Drop of knowledges of Python

Yan Zhao

Chapter 1

Basic Syntax

Aspect	**C++**	**Python**
Design Style	Generic & Performance-focused	Object-oriented & Readable
Sorting	Algorithm outside container	Method inside container
Flexibility	High control with iterators	High simplicity with methods
Typing System	Statically typed	Dynamically typed
Usage Focus	System-level, fine-grained control	High-level, developer-friendly

Table 1.1: Philosophical Differences Between C++ and Python

1.1 built-in type

All built-in types are:

- Numeric: int, float, complex
- Sequence: str, list, tuple, range. Range is not a function, but a class.
- set and dict
- Binary type: bytes, bytearray and memoryview
  item None type

Immutable sequenece: tuple, str and bytes. mutable sequences: list, bytearray, array.array, collections.deque,, and memoryview.

Parallel assignment (also called multiple assignment or unpacking assignment) in Python lets you assign multiple variables at once in a single line, using tuple/list unpacking.

```
x, y, z = 1, 2, 3

a, b = 5, 10
a, b = b, a
```

```
5
6 coords = (3, 4)
7 x, y = coords
8
9 a, *b = [1, 2, 3, 4]
10 print(a)  # 1
11 print(b)  # [2, 3, 4] and b is list
```

Container sequences are list, tuple, and collections.deque. They can hold different type. Flat sequences are: str, bytes, bytearray, memoryview and array.array. They can only hold the same type.

con.index(e), con.count(e) are two useful method for list and tuple class. In C++, we use std::find and std::count to implement this function.

1.1.1 list and tuple

In Python code, line breaks are ignored inside pairs of [], {}, or (). So you can build multiline lists, listcomps, tuples, dictionaries, etc.,

Listcomps do everything the map and filter functions do, without the contortions of the functionally challenged Python lambda.

```
1 beyond_ascii = [ord(s) for s in symbols if ord(s) > 127]
2 [162, 163, 165, 8364, 164]
3
4 #use filter and map function below.
5 beyond_ascii = list(filter(lambda c: c > 127, map(ord, symbols)))
```

Listcomps can build lists from the Cartesian product of two or more iterables.

```
1 colors = ['black', 'white']
2 sizes = ['S', 'M', 'L']
3 tshirts = [(color, size) for color in colors for size in sizes]
```

Genexps use the same syntax as listcomps, but are enclosed in parentheses rather than brackets. If the generator expression is the single argument in a function call, there is no need to duplicate the enclosing parentheses.

```
1 >>> tuple(ord(symbol) for symbol in symbols)
2 (36, 162, 163, 165, 8364, 164)
3 >>> import array
4 >>> array.array('I', (ord(symbol) for symbol in symbols))
5 array('I', [36, 162, 163, 165, 8364, 164])
```

The *rest syntax in assignment unpacking collects remaining values into a list. This is different from function parameters, where *args becomes a tuple.

```
a, b, *rest = range(5) #rest is list

def fun(a, *rest):
  print(type(rest))
  print(rest)
fun(1, 2, 3, 4) # rest is tuple.
```

nested unpacking

```
l = ['a', (2, 3)]
a, (b, c) = l # a, b, c = l will report error here.
print(b)
```

s.sort() will sort items in place in list, it with key and reverse. built-in sorted() will return new list

```
list1.sort(key=len, reverse = true)
newlist = sorted(list1)
```

First of all, unlike other languages, both lists and tuples in Python support negative indexing. -1 represents the last element, -2 represents the second-to-last element, and so on.

```
l = [1, 2, 3, 4]
l[-1]  #4
```

Slice is left close, right open, just like range in C++, that is very important to remember.

```
for x in range(len(l)-1, -1, -1) #iterate from end to beginning.
```

list and tuple can be transformed each other.

```
list((1, 2, 3))
[1, 2, 3]

tuple([1, 2, 3])
(1, 2, 3)
```

tuple is more efficient

```
l = [1, 2, 3]
l.__sizeof__() #64

tup = (1, 2, 3)
tup.__sizeof__() #48
```

Python	C++	
pop/append	push_back/pop_back	
insert/del	insert/erase	
remove	remove	
sort	std::sort	
sorted	std::sort_copy	
len	size	
reverse	reverse	

- If you're unsure whether to use the del statement or the pop() method, here's a simple rule of thumb: if you want to delete an element from a list and won't use it in any way afterward, use the del statement; but if you want to continue using the element after deleting it, use the pop() method.

There's a strong culture of tuples being for heterogeneous collections, similar to what you'd use structs for in C, and lists being for homogeneous collections, similar to what you'd use arrays for. But I've never quite squared this with the mutability issue mentioned in the other answers. Mutability has teeth to it (you actually can't change a tuple), while homogeneity is not enforced, and so seems to be a much less interesting distinction.

1.1.2 set and dict

In Python 3.7 and above, dictionaries are guaranteed to be ordered (Note: In Python 3.6, the ordered behavior of dictionaries is an implementation detail, and it wasn't officially a language feature until 3.7. Therefore, in 3.6, the order cannot be guaranteed 100%). Before Python 3.6, dictionaries were unordered. Their size is dynamic, and elements can be freely added, removed, or modified.

Use below ways to create a dictionary and set.

```
d1 = {'name': 'jason', 'age': 20, 'gender': 'male'}
d2 = dict( {'name': 'jason', 'age': 20, 'gender': 'male'} )
d3 = dict([('name', 'jason'), ('age', 20), ('gender', 'male')])
d4 = dict(name='jason', age=20, gender='male')
d1 == d2 == d3 ==d4
True

s1 = {1, 2, 3}
s2 = set([1, 2, 2, 3]) # use set constructor
s1 == s2
True
```

set also use `{}`, but it just include one element, not like dict, include colon and two items.

s.remove, s.discard(e) has different way to deal with non existing element.

Atomic immutable types are all hashable, a tuple is hashable only if all its items are hashable. A frozen set is hashable.

```
d = {'b': 1, 'a': 2, 'c': 10}
d_sorted_by_key = sorted(d.items(), key=lambda x: x[0]) #
d_sorted_by_value = sorted(d.items(), key=lambda x: x[1]) #
d_sorted_by_key
[('a', 2), ('b', 1), ('c', 10)]
d_sorted_by_value
[('b', 1), ('a', 2), ('c', 10)]
```

1.1.3 String and number

In Python, there is no distinct char type like in C or C++. Instead, Python treats a single-character as a string too.

If you're working with characters (e.g., converting to/from Unicode), you can use: `ord('A') → 65` (get Unicode code point) and `chr(65) → 'A'` (get character from code point)

In Python, strings can be defined using single quotes, double quotes, or triple quotes, and there is no difference in meaning between them. Triple-quoted strings are typically used for multi-line string scenarios.

Strings in Python are immutable (except for the '+' concatenation operation in newer Python versions, which is an exception). Therefore, arbitrarily changing the value of characters within a string is not allowed.

```
s = 'hello'
s[0] = 'H'
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
TypeError: 'str' object does not support item assignment

s = 'H' + s[1:]
s = s.replace('h', 'H')

str1 += str2  # str1 = str1 + str2
```

```
len(str(3.14159).replace('.', ''))
```

Since Python 2.5+, string concatenation has become much more efficient, so you can use it with confidence.

String formatting using string.format is often used in situations like output and logging. If you don't have any fined requirement about format, you can use f"output{var}".

```
'Hello, {0}, improved {1:.1f}%'.format('Yan', 17.125)
```

1.1.4 binary

In c++, read binary use char array, but in python, we use bytes or bytearray.

Feature	bytes	bytearray
Mutability	Immutable	Mutable
Type	`<class 'bytes'>`	`<class 'bytearray'>`
Use Case	Suitable for constant binary data	Suitable for binary data that needs to be modified
Syntax	`b'hello'` or `bytes([104, 101, 108, 108, 111])`	`bytearray(b'hello')` or `bytearray([104, 101, 108, 108, 111])`
Memory Efficiency	Slightly more memory efficient	Slightly more memory for mutability overhead
Support for Methods	Most string-like methods (excluding mutating ones)	Similar methods plus mutating ones like `append()`, `pop()`, etc.

Table 1.2: Comparison of `bytes` and `bytearray` in Python

1.2 operator

Difference between = and is.

```
x = [1, 2, 3]
y = x
z = [1, 2, 3]

print(x is y)  # True - same object
print(x is z)  # False - same content, but different objects
print(x == z)  # True - values are equal
```

Dividing any two numbers always results in a floating-point number, even if both numbers are integers and the division is exact.

Two for means multiply. if you want to combine, use zip function.

```
a = ['a', 'b', 'c']
b = [1,2,c]

c = {x:y for x, y in zip(a,b)}
c {'a': 1, 'c': 3, 'b': 2}
```

```
6
7 c = {x:y for x in a for y in b} #c is dict, so only 'a': 3 is
      valid, 'a':1
8 # is overwritten.
9 C {'a': 3, 'c': 3, 'b': 3}
```

Slice is built-in type. it only can be used in `[]`, `__getitem__()`. [start:stop] includes start index, but excludes stop index. It does not mean "start at 1 and take 3 elements" — that's a common misunderstanding.

1.3 loop and if

Use is None to judge if it's None

string	empty string is False
int	0 is False
Bool	
list/tuple/list/dict	empty
object	None is False,

How to use index and value at the same time.

```
1 l = [1, 2, 3, 4, 5, 6, 7]
2 for index in range(0, len(l)):
3   if index < 5:
4     print(l[index])
5
6 l = [1, 2, 3, 4, 5, 6, 7]
7 for index, item in enumerate(l):
8   if index < 5:
9     print(item)
```

It's important to know that the range() function is implemented directly in C, so calling it is very fast. On the other hand, in a while loop, the operation i += 1 is indirectly executed through Python's interpreter, which calls the underlying C implementation. Moreover, this seemingly simple operation involves object creation and deletion—because i is an integer and integers are immutable, i += 1 is essentially equivalent to i = new int(i + 1).Therefore, it's clear that a for loop is more efficient in comparison.

```
1 i = 0
2 while i < 1000000:
3   ...
4   i += 1
5
6 for i in range(0, 1000000):
7   ...
```

for and if at the same line, if you have else, put it in front, if no else, put after for.

```
for item in iterable:
if condition:
  expression1
else:
  expression2

expression for item in iterable if condition
expression1 if condition else expression2 for item in iterable

y = [value * 2 + 5 if value > 0 else -value * 2 + 5 for value in
    x]

text = ' Today,  , is , Sunday'
text_list = [s.strip() for s in text.split(',') if len(s.strip())
    > 3]
print(text_list)
['Today', 'Sunday']
```

ZeroDivisionError NameError and TypeError, three most used exceptions.

In this code, the try block attempts to read the file file.txt and perform a series of operations on its data. In the end—whether the file is read successfully or an error occurs—the program will execute the statements in the finally block, which closes the file stream to ensure the file's integrity. Therefore, the finally block is typically used to place statements that must be executed no matter what.

```
import sys
try:
  f = open('file.txt', 'r')
  .... # some data processing
except OSError as err:
  print('OS error: {}'.format(err))
except:
  print('Unexpected error:', sys.exc_info()[0])
finally:
  f.close()
```

1.4 magic method

For example, the statement for i in x actually causes the invocation of iter(x), which in turn may call x.__iter__() if that is available, or use __getitem__().

Normally, your code should not have many direct calls to special methods. Unless you are doing a lot of metaprogramming

The __repr__ special method is called by the repr built-in to get the string representation of the object for inspection. Without a custom __repr__, Python's console would display a Vector instance <Vector object at 0x10e100070>.

Basically, bool(x) calls x.__bool__() and uses the result. If __bool__ is not implemented, Python tries to invoke x.__len__(), and if that returns zero, bool returns False. Otherwise bool returns True.

__str__ is called by str() and implicitly used by the print function. __repr__is called by repr() built-in.

no custom __str__, then will call __repr__ as fallback.

1.5 Advance feature

1.5.1 Generator expression

Here's the key point to remember: a generator is the lazy version of an iterator. Any time you return back a list, you can use yield to replace

```
def index_normal(L, target):
  result = []
  for i, num in enumerate(L):
    if num == target:
    result.append(i)
  return result

print(index_normal([1, 6, 2, 4, 5, 2, 8, 6, 3, 2], 2))

######### output ######### [2, 5, 9]
```

```
def index_generator(L, target):
  for i, num in enumerate(L):
    if num == target:
      yield i

print(list(index_generator([1, 6, 2, 4, 5, 2, 8, 6, 3, 2], 2)))

######### output ######### [2, 5, 9]
```

Generator expression allows creating a generator on a fly without a yield keyword. However, it doesn't share the whole power of generator created with a yield function. The syntax and concept is similar to list comprehensions:

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

```
3
4 print(list_comp)
5 [0, 4, 16, 36, 64]
6
7 print(gen_exp)
8 <generator object <genexpr> at 0x7f600131c410>
```

listcomp is meant to do one thing: to build a new list generator expression can also be used to build tuple and array.

```
1 def is_subsequence(a, b):
2   b = iter(b)
3   return all(i in b for i in a)
4
5 print(is_subsequence([1, 3, 5], [1, 2, 3, 4, 5]))
6 print(is_subsequence([1, 4, 3], [1, 2, 3, 4, 5]))
7
8 ########## output ##########
9 True
10 False
```

1.5.2 name

Roughly speaking, namespaces are just containers for mapping names to objects.

From module import * will not import any variable _x. __X__ is system varaible, has specific meaning to interpreter.

The LEGB rule in Python defines the scope resolution order — how Python looks up variable names. It stands for:

1. Local can be inside a function or class method.
2. Enclosed can be its enclosing function, e.g., if a function is wrapped inside another function.

```
1 def outer():
2   x = 20  # Enclosing scope
3   def inner():
4     print(x)
```

3. Global refers to the uppermost level of the executing script itself, and
4. Built-in are special names that Python reserves for itself.

```
1 len = 5  # Overrides built-in temporarily
2 print(len("abc"))  # Error
```

```
1 a_var = 'global value'
2
```

```
def outer():
  a_var = 'local value'
  print('outer before:', a_var)
  def inner():
    nonlocal a_var
    a_var = 'inner value'
    print('in inner():', a_var)
  inner()
  print("outer after:", a_var)

>>>outer()
outer before: local value
in inner(): inner value
outer after: inner value
```

The CPython implementation uses a special optimization for local variables: They aren't dynamically looked up at runtime from a dictionary, as globals are, but rather are assigned indices statically at compile time, and are looked up by index at runtime, which is a lot faster. This requires the Python compiler to be able to identify all local names at compile time, which is impossible if you have a wildcard import at function level.

```
g = 1
def fun():
  print g # this will cause error.
  g = 2
  print g

fun()
print g

# output
2  # create a local name g
1
```

In practice, it is usually a bad idea to modify global variables inside the function scope, since it often be the cause of confusion and weird errors that are hard to debug. If you want to modify a global variable via a function, it is recommended to pass it as an argument and reassign the return-value.

```
a_var = 2

def a_func(some_var):
return 2**3

a_var = a_func(a_var)
print(a_var)
```

```
MIN_VALUE = 1
MAX_VALUE = 10
def validation_check(value):
  global MIN_VALUE
  ...
  MIN_VALUE += 1
  ...

validation_check(5)
```

```
def outer():
  x = "local"
  def inner():
    nonlocal x # nonlocal means x is outer defined x
    x = 'nonlocal'
    print("inner:", x)
  inner()
  print("outer:", x)

outer()
#output
inner: nonlocal
outer: nonlocal
```

For class and function, L and G are different, but for module, L and G are same.

For local, The locals are exposed there by the built-in locals() and vars() functions. **The locals should be considered a read-only namespace, as there is no language guarantee that changes you make to it directly will actually be applied.**

You can control global namespace inside a function.

```
X = 2
def fun():
  global X
  X = 44
  Y = 55
  globals()['Y'] = 3

>>>print(X,Y) # output 44 3
```

Warning: For-loop variables "leaking" into the global namespace. That is why global variable should name by global_*** to avoid conflict with local name. They all follow snake_case style.

```
b = 1
for b in range(5):
```

```
3     if b == 4:
4       print(b, '-> b in for-loop')
5
6 print(b, '-> b in global')
7
8 >>>4 -> b in for-loop
9 4 -> b in global
```

Regardless of the code block, the globals are exposed there by the built-in globals() function.

```
1 global_var
2 def f2():
3     fc = 3
4     fd = 4
5     print(globals())
6     print(locals())
```

For some code blocks, like function and class bodies, the globals and locals are distinct; and the locals are inaccessible from outside that code block. For other code, like modules, they are the same thing (and thus equally accessible). With exec(), you can pass both in, thus controlling the distinction.

If you want to access them at the same time, you can:

```
1 X = 99
2 def selector():
3       import __main__    # import enclosing module
4       print __main__.X   # qualify to get to global version of name
5       X = 88             # unqualified X classified as local
6       print X            # prints local version of name
7
8 >>> selector()
9 99
10 88
```

1.5.3 import and module

builtin

- builtins is conception for module.
- relationship between builtins and __builtins__, A good explanation is "What's the deal with __builtins__ vs __builtin__" link address is

  `http://mathamy.com/whats-the-deal-with-builtins-vs-builtin.html`

- Why use builtin, we can add a name to builtin, so in all the other modules, you can use it.

dir an dict

- dir includes __dict__
- not all instance has __dict__. Some internal type use slot instead.
- A good introduction is "whats-the-biggest-difference-between-dir-and-dict-in-python"
- search path. modify sys.path dynamiclly.
- import/from copies names but doesn't link.
- think it as a name space
- it just import once, then fetch imported module object
- package is different with module, you just put module into the different path to avoid name conflict. but you need to put __init__.py file in each path.
- If you put code before the function definitions, it will execute before the __name__ check.

```
print("This code executes before main.")

def functionA():
  print("Function A")

def functionB():
  print("Function B")

if __name__ == '__main__':
functionA()
functionB()
```

If this module is indeed being used as the main script, this code results in:
This code executes before main. Function A Function B If this module was imported instead of used as the main script, you get the following at import time:
This code executes before main.

```
gl = [1,2,3]
def fun():
  print gl
  gl[1] = 33
  print gl

fun()
```

```
print gl gl    ag
# output
[1,2,3]
[1,33,3]
[1,33,3]
```

blow gl = [] will cause gl point to a local variable. so the first print gl will cause error.in order to correct this error, you need to use global declare

```
gl = [1,2,3]
def fun():
  o = [4,5,6]
  def infun():
    o = [] # create a local name o
    o.append(1)
    print o  yan zhao
  print o

fun()
print gl
# output
[1]
[4,5,6]
[1,2,3]
```

```
gl = [1,2,3]
def fun():
  o = [4,5,6]
  def infun():
    o = [] # create a local name o
    o.append(1)
    return o
  o = infun()
  print o

fun()
print gl
# output
[1]
[1,2,3]
```

```
gl = [1,2,3]
if True:
  ol = [4,5,6]
  if True:
    ol = []
    gl = []
  print ol
  print gl
```

```
# output
[]
[]
```

- reference always work, it will look for name from inside to outside.
- for immutable object, such as int. you only can use assignment. for mutable object, such as list, if you use assignment. result will be seen below item
- assignment will create a new name, if you want to modify outside, you need to global or make function return.
- for mutalbe object, if you use list[0] = 'change'. then it will modify outside object.
- gint = 3 will always create local name. list =[] will create local name, list[0] = 'change' will modify outside object.
- for if while block, will not create local name, always modify outside.

1.6 module

By default, the Python interpreter searches the current directory, all built-in modules, and installed third-party modules. The search paths are stored in the path variable of the sys module:

```
import sys
>>> sys.path
['', '/Library/Frameworks/Python.framework/Versions/3.6/lib/
    python36.zip', ...
```

We can import modules using absolute or relative paths.

In large-scale projects, modularization is very important, and modules should be referenced using absolute paths, which start from the root directory of the program.

Also, remember to make good use of if

```
__name__ == '__main__'
```

to prevent code from being executed during an import.

1.7 reference

Put mutable items in tuples is not a good idea.

Different with C++, The key different is if the value is immutable.

```
a = 1
b = a
a = a + 1

# b is 1, but a is 2.

a—>1
————————————————
a—>1
b—/
————————————————
a—>2
     1
b—/
————————————————————
```

Remember that arguments are passed by assignment in Python. Since assignment just creates references to objects, there's no alias between an argument name in the caller and callee, and so no call-by-reference per Se

```
def my_func1(b):
  b = 2  # 2 is value, not a variable.

a = 1
my_func1(a)
a  # a is still 1
```

```
def my_func2(b):
  b = 2
  return b

a = 1
a = my_func2(a)
a #a is 2 now
```

```
def my_func4(l2):
l2 = l2 + [4]

l1 = [1, 2, 3]
my_func4(l1)
l1 # l1 is still [1, 2, 3]
```

```
def my_func5(l2):
l2 = l2 + [4]
return l2

l1 = [1, 2, 3]
```

```
l1 = my_func5(l1)
l1 #l1 is [1, 2, 3, 4]
```

Today, we learned about Python variables and the basic principles of assignment, and we also explained how parameter passing works in Python. Unlike other languages, Python's parameter passing is neither pass-by-value nor pass-by-reference; instead, it is assignment passing, or more precisely, **object reference passing.**

It's important to note that this assignment or reference passing does not refer to a specific memory address, but rather to a specific object. If the object is mutable, then when it changes, all variables referencing that object will reflect the change. If the object is immutable, a simple assignment will only change the value of one variable, while the others remain unaffected.

With this understanding, if you want to change the value of a variable through a function, there are generally two ways to do it. One is to pass a mutable data type (like a list, dictionary, or set) as an argument and modify it directly. The second is to create a new variable to store the modified value and then return it to update the original variable. In practice, we tend to prefer the second approach because it is clearer and less error-prone.

Assignment creates references, Not copies

```
l = [1,2,3]
m = ['a',L, 'b'] #shallow copy
m1 =['a',L[:], 'b'] #deep copy
```

Immutable Types can't be changed.

```
t=(1,2,3)
t[2] = 4 # error
t = t[:2] + (4,)  # OK
```

Same value vs. Same object

```
L1 = [1,2,3]
L2 = [1,2,3]
L1==L2  #True
L1 is L2 #False
```

there are three kind of "copy"

```
L1 = [1,2,3]
L2 = L1
L2 = L1[:]
```

You can use the builtin list.copy() method (available since python 3.3):

```
new_list = old_list.copy()
```

You can slice it:

```
new_list = old_list[:]
```

You can use the built in list() function:

```
new_list = list(old_list)
```

You can use generic copy.copy(): This is a little slower than list() because it has to find out the datatype of old_list first.

```
import copy
new_list = copy.copy(old_list)
```

If the list contains objects and you want to copy them as well, use generic copy.deepcopy():

```
import copy
new_list = copy.deepcopy(old_list)
```

Obviously the slowest and most memory-needing method, but sometimes unavoidable.

1.8 Application

- for file, can read, readline and readlines, for output, only write.
- json

```
import json

params = {
   'symbol': '123456',
   'type': 'limit',
   'price': 123.4,
   'amount': 23
}

with open('params.json', 'w') as fout:
params_str = json.dump(params, fout)

with open('params.json', 'r') as fin:
original_params = json.load(fin)

print('after json deserialization')
print('type of original_params = {}, original_params = {}'.
    format(type(original_params), original_params))

########## output ##########

```

```
21 after json deserialization
22 type of original_params =<class 'dict'>, original_params =
       {'symbol': '123456', 'type': 'limit', 'price': 123.4, '
       amount': 23}
```

- For python, you can use below code to see if it support wide-unicode.

```
1 import sys
2 print(sys.maxunicode)
3 1114111-->support wide-unicde.
4 65535 --> support usct
```

- –enable-unicode=usc2 or usc4 is gone in Python 3.x. You now only have the choice to use the default (which is UCS2) or switch on the optional support for UCS4 by using –with-wide-unicode.
- blender –background –python script.py or blender –background –python-console
- re of python, pay attention to r'...'.

```
1 import re
2 re.match(r'^\d{3}\-\d{3,8}$', '010-12345'),
3
4 re.split(r'\s+', 'a b   c')
5 ['a', 'b', 'c']
```

- re of c++, use a function to do this job. regex_match

```
1 string s1 = "ab123cdef"; //
2 string s2 = "123456789"; //
3 regex ex("\\d+"); //
4
5 cout << s1 << " is all digit: " << regex_match(s1, ex) <<
      endl;
6 cout << s2 << " is all digit: " << regex_match(s2, ex) <<
      endl;
```

Chapter 2

Array of sequences

Container sequences: Can hold items of different types, including nested containers. Some examples: list, tuple, and collections.deque. Flat sequences: Hold items of one simple type. Some examples: str, bytes, and array.array.

```
1 array('d', [1.2, 3.3, 4.5]) #just hold bytes, integers and floats
```

Chapter 3

function

3.1 basic

- Function in python are first-class objects.
  1. created at runtime
  2. assigned to a varaible or element in a data structure
  3. passed as an argument to a function
  4. returned as the result of a function.
-

```
def my_func(message):
  print('Got a message: {}'.format(message))

my_func('Hello World')  #Got a message: Hello World
```

As mentioned earlier, one of the key differences between Python and other languages is that Python is dynamically typed, meaning it can accept any data type (integers, floats, strings, etc.). This also applies to function parameters.

For example, take the my_sum function we just discussed—we can also pass in lists as arguments to concatenate them:

```
def my_sum(a, b):
  return a + b

print(my_sum('hello ', 'world'))
```

Generally speaking, when we want to perform operations on elements in a collection, and the operations are relatively simple—such as addition or accumulation—we prefer to use functions like map(), filter(), reduce(), or list comprehensions.

- As for choosing between the two approaches: when dealing with large amounts of data, such as in machine learning applications, we tend to favor functional programming expressions because they are more efficient.
- When the data volume is small and you want your code to be more Pythonic, list comprehensions are also a good choice.
- However, if you need to perform more complex operations on the elements, for the sake of readability, we usually opt for a for loop, as it makes the logic clearer and easier to understand.

decorate, usually has two returns.

```
def log(func):
  def wrapper(*args, **kw):
    print('call %s():' % func.__name__)
    return func(*args, **kw)
  return wrapper

@log
def now():
  print('2024-6-1')

now = log(now) #above sytax sugar is just this statement
```

A better version of log

```
import functools

def log(func):
  @functools.wraps(func)
  def wrapper(*args, **kw):
    print('call %s():' % func.__name__)
    return func(*args, **kw)
  return wrapper
```

Anonymous functions have a limitation: they can only contain a single expression. There's no need to write return—the result of the expression is automatically returned.

An example of partial function

```
import functools
int2 = functools.partial(int, base=2)
int2('1000000')
```

Chapter 4

class

4.1 basic

But what if we want to restrict the attributes of an instance? For example, we only want to allow name and age attributes to be added to a Student instance.

To achieve this restriction, Python allows us to define a special variable called __slots__ when defining a class. This limits the attributes that instances of the class can have:

```
class Student(object):
  __slots__ = ('name', 'age') # use tuple here
```

4.2 magic method

4.3 class attributes

- A good example is "Python Class Attributes: An Overly Thorough Guide" https://www.toptal.com/python/python-class-attributes-an-overly-thorough-

```
class MyClass:
class_var = "I am a class variable"

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

@classmethod
def class_method(cls):
print(f"Class variable: {cls.class_var}")
# Can modify class variable:
```

```
    cls.class_var = "Modified class variable"
    return cls() # Can be used as a factory method

    @staticmethod
    def static_method(arg):
    print(f"Static method called with: {arg}")
    # Cannot access or modify class/instance variables directly

    # Example Usage
    obj = MyClass("I am an instance variable")

    # Calling class method
    new_obj = MyClass.class_method()
    print(MyClass.class_var)

    # Calling static method
    MyClass.static_method("Hello")
    obj.static_method("World")
```

```
class MyNumber:
  def __init__(self, value):
    self.value = value

  def __add__(self, other):
    if isinstance(other, MyNumber):
      return MyNumber(self.value + other.value)
    return NotImplemented

  def __radd__(self, other):
    return MyNumber(self.value + other)

  def __repr__(self):
    return f"MyNumber({self.value})"


a = MyNumber(10)
print(a + MyNumber(5))    # MyNumber(15) calls __add__
print(5 + a)              # MyNumber(15)  calls __radd__
print(a + 5)              # TypeError  __add__ returned
    NotImplemented and int doesn't know MyNumber
```

Chapter 5

meta

- There are two importants meta methods, customize class creation, and customize attribute access and set.
- Heck, attributes are fundamental to everything in Python. The sooner you understand what attributes are, and how they work, the sooner you'll have a deeper understanding of Python
- In other words, the dot notation that we use in Python all of the time is nothing more than syntactic sugar for "getattr". Each has its uses; dot notation is far easier to read, and "getattr" gives us the flexibility to retrieve an attribute value with a dynamically built string.

```
1 getattr(t, '__class__')
2 t.__class__
```

5.1 descriptor

5.2 reference

- The magic behind Attribute Access in Python, `https://codesachin.wordpress.com/2016/06/` introduces Attribute acess magic
- Another good article is Understanding Python metaclasses `https://blog.ionelmc.ro/2015/02`
- Python Types and Objects `http://www.cs.utexas.edu/~cannata/cs345/Class%20Notes/15%` It tell two sides of type and object, object is both a class and type. when it's a class, it's parent class of type. When it's instance, it's instance of type.

Chapter 6

test

```
def add(a, b):
  """
␣␣Returns␣the␣sum␣of␣a␣and␣b.
␣␣>>>␣add(2,␣3)
␣␣5
␣␣>>>␣add(-1,␣1)
␣␣0
␣␣"""
  return a + b

python -m doctest your_script.py # run test case in comment.
```

Chapter 7

GUI

Chapter 8

lib

- collections has namedtuple function, can return named tuple class

Chapter 9

python and c++

1) value, reference and reference count 2) copy and deepcopy 3) RAII 4) collection, range, iterate and iterable 5) lazy range, itertools(python) and views(C++) 6) functional: lambda, std::function in C++, decorator and partial in python 7) I/O and memory I/O 8) serialization and deserialization 9) thread and coroutine

```python
import itertools

nums = range(10)
evens = filter(lambda x: x % 2 == 0, nums)
squares = map(lambda x: x * x, evens)

print(list(squares))  # [0, 4, 16, 36, 64]


//////////////////C++
#include <iostream>
#include <ranges>
#include <vector>

int main() {
  auto nums = std::views::iota(0, 10);
  auto evens = nums | std::views::filter([](int x) { return x % 2 == 0; });
  auto squares = evens | std::views::transform([](int x) { return x * x; });

  for (int x : squares) {
    std::cout << x << " ";  // 0 4 16 36 64
  }
}
```