
- **Overloaded the print operator**

- **We now want to use the atom class to build molecules...**

Molecule Class

```
class molecule:
  def __init__(self,name='Generic'):
       self.name = name
       self.atomlist = []
  def addatom(self,atom):
       self.atomlist.append(atom)
  def __repr__(self):
       str = 'This is a molecule named %s\n' % self.name
       str = str+'It has %d atoms\n' % len(self.atomlist)
       for atom in self.atomlist:
              str = str + `atom` + '\n'
       return str
```


Source: SAO Telescope Data Center

Using Molecule Class

```
>>> mol = molecule('Water')
>>> at = atom(8,0.,0.,0.)
>>> mol.addatom(at)
>>> mol.addatom(atom(1,0.,0.,1.))
>>> mol.addatom(atom(1,0.,1.,0.))
>>> print mol
This is a molecule named Water
It has 3 atoms
8   0.000 0.000 0.000
1   0.000 0.000 1.000
1   0.000 1.000 0.000
```

- **Note that the print function calls the atoms print function**
  - Code reuse: only have to type the code that prints an atom once; this means that if you change the atom specification, you only have to one place to update.

Inheritance

```
class qm_molecule(molecule):
  def addbasis(self):
    self.basis = []
    for atom in self.atomlist:
      self.basis = add_bf(atom,self.basis)
```

- **__init__, __repr__, and __addatom__ are taken from the parent class (molecule)**
- **Added a new function addbasis() to add a basis set**
- **Another example of code reuse**
  - Basic functions don't have to be retyped, just inherited
  - Less to rewrite when specifications change

Source: SAO Telescope Data Center

Overloading

```
class qm_molecule(molecule):
  def __repr__(self):
    str = 'QM Rules!\n'
    for atom in self.atomlist:
      str = str + `atom` + '\n'
    return str
```

- **Now we only inherit __init__ and addatom from the parent**
- **We define a new version of __repr__ specially for QM**

Adding to Parent Functions

- **Sometimes you want to extend, rather than replace, the parent functions.**

```
class qm_molecule(molecule):
  def __init__(self,name="Generic",basis="6-31G**"):
    self.basis = basis
    super(qm_molecule, self).__init__(name)
```

Public and Private Data

- **In Python anything with two leading underscores is private**

  __a, __my_variable

- **Anything with one leading underscore is semi-private, and you should feel guilty accessing this data directly.**

  _b

  - Sometimes useful as an intermediate step to making data private

Public and private data – cont'd

```
class A:
    def __init__(self):
        self.__x = 1
        self._y = 2
        self.z = 3

a1 = A()
print(a1.z) # 3
print(a1._y) # 2
print(a1.__x) # AttributeError: 'A' object has no attribute '__x'
```

Instance vs class attributes & static vs class methods

- Instance attributes: attributes whose value depends on the specific instance
- Class attributes: attributes whose value is bound to the class and independent of any specific instance
- Class method can access and modify the class state (thus require a reference to the class) while static methods can not

```
class A:
    count = 0
    def __init__(self):
        self.__x = 1

    @staticmethod
    def get_count():
        return A.count

    @classmethod
    def set_count(cls):
        cls.count += 1
```

Magic methods

Internal methods, not meant to be invoked directly

- Invocation happens internally from the class on a certain action
- Syntax: def __magicMethod__()
- Python examples: __init__(), __str__(), __add__()
- For instance when you add two values using the + operator, internally, the __add__() method will be called

Source: Majeed Kassis - BGU

Garbage collection

Python objects are automatically deleted once there are no references to them

- The garbage collector removes the object instance once the reference count reaches zero

```
class Employee:
    def __init__(self):
        print("created")

    def __del__(self):
        print("destroyed")

e1 = Employee() # created
e2 = e1
print('e1 id=', id(e1)) # e1 id = 2257554491936
print('e2 id=', id(e2)) # e2 id = 2257554491936
del e1 # removes the reference e1, reducing the ref count by 1
print('e2 id=', id(e2)) # e2 id = 2257554491936
del e2 # destroyed, ref count is zero
```

Source: Majeed Kassis - BGU

Decorators

Decorators add functionality to existing functions without modifying the original code

```
def debug(func):
    def wrapper(*args, **kwargs):
        print('received arguments:', args, kwargs)
        return_values = func(*args, **kwargs)
        print('return value:', return_values)
        return return_values
    return wrapper

def add_two_elements(a, b):
    return a + b

debug_add_two_elements = debug(add_two_elements)
debug_add_two_elements(5, b=6)
# received arguments: (5,) {'b': 6}
# return value: 11
```

Source: Majeed Kassis - BGU

Functions as first class objects

- Functions can act like objects
  - referenced by another function as a variable
  - passed to another function as a variable
  - returned to another function as variable
- Functions can act as variables
  - defined inside another function
  - passed as arguments to another function
  - returned as values from another function

```
def make_divisibility_test(n):
    def divisible_by_n(m):
        return m % n == 0
    return divisible_by_n

is_divisable_by_five = make_divisibility_test(5)
is_divisable_by_five(10) # True
make_divisibility_test(7)(10) # False

div_by_3 = make_divisibility_test(3)
print(filter(div_by_3, range(10))) # 0, 3, 6, 9
```

Source: Majeed Kassis - BGU

Typing in Python

- Names for all standard built-in types are defined in *types* module
- isinstance() function is recommended for testing the type of an object (it takes subclasses into account)

```
type(2) # => <class 'int'>
type("Hi!") # => <class 'str'>
type(None) # => <class 'NoneType'>
type(int) # => <class 'type'>
```

```
class Plant: pass # Dummy class
class Tree(Plant): pass # Dummy class derived from Plant
tree = Tree() # A new instance of Tree class
print isinstance(tree, Tree) # True
print isinstance(tree, Plant) # True
print isinstance(tree, object) # True
print type(tree) is Tree # False
print type(tree).__name__ == "instance" # True
print tree.__class__.__name__ == "Tree" # True
```

Source: Wikibooks Python Programming (accessed on Nov 2019)

Static vs dynamic typing

- Dynamically typed: Typing is not enforced by default in python and checked in **runtime**

- Strongly type: Once the type is detected, operations can be executed only for that detected type

```
def greeting(name: str) -> str:
    return 'Hello ' + name

greeting(2) # Run time TypeError: must be str, not int
```

- Typing syntax could be used for annotating the code

- Type checking can be enforced with mypy package

```
$ mypy greeting.py
greeting.py:4: error: Argument 1 to "greeting" has incompatible
type "int"; expected "str"
```

Sources: Python Software Foundation & Mypy web (accessed on Nov 2019)

Static vs dynamic typing - cont'd

```
from typing import Dict

def get_first_name(full_name: str) -> str:
    return full_name.split(" ")[0]

fallback_name: Dict[str, str] = {
    "first_name": "UserFirstName",
    "last_name": "UserLastName"
}

raw_name: str = input("Please enter your name: ")
first_name: str = get_first_name(raw_name)

if not first_name: # If the user didn't type anything in, use the fallback name
    first_name = get_first_name(fallback_name)

print(f"Hi, first_name!")
```

```
$ mypy my_script.py
my_script.py:21: error: Argument 1 to "get_first_name" has incompatible type "Dict[str, str]"; expected "str"
```

Source: Adam Geitgey (accessed on Nov 2019)

Introspection and reflection

Ability to examine and modify an object (its attributes) at runtime

- Everything in python is an object (inherited from the built-in object class)
- Reflection-enabling functions include type(), isinstance(), callable(), dir() and getattr()
- *inspect* module provide more inspection functionality
- type(), isinstance(), issubclass() check type of an object/class

```
# Check the type/class of an object
type(my_object)
my_object.__class__

# check is the object is a specific type or class
isinstance(my_str, str)

# check if an object's class is a subclass of a parent class
issubclass(my_object.__class__, ParentClass)
```

Source: Bonne Nuit Metrics Blog (accessed on Nov 2019)

Introspection and reflection – cont'd

- dir() returns the list of names of attributes of an object, including its methods

```
class A:
    def __init__(self):
        self.__x = 1

a1 = A()
print(dir(a1))
# ['__class__', '__delattr__', '__dict__', '__dir__', '__doc__',
'__eq__', '__format__', '__ge__', '__getattribute__', '__gt__',
'__hash__', '__init__', '__init_subclass__', '__le__', '__lt__',
'__module__', '__ne__', '__new__', '__reduce__', '__reduce_ex__',
'__repr__', '__setattr__', '__sizeof__', '__str__', '__subclasshook__',
'__weakref__', '_A__x']

print(a1._A__x) # 1
```

Source: Bonne Nuit Metrics Blog (accessed on Nov 2019)

Introspection and reflection – cont'd

- callable() determines whether the object can be called

```
callable([1,2].pop) # True
```

- a class can be made callable by providing a __call__() method

```
class Sq:
    def __call__(self, x):
        return x**2

f = Sq()
f(2) # 4
```

- getattr() returns the value of an attribute of an object using the attribute name passed as a string

```
getattr(3, "__add__")(2) # 5
```

Source: Wikibooks Python Programming (accessed on Nov 2019)

Introspection and reflection – cont'd

```
class MyClass:
    def __getattr__(self, attr):
        greeting = " ".join(attr.split('_')[1:])
    return greeting.capitalize()

m = MyClass()
m.say_hello() # TypeError: 'str' object is not callable
m.say_hello # 'Hello'
```

```
class MyClass:
    def __getattr__(self, attr):
        def _call():
            greeting = " ".join(attr.split('_')[1:])
            return greeting.capitalize()
        return _call

m = MyClass()
m.say_hello() # 'Hello'
m.say_bye_bye() # 'Bye bye'
```

Introspection and reflection – cont'd

```
class MyClass:

    def __init__(self):
        self.say_hello = 0

    def __getattr__(self, attr):
        def _call():
            greeting = " ".join(attr.split('_')[1:])
            return greeting.capitalize()
        return _call

    def say_hello(self):
        return 'Hi'

m = MyClass()
m.say_hello # 0
m.say_hello() # TypeError: 'int' object is not callable
m.say_bye_bye() # 'Bye bye'
```

Instance attribute (self.say_hello) takes precedence over class attribute (say_hello(self))

Introspection and reflection – cont'd

Instance attributes 'usually' take precedence over class attributes

(See this Stackoverflow post)

```
class Foo(object):

    def __init__(self, lst):
        self.lst = lst

    def sum(self):
        self.sum = sum(self.lst)
        return self.sum

f = Foo([1,2,3])
print(f.sum()) # None ⇒ self.sum = 6
print(f.sum()) # TypeError: 'int' object is not callable ⇒ print(6())
```

Source: Stackoverflow Blckknght (accessed on Nov 2019)

Introspection and reflection – cont'd

```
def capit(func):
    def wrapper(*args, **kwargs):
        return func(*args, **kwargs).capitalize()
return wrapper

class MyClass:

    def __getattr__(self, attr):
        @capit # decorator
        def _call():
            greeting = " ".join(attr.split('_')[1:])
            return greeting
        return _call

m = MyClass()
m.say_hello() # 'Hello'
m.say_bye_bye() # 'Bye bye'
```

Introspection and reflection – cont'd

```
class GreetMe:
    def __init__(self, name):
        self.name = name

    def __getattr__(self, attr):
        allowed = ['hello', 'bye', 'nice_to_meet_you', 'good_bye', 'goodnight']

        def call_(name=None):
            if attr in allowed:
                greeting = attr.replace('_', ' ')
                target = name if name else self.name
                return f"target, greeting.capitalize()"
            else:
                raise ValueError(f"Invalid name or greeting: name, attr")
        return call_

greet = GreetMe('Luna')
greet.hello() # Outputs: 'Luna, Hello'
greet.bye(name='John') # Outputs: 'John, Bye'
greet.nice_to_meet_you(name='Jane') # Outputs: 'Jane, Nice to meet you'
```

Source: Bonne Nuit Metrics Blog (accessed on Nov 2019)

Defining tests using unittest module

```
import unittest
class TestStatisticalFunctions(unittest.TestCase):
    def test_average(self):
        self.assertEqual(average([20, 30, 70]), 40.0)
        self.assertEqual(round(average([1, 5, 7]), 1), 4.3)
        with self.assertRaises(ZeroDivisionError):
            average([])
        with self.assertRaises(TypeError):
            average(20, 30, 70)

unittest.main() # Calling from the command line invokes all tests
```

Source: Paul Fodor - SBU (accessed on Nov 2019)

Debugging & profiling

```
import pdb

def make_bread():
    pdb.set_trace()
    return "I don't have time"

print(make_bread())
```

Debugging using pdb package

```
$ python -m pdb my_script.py
```

- c: continue execution
- w: shows the context of the current line it is executing
- a: prints the argument list of the current function
- s: executes the current line and stop at the first possible occasion
- n: continue execution until the next line in the current function is reached or it returns

Profiling using cProfile package

```
$ python -m cProfile my_script.py
```

Memoization: Caching Technique

- A technique to **speed** up Python programs using **caching**.
  - A cache **stores** the results of an operation for **later** use.
- It caches a function output based on the given input parameters.
  - It will compute the output **once** for each input set of parameters.
  - Every **consequent** call after the first will be quickly retrieved from a cache!
- When to use?
  - Only in cases where the code is expensive to run.
  - Expensive code in terms of storage space and execution time.

Source: Majeed Kassis - BGU

Memoization: Implementation

- Implementing Memoization:
  - Set up a cache data structure for function results
  - Every time the function is called, do one of the following:
    - Return the cached result, if any; *or*
    - Call the function to compute the missing result, and then update the cache before returning the result to the caller
  - Given enough cache storage this virtually guarantees that function results for a specific set of function arguments will only be computed once.
- As soon as we have a cached result we won't have to re-run the memoized function for the same set of inputs.
  - Instead, we can just fetch the cached result and return it right away!

Source: Majeed Kassis - BGU

Python Decorators: Memoization Decorator

```python
# Memoization Decorator
def memoize(func):
    cache = dict()
    def memoized_func(*args):
        if args in cache:
            return cache[args]
        result = func(*args)
        cache[args] = result
        return result
    return memoized_func
```

```python
def fibonacci(n):
    if n == 0:
        return 0
    elif n == 1:
        return 1
    return fibonacci(n - 1) + fibonacci(n - 2)

import timeit

# Normal Fibbonaci
print('normal run')
print(timeit.timeit('fibonacci(35)', globals=globals(),
number=1))
print(timeit.timeit('fibonacci(35)', globals=globals(),
number=1))

# Memoized Fibbonaci
print('memoized run')
m_fibonacci = memoize(fibonacci)
print(timeit.timeit('m_fibonacci(35)', globals=globals(),
number=1))
print(timeit.timeit('m_fibonacci(35)', globals=globals(),
number=1))
```

Source: Majeed Kassis - BGU

lru_cache (Python 3.2+)

```
from functools import lru_cache

@lru_cache(maxsize=32)
def fib(n):
    if n < 2:
        return n
    return fib(n-1) + fib(n-2)

>>> print([fib(n) for n in range(10)])
# Output: [0, 1, 1, 2, 3, 5, 8, 13, 21, 34]
```

Memoization Caveats

- Memoize **deterministic** functions only!
  - Caching non-deterministic functions will return incorrect or unexpected results.
- Ensure **caching space** complexity is much smaller than function execution complexity!
  - It is best to have a **bounded** (max size) caching storage.
- Bounded caches require a **smart purge** once it is full:
  - By gathering usage statistics
  - Discarding older entries
  - Discarding least frequently unused entries, etc

Source: Majeed Kassis - BGU

Using __slot__ for reducing memory usage

- Python uses a dict to store an object's instance attributes
- Creating a lot of objects ($>> 100,000$) uses large memory due to the storage of (default) attributes
- __slot__ can be used to tell Python not to use a dict and only allocate space for a fixed set of attributes
- Could give upto 50% reduction in memory usage (see PyPy which does this kind of optimizations by default)

```
class MyClass(object):

    __slots__ = ['name', 'identifier']

    def __init__(self, name, identifier):
        self.name = name
        self.identifier = identifier
        self.set_up()

    # ...
```

Source: Yasoob Khalid (accessed on Nov 2019)

Python General Good Programming Practices

- We will follow PEP8 coding conventions for good programming practices
  - Complete guidelines: https://www.python.org/dev/peps/pep-0008/
- Spacing:
  - 4 spaces to indent. No tabs.
  - Use blank lines to separate functions and logical sections inside functions.
  - Use spaces around operators and after commas, but not directly inside delimiters
- Commenting:
  - Comment all nontrivial functions.
  - Add header comments at the top of files before any imports.
    - If possible, put comments on a line of their own.

Source: Majeed Kassis - BGU

Python General Good Programming Practices

- Naming:
  - snake_case for variables/functions
  - CamelCase for classes
  - CAPS_CASE for constants
- Decomposition and Logic:
  - Simple is better than complex.
  - Seek abstractions and clean design.
- Automated Code Style Checking: (PEP8)
  - `PyLintOnline`: Captures mechanical violations (naming, spacing); advanced suggestions.
  - `pycodestyle`: `install using pip install pycodestyle`

Source: Majeed Kassis - BGU

Python Good Programming Practices

- Swapping two variables:

```python
# Bad
temp = a
a = b
b = temp
```

```python
# Good
a, b = b, a
```

Source: Majeed Kassis - BGU

Python Good Programming Practices

- Loop unpacking:

```python
# Bad
for bundle in zip([1,2,3],'abc'):
    num, let = bundle
    print(let * num)
# Bad
for key in d:
    val = d[key]
    print('{}: {}'.format(key, val))
```

```python
# Good
for num, let in zip([1,2,3],'abc'):
    print(let * num)

# Good
for key, val in d.items():
    print('{}->{}'.format(key,val))
```

Source: Majeed Kassis - BGU

Python Good Programming Practices

- Enumerate Iterables:

```python
# Bad
for index in range(len(arr)):
    elem = arr[index]
    print(elem)
# Bad
for index in range(len(arr)):
    elem = array[index]
    print(index, elem)
```

```python
# Good
for elem in arr:
    print(elem)

# Good
for index, elem in
enumerate(arr):
    print(index, elem)
```

Source: Majeed Kassis - BGU

Python Good Programming Practices

- Joining Strings:

```python
# Bad
s = ''
for color in colors:
    s += color

# Bad
s = ''
for color in colors:
    s += color + ', '
s = s[:-2]
```

```python
# Good
s = ''.join(colors)

# Good
s = ', '.join(colors)
```

Source: Majeed Kassis - BGU

Python Good Programming Practices

- Reduce In-Memory Buffering:

```python
# Bad
', '.join([color.upper() for color in colors])
# Bad
map(lambda x: int(x) ** 2, [line.strip() for line in
file])
# Bad
sum([n ** 2 for n in range(1000)])
```

```python
# Good
', '.join((color.upper() for color in colors))
# Good
map(lambda x: int(x) ** 2, (line.strip() for line in
file))
# Good
sum(n ** 2 for n in range(1000))
```

Source: Majeed Kassis - BGU

Python Good Programming Practices

- Chained Comparison Tests:

```python
# Bad
    return 0 < x and x < 10
# Bad
if a < x and x < b:
    return x
```

```python
# Good
    return 0 < x < 10
# Good
if a < x < b:
    return x
```

Source: Majeed Kassis - BGU

Python Good Programming Practices

- Using in:

```python
# Bad
if d.has_key(key):
    print("Here!")
# Bad
if x == 1 or x == 2 or x == 3:
    return True
# Bad
if 'hello'.find('lo') != -1:
    print("Found")
```

```python
# Good
if key in d:
    print("Here!")
# Good
if x in [1, 2, 3]:
    return True
# Good
if 'lo' in 'hello':
    print("Found")
```

Source: Majeed Kassis - BGU

Python Good Programming Practices

- Boolean Tests:

```python
# Bad
if x == True:
    print("Yes")
# Bad
if len(items) > 0:
    print("Nonempty")
# Bad
if items != []:
    print("Nonempty")
# Bad
if x != None:
    print("Something")
```

```python
# Good
if x:
    print("Yes")
# Good
if items:
    print("Nonempty")
# Good
if items:
    print("Nonempty")
# Good
if x is not None:
    print("Something")
```

Source: Majeed Kassis - BGU

Python Good Programming Practices

- Ignore values using underscore:

```python
# Bad
for i in range(10):
    x = input("> ")
    print(x[::-1])
```

```python
# Good
for _ in range(10):
    x = input("> ")
    print(x[::-1])
```

Source: Majeed Kassis - BGU

Python Good Programming Practices

- Looping Correctly:

```python
# Bad
for i in range(len(colors)):
    color = colors[i]
    name = names[i]
    print(color, name)
# Bad
for ind in range(len(elems) - 1, -1, -1):
    print(elems[ind])
```

```python
# Good
for color, name in zip(colors, names):
    print(color, name)

# Good
for elem in reversed(elems):
    print(elem)
```

Source: Majeed Kassis - BGU

Python Good Programming Practices

- Initializing Lists:

```python
# Bad
nones = [None, None, None, None]
# Bad
two_dim = [[None] * 4] * 5

# Good
nones = [None] * 4
# Good
two_dim = [[None] * 4 for _ in range(5)]
```

Source: Majeed Kassis - BGU

Python Good Programming Practices

- **Avoid** mutable default parameters:

```python
# Bad
def init_list(x, li=[]):
    li.append(x)
    print(li)


init_list(1, [4]) # =>
[4, 1]
init_list(3) # => [3]
init_list(3) # => [3, 3]
init_list(3) # => [3, 3, 3]
```

```python
# Good
def init_list(x, li=None):
    if li is None:
        li = []
    li.append(x)
    print(li)

init_list(1, [4]) # => [4, 1]
init_list(3) # => [3]
init_list(3) # => [3]
init_list(3) # => [3]
```

Source: Majeed Kassis - BGU

Python Good Programming Practices

- Use Comprehensions:

```python
# Bad
out = []
for word in lex:
    if word.endswith('py'):
        out.append(word[:-2])
# Bad
lengths = set()
for word in lex:
    lengths.add(len(word))
```

```python
# Good
out = [word[:-2] for word in lex if
word.endswith('py')]

# Good
lengths = {len(word) for word in lex}
```

Source: Majeed Kassis - BGU

Python Good Programming Practices

- Use Context Managers:

```python
# Bad
f = open('path/to/file')
try:
    raw = f.read()

except IOError as e:
    print('IOError:', e)
finally:
    f.close()
```

```python
# Good
with open('path/to/file') as f:
    raw = f.read()
```

Source: Majeed Kassis - BGU

Python Good Programming Practices

- In python, EAFP > LBYL:
  - EAFP: It's easier to ask for forgiveness than permission.
  - LBYL: Look before you leap

```python
# LBYL
def safe_div(m, n):
    if n == 0:
        print("Can't divide by 0")
        return None
    return m / n

# EAFP
def safe_div(m, n):
    try:
        return m / n
    except ZeroDivisionError:
        print("Can't divide by 0")
    return None
```

Source: Majeed Kassis - BGU

Python Good Programming Practices

- **Avoid** using Catch-Alls:

```python
# Bad
try:
    n = int(input("> "))
except:
    print("Invalid input.")
else:
    return n ** 2

# Good
try:
    n = int(input("> "))
except ValueError:
    print("Invalid input.")
else:
    return n ** 2
```

Source: Majeed Kassis - BGU

Python Good Programming Practices

- Use Custom Made Exceptions:

```python
# Bad
if not self.available_cheeses:
    raise ValueError("No cheese!")

# Good
class NoCheeseError(ValueError):
    pass
if not self.available_cheeses:
    raise NoCheeseError("I'm afraid we're right out, sir.")
```

Source: Majeed Kassis - BGU

Python Good Programming Practices

- Always implement Magic Methods for the defined Class:

```python
class Vector():
    def __init__(self, elems):
        self.elems = elms
    def size(self):
        return len(self.elems)

v = Vector([1,2])
len(v) # => fails!
```

```python
class Vector():
    def __init__(self, elems):
        self.elems = elms
    def __len__(self):
        return len(self.elems)

v = Vector([1,2])
len(v) # => succeeds!
```

Source: Majeed Kassis - BGU

Serialization using Pickle

```
# Importing pickle
try:
    import cPickle as pickle # Python 2
except ImportError:
    import pickle # Python 3

# Creating Pythonic object:
class Family(object):
    def __init__(self, names):
        self.sons = names
    def __str__(self):
        return ' '.join(self.sons)

my_family = Family(['John', 'David'])

# Dumping to string
pickle_data = pickle.dumps(my_family, pickle.HIGHEST_PROTOCOL)
```

Source: Python notes for professionals (accessed on Nov 2019)

Serialization using JSON

```
import json

families = (['John'], ['Mark', 'David', 'name': 'Avraham'])

# Dumping it into string
json_families = json.dumps(families)
# [["John"], ["Mark", "David", "name": "Avraham"]]
# Pretty printing
print(json.dumps(json_families, indent = 4, sort_keys=True))

# Dumping it to file
with open('families.json', 'w') as json_file:
    json.dump(families, json_file)

# Loading it from string
json_families = json.loads(json_families)

# Loading it from file
with open('families.json', 'r') as json_file:
    json_families = json.load(json_file)
```


Sources: Python notes for professionals and Programiz JSON tutorial (accessed on Nov 2019)

XML parsing

- xml.etree.ElementTree

```
import xml.etree.ElementTree as ET
tree = ET.parse('country_data.xml')
root = tree.getroot()

for child in root:
    print(child.tag, child.attrib)

for neighbor in root.iter('neighbor'):
    print(neighbor.attrib)

for country in root.findall('country'):
    rank = country.find('rank').text
    name = country.get('name')
    print(name, rank)
```

Sources: Python documentation (accessed on Nov 2019)

XML parsing – 3rd party

- untangle

```
import untangle

obj = untangle.parse('path/to/file.xml')
obj.root.child['my_attr']
```

- xmltodict

```
import xmltodict

with open('path/to/file.xml') as fd:
    doc = xmltodict.parse(fd.read())

doc['my_tag']['my_attr']
```

Source: THGtP (accessed on Nov 2019)

Misc

- Native C calls in Python (extending Python with C)
  - https://docs.python.org/3.7/extending/extending.html
  - https://www.csestack.org/calling-c-functions-from-python/
- Multiprocessing
  https://docs.python.org/3.7/library/multiprocessing.html
- Logging
  https://docs.python.org/3.7/howto/logging-cookbook.html