TYPE -1
Python 3 Cheat Sheet

Ashan Mir Founder - MTC

Base Types

```
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         unpacking of sequence in
*a,b=seq         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]
```

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`

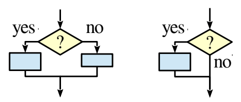
`import monmod` → access via `monmod.nom1` …

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 — *angles in radians*

Operators: `+ - * / // % **`
Priority `(…)` × ÷ — integer ÷ — ÷ remainder — a^b

`@` → 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

```
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
```

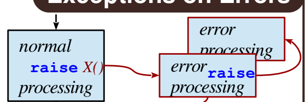
modules `math`, `statistics`, `random`, `decimal`, `fractions`, `numpy`, etc. (cf. doc)

Exceptions on Errors

Signaling an error:
`raise ExcClass(…)`

Errors processing:

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



`finally` block for final processing in all cases.

TYPE -1

Conditional Loop Statement

statements block executed *as long as* condition is true

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

beware of infinite loops!

```
s = 0    } initializations before the loop
i = 1    } condition with a least one variable value (here i)
while i <= 100:
    s = s + i**2
    i = i + 1        # make condition variable change !
print("sum:",s)
```

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"   } initializations before the loop
cnt = 0
           loop variable, assignment managed by for statement
for c in s:
    if c == "e":
        cnt = cnt + 1
print("found",cnt,"'e'")
```

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)`
Note: For dictionaries and sets, these operations use keys.
`sorted(c)` → `list` sorted *copy*
`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

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

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
```

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

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

formatting 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)

TYPE -2

MTC

Python Cheat Sheet
by Mikey Tech

Python sys Variables

argv	Command line args
builtin_module_-names	Linked C modules
byteorder	Native byte order
check_interval	Signal check frequency
exec_prefix	Root directory
executable	Name of executable
exitfunc	Exit function name
modules	Loaded modules
path	Search path
platform	Current platform
stdin, stdout, stderr	File objects for I/O
version_info	Python version info
winver	Version number

Python sys.argv

sys.argv[0]	foo.py
sys.argv[1]	bar
sys.argv[2]	-c
sys.argv[3]	qux
sys.argv[4]	--h

sys.argv for the command:
$ python foo.py bar -c qux --h

Python os Variables

altsep	Alternative sep
curdir	Current dir string
defpath	Default search path
devnull	Path of null device
extsep	Extension separator
linesep	Line separator
name	Name of OS
pardir	Parent dir string
pathsep	Patch separator
sep	Path separator

Registered OS names: "posix", "nt", "mac", "os2", "ce", "java", "riscos"

Python Class Special Methods

__new__(cls)	__lt__(self, other)
__init__(self, args)	__le__(self, other)
__del__(self)	__gt__(self, other)
__repr__(self)	__ge__(self, other)
__str__(self)	__eq__(self, other)
__cmp__(self, other)	__ne__(self, other)
__index__(self)	__nonzero__(self)
__hash__(self)	
__getattr__(self, name)	
__getattribute__(self, name)	
__setattr__(self, name, attr)	
__delattr__(self, name)	
__call__(self, args, kwargs)	

Python List Methods

append(item)	pop(position)
count(item)	remove(item)
extend(list)	reverse()
index(item)	sort()
insert(position, item)	

Python String Methods

capitalize() *	lstrip()
center(width)	partition(sep)
count(sub, start, end)	replace(old, new)
decode()	rfind(sub, start ,end)
encode()	rindex(sub, start, end)
endswith(sub)	rjust(width)
expandtabs()	rpartition(sep)
find(sub, start, end)	rsplit(sep)
index(sub, start, end)	rstrip()
isalnum() *	split(sep)
isalpha() *	splitlines()
isdigit() *	startswith(sub)
islower() *	strip()
isspace() *	swapcase() *

Python String Methods (cont)

istitle() *	title() *
isupper() *	translate(table)
join()	upper() *
ljust(width)	zfill(width)
lower() *	

Methods marked * are locale dependant for 8-bit strings.

Python File Methods

close()	readlines(size)
flush()	seek(offset)
fileno()	tell()
isatty()	truncate(size)
next()	write(string)
read(size)	writelines(list)
readline(size)	

Python Indexes and Slices

len(a)	6
a[0]	0
a[5]	5
a[-1]	5
a[-2]	4
a[1:]	[1,2,3,4,5]
a[:5]	[0,1,2,3,4]
a[:-2]	[0,1,2,3]
a[1:3]	[1,2]
a[1:-1]	[1,2,3,4]
b=a[:]	Shallow copy of a

Indexes and Slices of a=[0,1,2,3,4,5]

Python Datetime Methods

today()	fromordinal(ordinal)
now(timezoneinfo)	combine(date, time)
utcnow()	strptime(date, format)
fromtimestamp(timestamp)	
utcfromtimestamp(timestamp)	

TYPE -2

MTC

Python Cheat Sheet

by Mikey Tech

Python Time Methods

replace()	utcoffset()
isoformat()	dst()
__str__()	tzname()
strftime(format)	

Python Date Formatting

%a	Abbreviated weekday (Sun)
%A	Weekday (Sunday)
%b	Abbreviated month name (Jan)
%B	Month name (January)
%c	Date and time
%d	Day (leading zeros) (01 to 31)
%H	24 hour (leading zeros) (00 to 23)
%I	12 hour (leading zeros) (01 to 12)
%j	Day of year (001 to 366)
%m	Month (01 to 12)
%M	Minute (00 to 59)
%p	AM or PM
%S	Second (00 to 61[4])
%U	Week number[1] (00 to 53)
%w	Weekday[2] (0 to 6)
%W	Week number[3] (00 to 53)
%x	Date
%X	Time
%y	Year without century (00 to 99)
%Y	Year (2008)
%Z	Time zone (GMT)
%%	A literal "%" character (%)

[1] Sunday as start of week. All days in a new year preceding the first Sunday are considered to be in week 0.
[2] 0 is Sunday, 6 is Saturday.
[3] Monday as start of week. All days in a new year preceding the first Monday are considered to be in week 0.
[4] This is not a mistake. Range takes account of leap and double-leap seconds.

Contribute by TYPE-3

Variables and Data Types

Variable Assignment

```
>>> x=5
>>> x
 5
```

Calculations With Variables

`>>> x+2` → `7`	Sum of two variables
`>>> x-2` → `3`	Subtraction of two variables
`>>> x*2` → `10`	Multiplication of two variables
`>>> x**2` → `25`	Exponentiation of a variable
`>>> x%2` → `1`	Remainder of a variable
`>>> x/float(2)` → `2.5`	Division of a variable

Types and Type Conversion

`str()`	`'5', '3.45', 'True'`	Variables to strings
`int()`	`5, 3, 1`	Variables to integers
`float()`	`5.0, 1.0`	Variables to floats
`bool()`	`True, True, True`	Variables to booleans

Asking For Help

```
>>> help(str)
```

Lists

Also see NumPy Arrays

```
>>> a = 'is'
>>> b = 'nice'
>>> my_list = ['my', 'list', a, b]
>>> my_list2 = [[4,5,6,7], [3,4,5,6]]
```

Selecting List Elements

Index starts at 0

Subset

`>>> my_list[1]`	Select item at index 1
`>>> my_list[-3]`	Select 3rd last item

Slice

`>>> my_list[1:3]`	Select items at index 1 and 2
`>>> my_list[1:]`	Select items after index 0
`>>> my_list[:3]`	Select items before index 3
`>>> my_list[:]`	Copy my_list

Subset Lists of Lists

`>>> my_list2[1][0]`	my_list[list][itemOfList]
`>>> my_list2[1][:2]`	

List Operations

```
>>> my_list + my_list
['my', 'list', 'is', 'nice', 'my', 'list', 'is', 'nice']
>>> my_list * 2
['my', 'list', 'is', 'nice', 'my', 'list', 'is', 'nice']
>>> my_list2 > 4
True
```

List Methods

`>>> my_list.index(a)`	Get the index of an item
`>>> my_list.count(a)`	Count an item
`>>> my_list.append('!')`	Append an item at a time
`>>> my_list.remove('!')`	Remove an item
`>>> del(my_list[0:1])`	Remove an item
`>>> my_list.reverse()`	Reverse the list
`>>> my_list.extend('!')`	Append an item
`>>> my_list.pop(-1)`	Remove an item
`>>> my_list.insert(0,'!')`	Insert an item
`>>> my_list.sort()`	Sort the list

Strings

```
>>> my_string = 'thisStringIsAwesome'
>>> my_string
'thisStringIsAwesome'
```

String Operations

```
>>> my_string * 2
 'thisStringIsAwesomethisStringIsAwesome'
>>> my_string + 'Innit'
 'thisStringIsAwesomeInnit'
>>> 'm' in my_string
 True
```

String Operations

Index starts at 0

```
>>> my_string[3]
>>> my_string[4:9]
```

String Methods

`>>> my_string.upper()`	String to uppercase
`>>> my_string.lower()`	String to lowercase
`>>> my_string.count('w')`	Count String elements
`>>> my_string.replace('e', 'i')`	Replace String elements
`>>> my_string.strip()`	Strip whitespaces

Libraries

Import libraries

```
>>> import numpy
>>> import numpy as np
```

Selective import

```
>>> from math import pi
```

pandas — Data analysis

learn — Machine learning

NumPy — Scientific computing

matplotlib — 2D plotting

Install Python

ANACONDA — Leading open data science platform powered by Python

spyder — Free IDE that is included with Anaconda

jupyter — Create and share documents with live code, visualizations, text, ...

Numpy Arrays

Also see Lists

```
>>> my_list = [1, 2, 3, 4]
>>> my_array = np.array(my_list)
>>> my_2darray = np.array([[1,2,3],[4,5,6]])
```

Selecting Numpy Array Elements

Index starts at 0

Subset

`>>> my_array[1]` → `2`	Select item at index 1

Slice

`>>> my_array[0:2]` → `array([1, 2])`	Select items at index 0 and 1

Subset 2D Numpy arrays

`>>> my_2darray[:,0]` → `array([1, 4])`	my_2darray[rows, columns]

Numpy Array Operations

```
>>> my_array > 3
 array([False, False, False,  True], dtype=bool)
>>> my_array * 2
 array([2, 4, 6, 8])
>>> my_array + np.array([5, 6, 7, 8])
 array([6, 8, 10, 12])
```

Numpy Array Functions

`>>> my_array.shape`	Get the dimensions of the array
`>>> np.append(other_array)`	Append items to an array
`>>> np.insert(my_array, 1, 5)`	Insert items in an array
`>>> np.delete(my_array,[1])`	Delete items in an array
`>>> np.mean(my_array)`	Mean of the array
`>>> np.median(my_array)`	Median of the array
`>>> my_array.corrcoef()`	Correlation coefficient
`>>> np.std(my_array)`	Standard deviation

TYPE -4

Python Cheat Sheet

Python 3 is a truly versatile programming language, loved both by web developers, data scientists and software engineers. And there are several good reasons for that!

- Python is open-source and has a great support community,
- Plus, extensive support libraries.
- Its data structures are user-friendly.

Once you get a hang of it, your development speed and productivity will soar!

Table of Contents

Python Basics: Getting Started

Most Windows and Mac computers come with Python pre-installed. You can check that via a Command Line search. The particular appeal of Python is that you can write a program in any text editor, save it in .py format and then run via a Command Line. But as you learn to write more complex code or venture into data science, you might want to switch to an IDE or IDLE.

What is IDLE (Integrated Development and Learning)

IDLE (Integrated Development and Learning Environment) comes with every Python installation. Its advantage over other text editors is that it highlights important keywords (e.g. string functions), making it easier for you to interpret code.

Shell is the default mode of operation for Python IDLE. In essence, it's a simple loop that performs that following four steps:

- Reads the Python statement
- Evaluates the results of it
- Prints the result on the screen
- And then loops back to read the next statement.

Python shell is a great place to test various small code snippets.

Main Python Data Types

Every value in Python is called an "**object**". And every object has a specific data type. The three most-used data types are as follows:

Integers (int) — an integer number to represent an object such as "number 3".

`Integers`	`-2, -1, 0, 1, 2, 3, 4, 5`

Floating-point numbers (float) — use them to represent floating-point numbers.

`Floating-point numbers`	`-1.25, -1.0, --0.5, 0.0, 0.5, 1.0, 1.25`

Strings — codify a sequence of characters using a string. For example, the word "hello". In Python 3, strings are immutable. If you already defined one, you cannot change it later on.

While you can modify a string with commands such as **replace()** or **join()**, they will create a copy of a string and apply modification to it, rather than rewrite the original one.

`Strings`	`'yo', 'hey', 'Hello!', 'what's up!'`

Plus, another three types worth mentioning are **list**s, **dict**ionaries, and **tuple**s. All of them are discussed in the next sections.

For now, let's focus on the **strings**.

How to Create a String in Python

You can create a string in three ways using **single**, **double** or **triple** quotes. Here's an example of every option:

Basic Python String

```
my_string = "Let's Learn Python!"
another_string = 'It may seem difficult first, but you
can do it!'
a_long_string = '''Yes, you can even master multi-line
strings
 that cover more than one line
 with some practice'''
```

IMP! Whichever option you choose, you should stick to it and use it consistently within your program.

As the next step, you can use the **print()** function to output your string in the console window. This lets you review your code and ensure that all functions well.

Here's a snippet for that:

```
print("Let's print out a string!")
```

String Concatenation

The next thing you can master is **concatenation** — a way to add two strings together using the "+" operator. Here's how it's done:

```
string_one = "I'm reading "
string_two = "a new great book!"
string_three = string_one + string_two
```

Note: You can't apply + operator to two different data types e.g. string + integer. If you try to do that, you'll get the following Python error:

```
TypeError: Can't convert 'int' object to str implicitly
```

String Replication

As the name implies, this command lets you repeat the same string several times. This is done using * operator. Mind that this operator acts as a replicator only with string data types. When applied to numbers, it acts as a multiplier.

String replication example:

```
'Alice' * 5 'AliceAliceAliceAliceAlice'
```

And with print ()

```
print("Alice" * 5)
```

And your output will be Alice written five times in a row.

Math Operators

For reference, here's a list of other math operations you can apply towards numbers:

Operators	Operation	Example
**	Exponent	2 ** 3 = 8
%	Modulus/Remainder	22 % 8 = 6
//	Integer division	22 // 8 = 2
/	Division	22 / 8 = 2.75
*	Multiplication	3 * 3 = 9
-	Subtraction	5 - 2 = 3
+	Addition	2 + 2 = 4

How to Store Strings in Variables

Variables in Python 3 are special symbols that assign a specific storage location to a value that's tied to it. In essence, variables are like special labels that you place on some value to know where it's stored.

Strings incorporate data. So you can "pack" them inside a variable. Doing so makes it easier to work with complex Python programs.

Here's how you can store a string inside a variable.

```
my_str = "Hello World"
```

Let's break it down a bit further:

- my_str is the variable name.
- = is the assignment operator.
- "Just a random string" is a value you tie to the variable name.

Now when you print this out, you receive the string output.

```
print(my_str)
```

= Hello World

See? By using variables, you save yourself heaps of effort as you don't need to retype the complete string every time you want to use it.

Built-in Functions in Python

You already know the most popular function in Python — **print()**. Now let's take a look at its equally popular cousins that are in-built in the platform.

Input() Function

input() function is a simple way to prompt the user for some input (e.g. provide their name). All user input is stored as a string.

Here's a quick snippet to illustrate this:

```
name = input("Hi! What's your name? ")
print("Nice to meet you " + name + "!")

age = input("How old are you ")
print("So, you are already " + str(age) + " years old, "
+ name + "!")
```

When you run this short program, the results will look like this:

```
Hi! What's your name? "Jim"

Nice to meet you, Jim!

How old are you? 25

So, you are already 25 years old, Jim!
```

len() Function

len() function helps you find the length of any string, list, tuple, dictionary, or another data type. It's a handy command to determine excessive values and trim them to optimize the performance of your program.

Here's an input function example for a string:

```
# testing len()
str1 = "Hope you are enjoying our tutorial!"
print("The length of the string is :", len(str1))
```

Output:

```
The length of the string is: 35
```

filter()

Use the **Filter()** function to exclude items in an iterable object (lists, tuples, dictionaries, etc)

```
ages = [5, 12, 17, 18, 24, 32]

def myFunc(x):
  if x < 18:
    return False
  else:
    return True

adults = filter(myFunc, ages)

for x in adults:
  print(x)
```

(Optional: The PDF version of the checklist can also include a full table of all the in-built functions).

How to Define a Function

Apart from using in-built functions, Python 3 also allows you to define your own functions for your program.

To recap, a **function** is a block of coded instructions that perform a certain action. Once properly defined, a function can be reused throughout your program i.e. re-use the same code.

Here's a quick walkthrough explaining how to define a function in Python:

First, use **def** keyword followed by the function **name()**:. The parentheses can contain any parameters that your function should take (or stay empty).

```
def name():
```

Next, you'll need to add a second code line with a 4-space indent to specify what this function should do.

```
def name():
    print("What's your name?")
```

Now, you have to call this function to run the code.

```
name.py
def name():
    print("What's your name?")

name()
```

Now, let's take a look at a defined function with a parameter — an entity, specifying an argument that a function can accept.

```
def add_numbers(x, y, z):
    a = x + y
    b = x + z
    c = y + z
    print(a, b, c)

add_numbers(1, 2, 3)
```

In this case, you pass the number 1 in for the x parameter, 2 in for the y parameter, and 3 in for the z parameter. The program will that do the simple math of adding up the numbers:

Output:

```
a = 3
b = 4
c = 5
```

How to Pass Keyword Arguments to a Function

A function can also accept keyword arguments. In this case, you can use parameters in random order as the Python interpreter will use the provided keywords to match the values to the parameters.

Here's a simple example of how you pass a keyword argument to a function.

```
# Define function with parameters
def product_info (product name, price):
    print(”Product Name: “ + product_name)
    print(”Price: “ + str(price))

# Call function with parameters assigned as above
product_info(”White T-Shirt: “, 15)

# Call function with keyword arguments
product_info(productname=”Jeans“, price=45)
```

Output:

```
Product Name: White T-Shirt
Price: 15
Product Name: Jeans
Price: 45
```

Lists

Lists are another cornerstone data type in Python used to specify an ordered sequence of elements. In short, they help you keep related data together and perform the same operations on several values at once. Unlike strings, lists are mutable (=changeable).

Each value inside a list is called an **item** and these are placed between square brackets.

Example lists

```
my_list = [1, 2, 3]
my_list2 = [“a”, “b”, “c”]
my_list3 = [“4”, d, “book”, 5]
```

Alternatively, you can use **list()** function to do the same:

```
alpha_list = list((“1”, “2”, “3”))
print(alpha_list)
```

How to Add Items to a List

You have two ways to add new items to existing lists.

The first one is using **append()** function:

```
beta_list = [“apple”, “banana”, “orange”]
beta_list.append(“grape”)
print(beta_list)
```

The second option is to **insert()** function to add an item at the specified index:

```
beta_list = [“apple”, “banana”, “orange”]
beta_list.insert(2,“grape”)
print(beta_list)
```

How to Remove an Item from a List

Again, you have several ways to do so. First, you can use **remove()** function:

```
beta_list = [“apple”, “banana”, “orange”]
beta_list.remove(“apple”)
print(beta_list)
```

Secondly, you can use the **pop() function**. If no index is specified, it will remove the last item.

```
beta_list = [“apple”, “banana”, “orange”]
beta_list.pop()
print(beta_list)
```

The last option is to use **del keyword** to remove a specific item:

```
beta_list = [“apple”, “banana”, “orange”]
del beta_list [1]
print(beta_list)
```

P.S. You can also apply del towards the entire list to scrap it.

Combine Two Lists

To mash up two lists use the + operator.

```
my_list = [1, 2, 3]
my_list2 = [“a”, “b”, “c”]
combo_list = my_list + my_list2
combo_list
[1, 2, 3, ‘a’, ‘b’, ‘c’]
```

Create a Nested List

You can also create a list of your lists when you have plenty of them :)

```
my_nested_list = [my_list, my_list2]
my_nested_list
[[1, 2, 3], [‘a’, ‘b’, ‘c’]]
```

Sort a List

Use the **sort()** function to organize all items in your list.

```
alpha_list = [34, 23, 67, 100, 88, 2]
alpha_list.sort()
alpha_list
[2, 23, 34, 67, 88, 100]
```

Slice a List

Now, if you want to call just a few elements from your list (e.g. the first 4 items), you need to specify a range of index numbers separated by a colon [x:y]. Here's an example:

```
alpha_list[0:4]
[2, 23, 34, 67]
```

Change Item Value on Your List

You can easily overwrite a value of one list items:

```
beta_list = [“apple”, “banana”, “orange”]
beta_list[1] = “pear”
print(beta_list)
```

Output:

```
['apple', 'pear', 'cherry']
```

Loop Through the List

Using **for loop** you can multiply the usage of certain items, similarly to what * operator does. Here's an example:

```
for x in range(1,4):
    beta_list += ['fruit']
    print(beta_list)
```

Copy a List

Use the built-in **copy()** function to replicate your data:

```
beta_list = [“apple”, “banana”, “orange”]
beta_list = beta_list.copy()
print(beta_list)
```

Alternatively, you can copy a list with the **list()** method:

```
beta_list = [“apple”, “banana”, “orange”]
beta_list = list (beta_list)
print(beta_list)
```

List Comprehensions

List comprehensions are a handy option for creating lists based on existing lists. When using them you can build by using **strings** and **tuples** as well.

List comprehensions examples

```
list_variable = [x for x in iterable]
```

Here's a more complex example that features math operators, integers, and the **range()** function:

```
number_list = [x ** 2 for x in range(10) if x % 2 == 0]
print(number_list)
```

Tuples

Tuples are similar to lists — they allow you to display an ordered sequence of elements. However, they are immutable and you can't change the values stored in a tuple.

The advantage of using tuples over lists is that the former are slightly faster. So it's a nice way to optimize your code.

How to Create a Tuple

```
my_tuple = (1, 2, 3, 4, 5)
my_tuple[0:3]
(1, 2, 3)
```

Note: Once you create a tuple, you can't add new items to it or change it in any other way!

How to Slide a Tuple

The process is similar to slicing lists.

```
numbers = (0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12)
print(numbers[1:11:2])
```

Output:

```
(1, 3, 5, 7, 9)
```

Convert Tuple to a List

Since Tuples are immutable, you can't change them. What you can do though is convert a tuple into a list, make an edit and then convert it back to a tuple.

Here's how to accomplish this:

```
x = ("apple", "orange", "pear")
y = list(x)
y[1] = "grape"
x = tuple(y)
print(x)
```

Dictionaries

A dictionary holds indexes with keys that are mapped to certain values. These key-value pairs offer a great way of organizing and storing data in Python. They are mutable, meaning you can change the stored information.

A key value can be either a **string**, **Boolean**, or **integer**. Here's an example dictionary illustrating this:

```
Customer 1= {'username': 'john-sea', 'online': false,
'friends':100}
```

How to Create a Python Dictionary

Here's a quick example showcasing how to make an empty dictionary.

Option 1: **`new_dict = {}`**

Option 2: **`other_dict= dict()`**

And you can use the same two approaches to add values to your dictionary:

```
new_dict = {
  "brand": "Honda",
  "model": "Civic",
  "year": 1995
}
print(new_dict)
```

How to Access a Value in a Dictionary

You can access any of the values in your dictionary the following way:

```
x = new_dict["brand"]
```

You can also use the following methods to accomplish the same.

- **dict.keys()** isolates keys
- **dict.values()** isolates values
- **dict.items()** returns items in a list format of (key, value) tuple pairs

Change Item Value

To change one of the items, you need to refer to it by its key name:

```
#Change the "year" to 2020:

new_dict= {
  "brand": "Honda",
  "model": "Civic",
  "year": 1995
}
new_dict["year"] = 2020
```

Loop Through the Dictionary

Again to implement looping, use for loop command.

Note: In this case, the return values are the keys of the dictionary. But, you can also return values using another method.

```
#print all key names in the dictionary

for x in new_dict:
  print(x)

#print all values in the dictionary

for x in new_dict:
  print(new_dict[x])

#loop through both keys and values

for x, y in my_dict.items():
  print(x, y)
```

If Statements (Conditional Statements) in Python

Just like other programming languages, Python supports the basic logical conditions from math:

- Equals: a == b
- Not Equals: a != b
- Less than: a < b
- Less than or equal to a <= b
- Greater than: a > b
- Greater than or equal to: a >= b

You can leverage these conditions in various ways. But most likely, you'll use them in "**if statements**" and **loops.**

If Statement Example

The goal of a conditional statement is to check if it's True or False.

```
if 5 > 1:
        print("That's True!")
```

Output:

```
That's True!
```

Nested If Statements

For more complex operations, you can create nested if statements. Here's how it looks:

```
x = 35

if x > 20:
  print("Above twenty,")
  if x > 30:
    print("and also above 30!")
```

Elif Statements

elif keyword prompts your program to try another condition if the previous one(s) was not true. Here's an example:

```
a = 45
b = 45
if b > a:
  print("b is greater than a")
elif a == b:
  print("a and b are equal")
```

If Else Statements

else keyword helps you add some additional filters to your condition clause. Here's how an if-elif-else combo looks:

```
if age < 4:
ticket_price = 0
elif age < 18:
ticket_price = 10
else: ticket_price = 15
```

If-Not-Statements

Not keyword let's you check for the opposite meaning to verify whether the value is NOT True:

```
new_list = [1, 2, 3, 4]
x = 10
if x not in new_list:
    print("'x' isn't on the list, so this is True!")
```

Pass Statements

If statements can't be empty. But if that's your case, add the **pass** statement to avoid having an error:

```
a = 33
b = 200

if b > a:
  pass
```

Python Loops

Python has two simple loop commands that are good to know:

- for loops
- while loops

Let's take a look at each of these.

For Loop

As already illustrated in the other sections of this Python checklist, **for loop** is a handy way for iterating over a sequence such as a list, tuple, dictionary, string, etc.

Here's an example showing how to loop through a string:

```
for x in “apple”:
  print(x)
```

Plus, you've already seen other examples for lists and dictionaries.

While Loops

While loop enables you to execute a set of statements as long as the condition for them is true.

```
#print as long as x is less than 8

i = 1
while i< 8:
  print(x)
  i += 1
```

How to Break a Loop

You can also stop the loop from running even if the condition is met. For that, use the break statement both in while and for loops:

```
i = 1
while i < 8:
  print(i)
  if i == 4:
    break
  i += 1
```

Class

Since Python is an object-oriented programming language almost every element of it is an **object** — with its methods and properties.

Class acts as a blueprint for creating different objects. **Objects** are an instance of a class, where the class is manifested in some program.

How to Create a Class

Let’s create a class named TestClass, with one property named z:

```
class TestClass:
  z = 5
```

How To Create an Object

As a next step, you can create an object using your class. Here’s how it’s done:

```
p1 = TestClass()
print(p1.x)
```

Further, you can assign different attributes and methods to your object. The example is below:

```
class car(object):
    “””docstring”””

    def __init__(self, color, doors, tires):
        “””Constructor”””
        self.color = color
        self.doors = doors
        self.tires = tires

    def brake(self):
        “””
        Stop the car
        “””
        return “Braking”

    def drive(self):
        “””
        Drive the car
        “””
        return “I’m driving!”
```

How to Create a Subclass

Every object can be further sub-classified. Here's an example

```
class Car(Vehicle):
    “”””
    The Car class
    “”””

    def brake(self):
        “”””
        Override brake method
        “”””
        return “The car class is breaking slowly!”

if __name__ == “__main__”:
    car = Car(“yellow”, 2, 4, “car”)
    car.brake()
    ‘The car class is breaking slowly!’
    car.drive()
    “I’m driving a yellow car!”
```

Dealing with Python Exceptions (Errors)

Python has a list of in-built exceptions (errors) that will pop up whenever you make a mistake in your code. As a newbie, it's good to know how to fix these.

The Most Common Python Exceptions

- `AttributeError` — pops up when an attribute reference or assignment fails.
- `IOError` — emerges when some I/O operation (e.g. an `open()` function) fails for an I/O-related reason, e.g., “file not found” or “disk full”.
- `ImportError` — comes up when an import statement cannot locate the module definition. Also, when a from… import can't find a name that must be imported.
- `IndexError` — emerges when a sequence subscript is out of range.
- `KeyError` — raised when a dictionary key isn't found in the set of existing keys.
- `KeyboardInterrupt` — lights up when the user hits the interrupt key (such as Control-C or Delete).
- `NameError` — shows up when a local or global name can't be found.

- `OSError` — indicated a system-related error.
- `SyntaxError` — pops up when a parser encounters a syntax error.
- `TypeError` — comes up when an operation or function is applied to an object of inappropriate type.
- `ValueError` — raised when a built-in operation/function gets an argument that has the right type but not an appropriate value, and the situation is not described by a more precise exception such as IndexError.
- `ZeroDivisionError` — emerges when the second argument of a division or modulo operation is zero.

How to Troubleshoot the Errors

Python has a useful statement, design just for the purpose of handling exceptions – **try/except** statement. Here's a code snippet showing how you can catch KeyErrors in a dictionary using this statement:

```
my_dict = {“a”:1, “b”:2, “c”:3}
try:
        value = my_dict[“d”]
    except KeyError:
        print(“That key does not exist!”)
```

You can also detect several exceptions at once with a single statement. Here's an example for that:

```
my_dict = {“a”:1, “b”:2, “c”:3}
try:
    value = my_dict[“d”]
except IndexError:
    print(“This index does not exist!”)
except KeyError:
    print(“This key is not in the dictionary!”)
except:
    print(“Some other problem happened!”)
```

try/except with else clause

Adding an else clause will help you confirm that no errors were found:

```
my_dict = {“a”:1, “b”:2, “c”:3}

try:
    value = my_dict[“a”]
except KeyError:
    print(“A KeyError occurred!”)
else:
    print(“No error occurred!”)
```

Conclusions

Now you know the core Python concepts!

By no means is this Python checklist comprehensive. But it includes all the key data types, functions and commands you should learn as a beginner.

As always, we welcome your feedback in the comment section below!

Welcome to Mikey Tech Community!

Community Link :
https://chat.whatsapp.com/JnslGjYOBK38fJpJbAYWhH
Group Link :
https://chat.whatsapp.com/KKBQZotM2kXGWOVjZccm99