Basic and Intermediate Python 3 – Notes/Cheat Sheet

The absolute basics

Comments

```
# from the hash symbol to the end of a line.
```

```
"""A docstring, is the first statement in a
module, function, class, or method. Enclosed in
triple quotes, it describes what the code does."""
```

```
"""Stand alone string literals occurring elsewhere
in Python code are also used for documentation."""
```

Line breaks

- Typically, a Python statement must be on one line.
- Bracketed code – () [] {} – can run across lines.
- Backslash (\) before the newline extends a statement over a line (but bracketed code is better).
- More than one statement on a line is semi-colon (;) separated (one statement per line is best practice).

Hint: Limit all lines to a maximum of 79 characters.

Everything is an object

Objects

Everything is an object in Python. Every entity has data (or attributes) and functionality (methods). For example, all objects have a __doc__ attribute that holds the docstring defined in the source code. Because the number 5 is an instance of the int class, we can see the int class docstring using dot-notation as follows.

```
print((5).__doc__)
print(int.__doc__)  # same result as previous line
```

Note: we bracket the 5 so the interpreter knows we want the 5 instance. Without brackets it is invalid syntax.

dir(*object*) yields a list of all the attributes and methods.

```
print(dir(int))      # from the class identifier
print(dir(5))        # from an instance
x = 5; print(dir(x)) # from an assigned identifier
```

Identifiers

Variables (more accurately identifiers) in Python are not containers or locations in memory. They are references or pointers to an object. Identifiers are assigned and reassigned with = (equals). They are deleted with del.

```
x = "Hello" # x refers to a string object
del x       # removes the reference not the object
```

The Python interpreter can automatically delete an instance when there are no longer any live references to that instance (but it may not, so don't rely on it).

Dynamic typing

Objects are strongly typed. Identifiers are not typed. Identifiers can be created whenever as needed. They can reference differently typed objects without problem.

```
x = "a string"      # x references a string
x = [1, 2, 3]       # now it references a list
x = 19              # now it references an integer
print(type(x))      # <class 'int'>
isinstance(x, int)  # True
```

Immutable and mutable objects

Some objects are immutable (int, float, string, bool, tuple, frozenset). Immutable objects cannot be changed after they have been created. Most objects are mutable (including: list, set, dictionary, numpy arrays, etc.)

Creating your own object types

You can create your own object types by defining a new class (more on this later).

Operators

Arithmetic Operations

```
a + b             # addition
a - b             # subtraction
a * b             # multiplication
a / b             # division
a // b            # floor division (rounds down)
a % b             # modulus
a ** b            # exponentiation
-a                # unary negation
+a                # unary plus (unchanged)
```

Bitwise operators

```
a & b             # bitwise AND
a | b             # bitwise OR
a ^ b             # bitwise XOR
a << b            # left bit shift
a >> b            # right bit shift
~a                # bitwise complement
```

Assignment operators

```
a = 4             # a refers to the int instance 4
a = b             # a now refers to the same
                  # object as the identifier b
a += 2            # assignment operator  a = a + 2
a := b * c        # assignment expressions
```

Note: many assignment ops: -=, +=, /=, *=, %=, //=, **=

Boolean comparisons

```
a == b            # a equals b
a != b            # a not equal to b
a < b             # a less than b
a > b             # a greater than b
a <= b            # a less than or equal to b
a >= b            # a greater than or equal to b
a < b <= c        # range comparisons!
```

Boolean operators: and, or, not

```
(a < 6) and (b > 4)          # logical and
(a < 6) or (b % 2 == 0)      # logical or
not (a < b)                  # logical not
```

Object identity – is (also: is not)

```
x = 1; x is 1       # True – same object
[1, 2] == [1, 2]    # True – same values
[1, 2] is [1, 2]    # False – different objects
```

Membership – in (also: not in)

```
1 in [1, 2, 3]          # True
4 not in [1, 2, 3]      # True
```

Note: this test works for lists, strings, tuples, sets, and dictionaries (where it checks the keys).

Numeric objects (all immutable)

Integers (class: int)

```
x = 1                    # int – integers – any size
```

Any number without a decimal point. Integers in Python 3 are of unlimited size. While Python 2 had a long type for arbitrary sized integers, this no longer exists.

Floating point numbers (class: float)

```
x = 1.0                  # float – real numbers
```

Double precision, 64-bit approximation of real numbers, includes "not a number", infinity and negative infinity.

```
x = float('nan')         # not a number
x = float('inf')         # infinity
x = -float('inf')        # negative infinity
x = float('inf') / float('inf') # not a number
x = 0 * float('inf')     # not a number
```

Comparisons with infinity do what you would expect.

```
6.0 < float('inf')       # True
```

Trap: Floating points don't always behave as expected – issues with fractional decimal approximation in base 2.

```
0.5 + 0.75 == 1.25       # True
0.1 + 0.2 == 0.3         # False
```

Hint: use the Python decimal module for accurate floating-point maths with user alterable precision.

Complex numbers (class: complex)

```
x = 1 + 4j               # complex – complex numbers
```

Complex numbers have a real and imaginary part. The imaginary part is written with a j suffix.

Booleans (class: bool, which is a subclass of int)

```
x = True                 # bool – Boolean True/False
issubclass(bool, int)    # True
```

Two instances of Boolean type: True and False.

Lists (class: list)

List (mutable, indexed, iterable, ordered container)
Lists are Python's array-like built-in type.

```
a = []                       # the empty list
a = ['dog', 'cat', 'bird']   # simple list
a = [[1, 2], ['a', 'b']]     # nested lists
a = [1, 2, 3] + [4, 5, 6]    # list concatenation
a = [1, 2, 3] * 4            # list replication
a = list(other_type)         # conversion
```

Note: list elements can be of different types

Size of lists – use len() function

```
simple_list = [1, 2, 3]
len(simple_list)             # list size is 3
list_wth_sublists = [1, 2, 3, [4, 5, 6]]
len(list_wth_sublists)       # top list size is 4
```

Indexed with integers from 0 to (length-1)

```
my_list = ['dog', 'cat']
print(my_list [0])    # prints 'dog'
my_list [1] = 'bird'  # now ['dog', 'bird']
my_list.append('bat') # now ['dog', 'bird', 'bat']
```

Trap: use .append(item) to extend a list. You can only use assignment to replace existing elements.

Negative indexed from the other end of the list

```
my_list = ['dog', 'cat', 'fish']
print(my_list[-1]) # prints last element 'fish'
print(my_list[-2]) # prints 2nd last element 'cat'
```

Checking list membership: in

```
if 'dog' in my_list:
    print('I found the dog')
if 'bat' not in my_list:
    print('A bat is missing')
```

Iterating lists

```
for item in my_list:
    print(item)
for index, item in enumerate(my_list):
    print(index, item)
```

Key list methods

Method	What it does
l.append(x)	Add x to end of list
l.clear()	Remove all elements from list
l.copy()	Return a shallow copy of the list l
l.count(x)	Count the number of times x is found in the list
l.extend(other)	Append items from other
l.index(x)	Get index of first x occurrence; An error if x not found
l.insert(pos, x)	Insert x at position
l.pop([pos])	Remove last item from list (or item from pos); An error if empty list
l.remove(x)	Remove first occurrence of x; An error if no x
l.reverse(x)	In place list reversal
l.sort()	In place list sort

List slicing
General format

```
x[from_inclusive_first:to_exclusive_last]
x[from_inclusive:to_exclusive:step]
```

If the "from" is not given, it assumed to be zero.
If the "to" is not given, is assumed to be the list length.
If the step is not given, it is assumed to be one.

Examples

```
x = [0, 1, 2, 3, 4, 5, 6, 7, 8] # play data
x[2]     # 3rd element - reference not slice
x[1:3]   # 2nd to 3rd element [1, 2]
x[:3]    # the first three elements [0, 1, 2]
x[-3:]   # last three elements
x[:-3]   # all but the last three elements
x[:]     # every element of x – copies x
x[1:-1]  # all but first and last element
x[::3]   # [0, 3, 6] 1st then every 3rd
x[1:5:2] # [1, 3] start 1, stop >= 5, by every 2nd
x[::-1]  # every element of x in reverse order
```

Note: All Python sequence types support the above index slicing (strings, lists, tuples, bytearrays, buffers)

Deleting elements from a list

```
x.remove('fish') # remove first occurrence from x
y = x.pop()      # return and remove last element
del x[2]         # remove third element from x
del x[-3:]       # delete the last three elements
```

Tuples (class: tuple)

Tuples (immutable, indexed, ordered container)
Tuples are immutable lists. They can be searched, indexed, sliced and iterated much like lists. List methods that do not change the list also work on tuples.

```
a = ()                      # the empty tuple
a = (1,)  # note comma      # one item tuple
a = (1, 2, 3)               # multi-item tuple
a = ((1, 2), (3, 4))        # nested tuple
a = tuple(['a', 'b'])       # conversion
```

Note: the comma is the tuple constructor, not the parentheses. The parentheses, arguably, add clarity.

```
a = 1, 2, 3                 # this is also a tuple
a = 1,                      # this is a tuple too
```

Tuple packing and unpacking

```
a = 1, 2, 3                 # tuple packing
x, y, z = a                 # tuple unpacking
print(x, y, z)              # 1 2 3
```

Unpacking with * (the unpacking operator)

```
a, *b, c = (1, 2, 3, 4, 5) # a=1, b=[2,3,4] c=5
a, b, *c = [1, 2, 3, 4, 5] # a=1, b=2 c=[3,4,5]
f = [1, 2, *[3, 4, 5]]     # f = [1, 2, 3, 4, 5]
f = (1, 2, *(3, 4, 5))     # f = (1, 2, 3, 4, 5)
func(*[a, b, 3])           # same as func(a, b, 3)
*b, = [1, 2] # note comma  # same as b = [1, 2]
print(*sequence, sep=', ')
```

Note: tuple unpacking works with all iterables now days.

The Python swap identifier idiom

```
a, b = b, a    # no need for a temp variable
```

This uses tuple packing/unpacking to achieve its magic.

The Python underscore (_) idiom
By convention, unnecessary values when unpacking a tuple are assigned to the _ identifier. This idiom is often used with functions that return tuples.

```
_, u, v = some_function_that_returns_a_tuple()
_, x, _ = another_function_returning_a_tuple()
```

Strings (class: str)

String (immutable, ordered, iterable, characters)
A Python string (str) is an immutable list of characters, stored in Unicode. [**Note**: There is also a bytes type.]

```
s = 'string'.upper()          # 'STRING'
s = "fred" + "was" + "here"   # plus concatenation
s = 'fred' 'was' 'here'       # space concatenation
s = "spam" * 3                # replication
s = str(x)                    # conversion
```

Note: 'single' or "double" quotes. Multiline strings in """these triple quotes""" or '''these'''. Escape sequences: "\n" is newline; "\t" is tab; "\r" is return; "\\" for backslash; "\"" for quotes (but " ' " or ' " ' is often easier to read).

String literal prefixes (b, f, r, u)

```
bytes_data = b'ascii bytes'   # bytes type not str
format_str = f'{5 - 7}'       # '-2'
raw_str =    r'raws have different \ escape rules'
unicde_str = u'encoded as Unicode' ' so is this'
```

Note: Upper- or lower-case prefixes allowed.

Iterating/searching strings

```
for character in 'str': pass
for index, character in enumerate('str'): pass
if 'red' in 'Fred': print ('Fred is red')
```

String methods (not a complete list)
capitalize, center, count, decode, encode, endswith, expandtabs, find, format, index, isalnum, isalpha, isdigit, islower, isspace, istitle, isupper, join, ljust, lower, lstrip, partition, replace, rfind, rindex, rjust, rpartition, rsplit, rstrip, split, splitlines, startswith, strip, swapcase, title, translate, upper, zfill

String constants (not a complete list)

```
from string import *  # global import is not good
print ([digits, hexdigits, ascii_letters,
    ascii_lowercase, ascii_uppercase,
    punctuation])
```

Printing numbers and identifiers
As Python has evolved, the way in which numbers and identifiers are printed has changed a few times.
In Python 3.6 a new and much better approach known as f-strings (formatted strings) was adopted.
Hint: Use the f-strings approach.

Examples – old string formatting (using % operator)

```
print ("It %s %d times" % ('occurred', 5))
# prints: 'It occurred 5 times'
import math
'%f' % math.pi        # '3.141593'
'%.2f' % math.pi      # '3.14'
'%.2e' % 3000         # '3.00e+03'
```

Examples – string formatting (using format method)

```
import math
'Hello {}'.format('World')   # 'Hello World'
'{}'.format(math.pi)         # ' 3.14159265359'
'{0:.2f}'.format(math.pi)    # '3.14'
'{0:+.2f}'.format(5)         # '+5.00'
'{:.2e}'.format(3000)        # '3.00e+03'
'{:0>3d}'.format(5)          # '005' (left pad)
'{1}{0}'.format('a', 'b')    # 'ba'
'{num:}'.format(num=7)       # '7' (named args)
```

Now everyone uses f-strings to format strings

```
f'Hello {"World"}'.         # 'Hello World'
my_name = 'Mark'
f'Hello {my_name}'          # 'Hello Mark'
f'{my_name.lower()}'        # 'mark'
f'{"right":->10}'           # '-----right'
f'{"left":-<10}'            # 'left------'
f'{"centre":-^10}'          # '--centre--'
import math
f'π={math.pi}'              # 'π=3.141592653589793'
f'π={math.pi:0.3f}'         # 'π=3.142'
f'π={math.pi:0.2e}'         # 'π=3.14e+00'
f'{1000000:,}'              # '1,000,000'
f'{1000000:_.2f}'           # '1_000_000.00'
f'{0.251342:.1%}'           # '25.1%'
f'{365:+}'                  # '+365'
f'{-365:+}'                 # '-365'
f'Route {37 + 51 - 22}'     # 'Route 66'
f'Leading zeros {12:0>5}'   # 'Leading zeros 00012'
f'Like above {-12:+05}'     # 'Like above -0012'
f'{6:b}'                    # '110'    # binary
f'{127:o}'                  # '177'.   # octal
f'{60000:x}'                # 'ea60'   # hex
f'{66:c}'                   # 'B'.     # character
```

Indexing and slicing strings

Strings are a list of characters. They can be indexed and sliced in the same way as other Python lists.

```
s = 'Alphabet soup'
s[0]                          # 'A'
s[-1]                         # 'p'
s[:5]                         # 'Alpha'
s[-4:]                        # 'soup'
s[::-1]                       # 'puos tebahplA'
```

Dictionaries (class: dict)

Dictionary (indexed, ordered, map-container)

A dict is the Python hash-map or associative array type. It is a mutable hash map of unique key/value pairs. Prior to Python 3.7, dictionaries were unordered. From Python 3.7, the dictionary keys are maintained and returned in insertion order.

Create a dictionary

```
a = {}                        # empty dictionary
a = {1: 1, 2: 4, 3: 9}        # simple dict
a = dict(x)                   # convert paired data

# Create from a list
l = ['alpha', 'beta', 'gamma', 'delta']
a = dict(zip(range(len(l)), l))

# Create from a comma separated paired string
s = 'a=apple,b=bird,c=cat,d=dog,e=egg'
a = dict(i.split("=") for i in s.split(","))
# {'a': 'apple', 'c': 'cat', 'b': 'bird',
#  'e': 'egg', 'd': 'dog'}
```

Note: dictionary keys must be of an immutable type. They do not need to be all of the same type.

Add a key/value pair to a dictionary

```
d['key'] = 'value'
```

Retrieve value using a key

```
x = d['key']     # exception raised if key missing
x = d.get('key', 'default value if key missing')
```

Change value using a key

```
d['key'] = 'new value'
```

Delete a key/value pair from a dictionary

```
x = d.pop('key')
del d['key']
```

Iterating a dictionary

```
for key in dictionary:
    print (key)
for key in dictionary.keys():
    print(key)
for key, value in dictionary.items():
    print (key, value)
for value in dictionary.values():
    print(value)
```

Searching a dictionary

```
if key in dictionary:
    print (key)
if value in dictionary.values():
    print(f'{value} found')
```

The size of a dictionary

```
size = len(d)
```

Dictionary methods (not a complete list)

Method	What it does
d.clear()	Remove all items from d
d.copy()	Shallow copy of dictionary
d.get(key[, def])	Get value else default
d.items()	Dictionary's (k,v) pairs
d.keys()	Dictionary's keys
d.pop(key[, def])	Get value else default; remove key from dictionary
d.popitem()	Remove and return the last (k, v) pair from a dictionary
d.setdefault(k[,def]))	If k in dict return its value otherwise set def
d.update(other_d)	Update d with key:val pairs from other
d.values()	The values from dict

Dictionary unpacking

Dictionaries can be unpacked with ** to key=value pairs when functions are called.

```
d = {'a': 1, 'b': 2, 'c': 3}
s = some_function(**d)  # is equivalent to ...
s = some_function(a=1, b=2, c=3)
```

The keys of a dictionary can be unpacked to a list, etc.

```
key_list = [*d]
key_tuple = *d,
key_set = {*d}
```

The key/value pairs of a dictionary can be unpacked into another dictionary.

```
expanded = {99: 'more', **d, 100:'and another'}
```

Merging two or more dictionaries

```
merged = a.update(b)
merged = {**a, **b, **c}    # dictionary unpacking
merged = a | b              # from Python 3.8
```

Trap: if the second dictionary has keys in common with the first dictionary, these key/value pairs will overwrite those from the first dictionary.

Sets (class: set)

Set (unique, unordered container)

A Python set is an unordered, mutable collection of unique hashable objects.

```
a = set()                     # empty set
a = {'red', 'white', 'blue'} # simple set
a = set(x)                    # convert to set
```

Trap: {} creates an empty dict, not an empty set

Iterating a set

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

Searching a set

```
if item in set:
    print(item)
if item not in set:
    print(f'{item} is missing from our set')
```

The size of a set

```
size = len(s)
```

Set methods (not a complete list)

Method	What it does
s.add(item)	Add item to set
s.remove(item)	Remove item from set. Raise KeyError if item not found.
s.discard(item)	Remove item from set if present.
s.pop()	Remove and return an arbitrary item. Raise KeyError on empty set.
s.clear()	Remove all items from set
s.copy()	Get shallow copy of set
s.isdisjoint(o)	True if s has not items in common with other set o
s.issubset(o)	Same as set <= other
s.issuperset(o)	Same as set >= other
s.union(o[, ...])	Return new union set
s.intersection(o)	Return new intersection
s.difference(o)	Get net set of items in s but not others (Same as set – other)

Frozenset (class: frozenset)
Similar to a Python set above, but a frozenset is immutable (and therefore hashable). It can be used as a dictionary key.

```
f = frozenset(s)            # convert set
f = frozenset(o)            # convert other
```

Other collections

Collections module
Beyond the built-in collections, there are many more data types that can be imported. From the collections module you can import (for example): Counter, deque (double ended queue) and namedtuple.

Program flow control

Code blocks
A code block (the body of a function, a loop, etc.) starts with indentation and ends with the first unindented line. The indents are always four spaces. Never use tabs.
Hint: set your editor to replace tabs with four spaces.

Assert – or die trying – for development code only

```
assert x > 0                # ensure a condition
assert False                # always fails
if not x > 0:
    sys.exit(s)             # production safe
```

If the conditional fails, assert raises an AssertionError
Trap: assert statements are ignored in optimised mode.
Hint: If program termination is required in production code, import the sys module and call sys.exit().

Ternary statements

```
x = y if a > b else z
r = a if x in y else b if u == v else c # nested
z = (func1 if x > 6 else func2)(arg1, arg2) # wow
```

If - flow control

```
if condition:   # for example: if x < 5:
    statements
elif condition: # optional – and can be multiple
    statements
else:           # optional
    statements
```

Hint: multiple nested if/elif statements can be hard to read. There is almost always a better way to code these beasts (for example a dictionary lookup table).

For – flow control

```
for x in iterable:
    statements
    if conditional:
        continue # go back to the start
    if conditional:
        break # exit the loop
else:           # optional completion code
    statements
```

Trap: break skips the else completion code
Useful: pass is a statement that does nothing

Common for-loop patterns

```
for i in range(0, 10): pass
for i, value in enumerate(list_of_items): pass
for a, b in zip(first_list, second_list): pass
for x, y, z in zip(list1, list2, list3): pass
for element in set_: pass
for key in dictionary: pass
for key in dictionary.keys(): pass # same as above
for value in dictionary.values(): pass
for key, value in dictionary.items(): pass
```

Hint: for-loops are often not the best solution in python.
Trap: The *for i in range(len(x)):* pattern is particularly pernicious. Some consider it a code-smell.

Rather than a for-loop, think about using a list comprehension, a generator expression, a dictionary comprehension, a set comprehension, or even using the map() function. More to come on these options.

```
u = [do_something_with(i) for i in list_of_items]
v = (do_something_with(i) for i in list_of_items)
w = {i: function(i) for i in iterable}
s = {exp_with_i for i in iterable if condition}
z = map(function, list_items)
```

While – flow control

```
while condition:
    statements
    # break and continue can be used here too
else:           # optional completion code
    statements
```

Common while-loop patterns

```
while container:    # an empty container is False
    element = container.pop()
    # do something with this element
```

Exceptions – flow control

```
try:
    statements
except (tuple_of_errors): # can be multiple
    statements
else:           # optional no exceptions
    statements
finally:        # optional all
    statements
```

Raising exceptions

```
raise TypeError('function expects a string '
                f'but it got a {type(x)}')
```

Creating new errors

```
class MyError(Exception):
    def __init__(self, value):
        self.value = value
    def __str__(self):
        return repr(self.value)
```

Common exceptions (not a complete list)

Exception	Why it happens
AsserionError	Assert statement failed
AttributeError	Class attribute assignment or reference failed
IOError	Failed I/O operation
ImportError	Failed module import
IndexError	Subscript out of range
KeyError	Dictionary key not found
NameError	Name not found
TypeError	Value of the wrong type
ValueError	Right type but wrong value

Hint: avoid creating your own subclassed exceptions. Try and use an existing exception if at all possible.

With – using a context manager

Some classes have been written to return a context manager that handles exceptions behind the scene and free-up or close a resource when it is finished.

General form

```
with object_that_returns_a_cm() [as identifier]:
    do_something(identifier)
    # when done, close/free-up the resource
```

File IO is a good example.

```
with open("test.txt", 'w', encoding='utf-8') as f:
   f.write("This is an example\n")
   # when we exit the with code block,
   # the file will be closed automatically.
```

Classes that are context managers will have implemented __enter__() and __exit__() methods.

Truthiness

Truthiness

Many Python objects are said to be "truthy", with an inbuilt notion of "truth".

False	True
None	
0 **int(False) # → 0**	Any number other than 0 int(True) # → 1
"" **# the empty string**	' ', 'fred', "False", 'None', '0', '[]' # all other strings
() [] {} set() **# empty containers**	[None] (False,) [0] [""] # non-empty containers, including those containing False, None or zero.

Trap: Not all objects follow this convention. For example, numpy arrays, and pandas DataFrames and Series do not follow the empty container convention.

It is pythonic to use the truth of container objects.

```
if container:                # test not empty
    # do something

while container:             # common looping idiom
    item = container.pop()
    # process item
```

Built-in functions

Key built-in functions include … (not a complete list)

Function	What it does
abs(num)	Absolute value of num
all(iterable)	True if all are True
any(iterable)	True if any are True
assert(condition)	Throw an error if condition fails
bytearray(source)	A mutable array of bytes
bool(obj)	Get the truthiness of the object
callable(obj)	True if obj is callable
chr(int)	Character for ASCII int
complex(re[, im])	Create a complex number
divmod(a, b)	Get (quotient, remainder)
enumerate(seq)	Get an enumerate object, with next() method returns an (index, element) tuple
eval(string)	Evaluate an expression
filter(fn, iter)	Construct a list of elements from *iter* for which *fn()* returns True
float(x)	Convert from int/string
getattr(obj, str)	Like obj.str
hasattr(obj, str)	True if obj has attribute
hex(x)	From int to hex string
id(obj)	Return unique (run-time) identifier for an object
int(x)	Convert from float/string
isinstance(o, c)	Eg. isinstance(2.1, float)
len(object)	Number of items in x; x is string, tuple, list, dict
list(iterable)	Make a list
long(x)	Convert a string or number to a long integer
map(fn, iterable)	Apply fn() to every item in iterable; return results in a list
max(a, b) **max(iterable)**	What it says on the tin
min(a, b) **min(iterable)**	Ditto
next(iterator)	Get next item from an iter
open(name[,mode])	Open a file object
ord(character)	Opposite of chr(int)
pow(x, y)	Same as x ** y
print (objects)	What it says on the tin
range(stop) **range(start, stop)** **range(fr, to, step)**	integer list; stops < stop default start=0; default step=1
reduce(fn, iterator)	Applies the two argument fn(x, y) cumulatively to the items of iter.
repr(object)	Printable representation of an object
reversed(seq)	Get a reversed iterator

Function	What it does
round(n[, digits])	Round to number of digits after the decimal place
setattr(obj, n, v)	Like obj.n = v #name/value
sorted(iterable)	Get new sorted list
str(object)	Get a string for an object
sum(iterable)	Sum list of numbers
type(object)	Get the type of object
zip(x, y[, z])	Return a list of tuples

Importing modules

Modules
Modules open up a world of Python extensions. Access to the functions, identifiers and classes of a module depends on how the module was imported.

```
import math                 # math.cos(math.pi/3)
import math as m            # m.cos(m.pi/3)
from math import cos, pi    # cos(pi/3)
from math import *          # log(e)
```

Hint: It is best to avoid global imports (last line above).

There are hundreds of thousands of python packages available for import. Frequently used packages include:

- os – operating system interface
- sys – system parameters and functions
- pathlib – file system interface
- datetime – for dates and times
- re – for regular expressions / pattern matching
- math – for maths
- requests – access the internet
- scrapy, selenium – web crawling/web scraping
- beautifulsoup4 – web scraping
- flask – lite-weight web server
- scipy – linear algebra and statistics
- statsmodels – classical statistical models
- PyStan, PyMC3, ArviZ – Bayesian models
- numpy – for linear algebra (import as np)
- pandas – for data manipulation (import as pd)
- matplotlib.pyplt – for charts and graphs (as plt)
- SQLAlchemy – database access

Note: you may need to install a module package on your system before you can use it in Python. From the operating system command line you can use pip or conda, depending on how your system was set-up.

Writing you own functions (basic)

Boilerplate code for a simple function

```
def f(arg1: type1, arg2: type2 = default)-> rtype:
    """Docstring – optional – high level, plain
       English explanation of what the function
       does – the parameters it takes – and what
       it returns."""
    statements
    return return_value # returns None by default
```

Note: functions are first class objects. They have attributes and they can be referenced by identifiers.
Note: positional arguments before named arguments.

Default arguments
Default arguments can be specified in a function definition with a key=value pair. If an augment which has been declared with a default value is not passed to the function, the function will use the default argument.

```
def funct(arg1, arg2=None): pass
```

In this case, arg2 has the default value of None.
Note: default arguments are optional in the function call. Positional arguments must be passed in the call.

Avoid mutable default arguments
Expressions in default arguments are evaluated when the function is defined, not when it's called. Changes to mutable default arguments survive between calls.

```
def nasty(value=[]):       # <-- mutable arg
    value.append('a')
    return value
print(nasty ())            # --> ['a']
print(nasty ())            # --> ['a', 'a']

def better(val=None):      # <-- immutable arg
    val = [] if val is None else val
    value.append('a')
    return value
```

Type hints or type annotations
From Python 3.5, functions may be annotated with the expected types for the parameters and the return value. Identifiers can also be type-hinted at creation. There are lots of abstractions in the typing module you can use.

```
from typing import List, Tuple
Vector = List[float]   # a type alias you can use

def combo5(strings: List[str])-> Tuple[int, str]:
    number5: int = 5   # hint for an identifier
    return number5, ', '.join(strings)
```

The Python interpreter ignores these hints. They are **not enforced**. But they help document your code.
They can be used by external type checkers (eg. mypy).

Hint: As a rule-of-thumb, you do not need docstrings nor type-hints for short throw away scripts. But for code others will read/use, for packages, or where unit tests are necessary, they should be included in your code.

Lambda (inline expression) functions
Lambdas are small anonymous functions. They are sometimes used for brevity when you would pass a function argument to a function or method.

For example,

```
def is_divisible_by_three(x):
    return x % 3 == 0
div3 = filter(is_divisible_by_three, range(1, 10))
```

becomes,

```
div3 = filter(lambda x: x % 3 == 0, range(1, 10))
```

Lambdas are typically used with the functions filter(), map() and reduce(); and with the pandas methods Series.apply(), DataFrame.groupby().agg(), and the like.

Hint: Lambdas should be used sparingly. They can result in very hard to read code.
Hint: Assigning a lambda to an identifier is a code smell. Use *def* for named functions (it is better for debugging).

Writing you own functions (intermediate)

Function argument unpacking (*args and **kwargs)
*args and will match all the positional arguments and **kwargs will match all the keyword arguments to a function, that were not explicit in the function definition.

```
def my_args(arg1, *args, fish='dead', **kwargs):
    print(args, kwargs)

my_args(1, 'a', u=15, garbage=22)
# prints: ('a',) {'u': 15, 'garbage': 22}
my_args(1, 'a', u=15, fish=5, garbage=22)
# prints: ('a',) {'u': 15, 'garbage': 22}
```

Note: while "args" and "kwargs" are used by convention, they are just identifier names; they are not special names. In the next example the identifier "numbers" captures all of the positional calling arguments into a list (regardless of the number of positional arguments).

```
def my_sum(*numbers):
    return sum(numbers) # sum() takes an iterable
print(my_sum(1, 2, 3, 4, 5.2))     # prints 15.2
print(my_sum(1, 2, 3, 4, 5, 6+3j)) # prints 21+3j
```

Closures
Closures are functions that have inner functions (or inner classes) with data fixed in the inner function (or class) by the lexical scope of the outer function. They allow for code reuse, with similar but different functions (or classes) being created as needed. They are useful for avoiding hard constants in the function call.

Wikipedia has an example derivative "function factory" for any function (f) and value of Δx, using a closure.

```
from typing import Callable

def derivative(f: Callable, Δx: float)-> Callable:
    """Return a function that approximates
       the derivative of f using an interval
       of Δx, which is appropriately small."""
    def f_dash_at_x(x: float)-> float:
        return (f(x + Δx) - f(x)) / Δx
    return f_dash_at_x   # from derivative(f, Δx)

f_dash_x_cube = derivative(lambda x: x**3, 1.0e-7)
f_dash_x_cube(10) # approx 300 (as f'(x) = 3x**2)
```

Identifier scope

Accessible identifiers (LEGB)
Not all identifiers are accessible from all parts of a program. For example, identifiers declared within a function are only visible within that function, from the point at which it is defined until the end of the function.

When you access an identifier, Python looks for it locally, then within the enclosing scope, then the global scope of the module/file and finally the library of built-in identifiers (LEGB: local, enclosing, global, built-in).

Note: The local scope only exists within functions. The enclosing scope only exists for functions defined within functions. (Closures use this enclosing scope).

Trap: your own functions and identifiers will hide the built-ins, if they have the same name.

Trap: Also, if you assign an identifier within a function, Python assumes that it is a local identifier. If you have a global identifier with the same name it will be hidden (unless you make it explicit with the global keyword).

```
def some_function():
    global the_global_identifier
    the_global_identifier = 5
```

Hint: modifying a global identifier from inside a function is usually bad practice. It is a code smell.

Comprehensions, iterators and generators

List comprehensions (can be nested)

```
t3 = [x*3 for x in [5, 6, 7]] # [15, 18, 21]
z = [complex(x, y)
        for x in range(0, 4, 1)
            for y in range(4, 0, -1)
                if x > y]
# z --> [(2+1j), (3+2j), (3+1j)]
```

Set comprehensions

```
# a set of selected letters...
s = {e for e in 'ABCHJADC' if e not in 'AB'}
# --> {'H', 'C', 'J', 'D'}

# a set of tuples ...
s = {(x, y) for x in range(-1, 2)
                for y in range (-1, 2)}
```

Dictionary comprehensions
Conceptually like list comprehensions; but it constructs a dictionary rather than a list

```
a = {n: n*n for n in range(7)}
# a -> {0:0, 1:1, 2:4, 3:9, 4:16, 5:25, 6:36}

odd_sq = {n: n * n for n in range(7) if n % 2}
# odd_sq -> {1: 1, 3: 9, 5: 25}

# next example -> swaps the key:value pairs, but
# risks information loss with non-unique values
b = {val: key for key, val in odd_sq.items()}
```

An iterable object
The contents of an iterable object can be selected one at a time. Strings, lists, tuples, dictionaries, and sets are all iterable objects. Indeed, any object with the magic method __iter__(), which returns an iterator.

An iterable object will produce a fresh iterator with each call to iter().

```
iterator = iter(iterable_object)
```

Iterators
Objects with a next() or __next__() method, that:

- returns the next value in the iteration
- updates the internal note of the next value
- raises a StopIteration exception when done

Note: with the loop `for x in y:` if y is not an iterator; Python calls iter() to get one. With each loop, it calls next() on the iterator until a StopIteration exception.

```
x = iter('XY')       # iterate a string by hand
print(next(x))       #  X
print(next(x))       #  Y
print(next(x))       #  StopIteration exception
```

Generators

Generator functions are resumable functions that work like iterators. They can be more space or time efficient than iterating over a list, (especially a very large list), as they only produce items as they are needed.

```
def fib(max=None):
    """ generator for Fibonacci sequence"""
    a, b = 0, 1
    while max is None or b <= max:
        yield b    # ← yield is like return
        a, b = b, a+b

[i for i in fib(10)] #  [1, 1, 2, 3, 5, 8]
```

Note: a return statement (or getting to the end of the function) ends the iteration.
Trap: a yield statement is not allowed in a try clause.

Messaging the generator

```
def resetableCounter(max=None):
    j = 0
    while max is None or j <= max:
        x = yield j # ← x gets the sent arg
        j = j + 1 if x is None else x

counter = resetableCounter(10)
print(counter.send(None))      #  0
print(counter.send(5))         #  5
print(counter.send(None))      #  6
print(counter.send(11))        #  StopIteration
```

Note: must send None on first send() call

Generator expressions

Generator expressions build generators, just like building a list from a comprehension. You can turn a list comprehension into a generator expression simply by replacing the square brackets [] with parentheses ().

```
[i for i in range(10)]      # list comprehension
(i for i in range(10))      # generator expression
```

Hint: if you want to see the sequence produced by a generator expression, convert it to a list.

```
gen_exp = (x**2 for x in range(16) if not x % 5)
print(list(gen_exp))  # prints [0, 25, 100, 225]
```

Classes

Inheritance

```
class DerivedClass1(BaseClass):
    statements
class DerivedClass2(module_name.BaseClass):
    statements
```

Multiple inheritance

```
class DerivedClass(Base1, Base2, Base3):
    statements
```

Classes

Python has a multiple inheritance class mechanism that encapsulates program code and data. Example follows:

```
import math

class Point:
    count = 0  # static class variable

    def __init__(self, x, y):
        """Instantiate with cartesian co-ords."""
        self.x = float(x)  # instance variable x
        self.y = float(y)  # instance variable y
        Point.count += 1

    def __str__(self):
        return f'(x={self.x}, y={self.y})'

    def to_polar(self):
        """Return tuple with polar co-ords."""
        r = math.sqrt(self.x**2 + self.y**2)
        θ = math.atan2(self.y, self.x)
        return r, θ

my_point = Point(1, 2)
print(my_point)           # uses __str__() method
print(Point(0, 0).to_polar())        # (0.0, 0.0)
print(Point.count)        # prints 2
```

Methods and attributes

Most objects have associated functions or “methods” that are called using dot syntax:

```
object.method(*arguments, **keyword_arguments)
```

Objects also often have attributes or values that are directly accessed without using getters and setters (most unlike Java or C++)

```
instance = Example_Class()
print (instance.attribute)
```

The self

Class methods have an extra argument over functions. Usually named 'self'; it is a reference to the instance. It is not used in the method call; and is provided by Python to the method. Self is like 'this' in C++ & Java

Public and private methods and variables

Python does not enforce the public v private data distinction. By convention, identifiers and methods that begin with an underscore should be treated as private (unless you really know what you are doing). Identifiers that begin with double underscore are mangled by the compiler (and hence more private).

Magic class methods (not a complete list)

Magic methods begin and end with double underscores, and they are known as dunders (or more formally as the Python data model). They add functionality to your classes consistent with the broader language.

Magic method	What it does
__init__(self,[...])	Constructor
__del__(self)	Destructor pre-garbage collection
__str__(self)	Human readable string for class contents. Called by str(self)

Magic method	What it does
__repr__(self)	Machine readable unambiguous Python string expression for class contents. Called by repr(self) Note: str(self) will call __repr__ if __str__ is not defined.
__eq__(self, other)	Behaviour for ==
__ne__(self, other)	Behaviour for !=
__lt__(self, other)	Behaviour for <
__gt__(self, other)	Behaviour for >
__le__(self, other)	Behaviour for <=
__ge__(self, other)	Behaviour for >=
__add__(self, other)	Behaviour for +
__sub__(self, other)	Behaviour for -
__mul__(self, other)	Behaviour for *
__div__(self, other)	Behaviour for /
__mod__(self, other)	Behaviour for %
__pow__(self, other)	Behaviour for **
__pos__(self, other)	Behaviour for unary +
__neg__(self, other)	Behaviour for unary -
__hash__(self)	Returns an int when hash() called. Allows class instance to be put in a dictionary
__len__(self)	Length of container
__contains__(self, i)	Behaviour for in and not in operators
__missing__(self, i)	What to do when dict key i is missing
__copy__(self)	Shallow copy constructor
__deepcopy__(self, memodict={})	Deep copy constructor
__iter__(self)	Provide an iterator
__nonzero__(self)	Called by bool(self)
__index__(self)	Called by x[self]
__setattr__(self, name, val)	Called by self.name = val
__getattribute__(self, name)	Called by self.name
__getattr__(self, name)	Called when self.name does not exist
__delattr__(self, name)	Called by del self.name
__getitem__(self, key)	Called by self[key]
__setitem__(self, key, val)	Called by self[key] = val
__delitem__(self, key)	del self[key]

Decorators

Decorators
Decorators are a syntactic convenience that allows a Python source file to say what it is going to do with the result of a function or a class statement before rather than after the statement. They are callable objects that return a callable.

Decorators allow you to augment a function call and/or the return process from a function with additional code. You can write a decorator to (for example):

- validate the input values to a function;
- count function calls
- log to file the calls to a function
- caching the return values from a function
- see how long a function takes to run (below)
- call a function repeatedly (bottom this column)

```
from functools import wraps
import time

def timer(func):

    @wraps(func)           # this line is optional
    def inner_timer(*args, **kwargs):
        start = time.time()               # before
        ret_val = func(*args, **kwargs)   # call
        secs = time.time() - start        # after
        print(f'This took {secs:.1f} seconds.')
        return ret_val

    return inner_timer
```

Note: the @wraps decorator from functools allows us to see the name, docstring and arguments of the passed function (func). Without @wraps, the docstring, name, and so would come from the inner_timer function.

You would then use this decorator as follows

```
@timer
def slow_function():
    return {x: x**5 for x in range(10_000_000)}

x = slow_function()
# prints: 'This took 2.9 seconds.'
```

The above code block is an easier-to-read version of the following block.

```
def slow_function ():
    return {x: x**5 for x in range(10_000_000)}

timed_slow_function = timer(slow_function)
x = timed_slow_function()
# prints: 'This took 2.9 seconds.'
```

Decorators with parentheses
Decorators with parentheses are callable objects that return a callable that returns a callable. They allow us to pass arguments to the decorator (see example below).

```
from typing import Callable
from functools import wraps

def repeat(n: int)-> Callable:
    def rep_decorator(func: Callable)-> Callable:
        @wraps(func)
        def inner(*args, **kwargs):
            for _ in range(n):
                val = func(*args, **kwargs)
            return val # only last one returned
        return inner        # from rep_decorator()
    return rep_decorator    # from repeat()

@repeat(3)
def example():
    print('One line')

example() # is called 3 times
```

Note: more than one decorator can be applied to a function. They are simply stacked above the function. Stacked decorators are executed from bottom to top.

Built-in decorators
Python has many useful built-in decorators, including:

- The @classmethod and @staticmethod decorators transform a method into a class level function.
- The @property decorator is used to customize getters and setters for class attributes
- The @dataclass method builds boiler-plate code for classes that primarily hold data.

```
from dataclasses import dataclass
```

Advanced Python

These notes are deliberately aimed at new and intermediate programmers to Python. There are a host of advanced Python features that you may wish to learn, once you have progressed beyond these notes. Advanced Python topics include:

- the Python data model
- abstract classes/methods and interfaces in Python
- metaclasses and the type type
- namespaces in Python
- writing modules and packages
- writing context managers
- using Cython to speed up Python
- calling C, C++, R, or Java from Python
- multi-threaded and multi-processor Python

See also …

You might also want to read:

- PEP 8 – Style Guide for Python Code – https://www.python.org/dev/peps/pep-0008/
- PEP 20 – The Zen of Python – https://www.python.org/dev/peps/pep-0020/
- PEP 257 – Docstring Conventions – https://www.python.org/dev/peps/pep-0257/
- PEP 484 – Type Hints – https://www.python.org/dev/peps/pep-0484/