23rd March 2020

Tuples are data structures which are ‘read only lists’. Tuples are immutable variant of lists.

--

Import this -> in Python shell will display the Zen of Python by Tim Peters.

--

**Pylint** is a source-code, bug and quality checker for the Python programming language. It is named following a common convention in Python of a "py" prefix, and a nod to the C programming lint program. It follows the style recommended by PEP 8, the Python style guide.[4] It is similar to Pychecker and Pyflakes, but includes the following features:

Checking the length of each line

Checking that variable names are well-formed according to the project's coding standard

Checking that declared interfaces are truly implemented.[5]

It is also equipped with the Pyreverse module that allows UML diagrams to be generated from Python code.

It can be used as a stand-alone program, but also integrates with IDEs such as Eclipse with PyDev[6] and Visual Studio,[7] and editors such as Atom[8], GNU Emacs and Vim.

--

Numbers: Integral; non-integral

Integral : integers, Booleans

Non-integral: real numbers(floats, decimal(gives precision), complex(real and imaginary part), fractions are rational number

--

Collections are: sequences(lists(mutable), tuples, strings(both immutable), sets(mutable, frozen-sets(immutable), maps(dictionary)

Sets and dictionaries are hash maps.

Callables are anything that can be called: functions, generators, classes, class instances (by implementing \_\_class\_\_), instance methods, built-in functions, built-in methods.

Singletons: None, Notimplemented, Ellipses

--

Physical code > logical code > tokenize / NEWLINE token.

Implicit end of line, explicit end of line. Explicit multiline statement has to be ended using backslash ‘\’.

Comments and docstrings are different. Comments are left out when the code is compiled.

Private objects declared with single underscore in the beginning cannot be imported from a module using this syntax. \_var x = 10 // from module import \* // cannot access \_var

\_\_var is used to mangle class attributes. Useful in inheritance chains.

\_\_var\_\_ system defined names that have special meaning to the interpreter.

Naming conventions: packages: shortones, no underscores, lowercase.

Modules: short, lowercase, can have underscores.

Classes: CapWords or upper camel case

Functions: lowercase, separate words with underscore – snake\_case

Variables: same as functions

Constants: all upper case, words separated by underscore.

Ternary operation: a = 5

b = 'a < 10' if a < 10 else 'a > 10'

print(b)

function name is the variable assign to the function code.

When a variable is assigned to a lambda function, that variable is the name of the lambda function.

Do .. while is not there in Python. Instead use ‘while(true)’

a = 1

while True: # like 'do' while

a += 1

print(a)

if a > 10:

break

‘continue’ statement skips the remaining code and continue with the next iteration.

While .. else – ‘else’ will execute only when the ‘while’ is iterated over without break.

l = [1,2,3]

idx = 0

while idx < len(l):

if l[idx] == 10:

break

idx += 1

else:

l.append(10)

print(l)

**Iterable** is an object capable of returning values one at a time.

Lists, tuples, strings are iterables.

# for loop. Here 'i' is requesting for the next value from the iterable object.

for i in [10,15,12,5,20]:

print(i)

for also has an ‘else’ clause, which runs once the loop completes without a break.

Sets and dictionaries are iterables, but they are not ordered and so not indexed.

When referencing a class instance object, ‘str’ will look for the objects class and its memory address by default.

\_\_eq\_\_ to compare objects of a class.

Property setter and getter

class Rectangle:

def \_\_init\_\_(self, width, height):

self.\_width = width

self.\_height = height

def area(self):

return self.width \* self.height

# property getter - method name is same as property name

@property

def width(self):

return self.\_width

@property

def height(self):

return self.\_height

#property setter - method name is same as the property name

@width.setter

def width(self, width):

if width <= 0:

raise ValueError("Width should be greater than zero")

self.\_width = width

@height.setter

def height(self, height):

if height <= 0:

raise ValueError("Height should be greater than zero")

self.\_height = height

# checks if two objects are equal

def \_\_eq\_\_(self, other):

if isinstance(other, Rectangle):

if (self.\_width, self.\_height) == (other.\_width, other.\_height):

return True

return False

return "Can't compare instances of different classes"

# when checking str(object)

def \_\_str\_\_(self):

return "Rectangle object"

# object building details

def \_\_repr\_\_(self):

return "{0}, {1}".format(self.\_width, self.\_height)

obj = Rectangle(5,15)

obj.width = 25

obj.height = 50

print(obj.area())

print(obj.height)

output: 1250

50

**Python is a dynamically typed language. Everything in python are objects**

Id function is used to refer the memory address of the variable. Hex(id(var))

Sys.getrefcount(var) // ctypes.c\_long.from\_address(address).value

When the variable reference to the object having circular reference goes away, the garbage collector will remove the objects with circular references. Memory leak

The gc module. By default, gc is turned on. Object destructor is implemented by \_\_del\_\_

Gc.get\_objects() // gc.disable() // gc.collect

Stack is a linear data structure whereas Heap is a hierarchical data structure. Stack memory will never become fragmented whereas Heap memory can become fragmented as blocks of memory are first allocated and then freed. Stack accesses local variables only while Heap allows you to access variables globally.

An object, whose internal state can be changed, called mutable.

Immutable objects: numbers, strings, tuples, frozen-sets, user defined classes.

Mutable objects: lists, sets, dictionaries, user defined class

**A tuple containing a list elements can be modified.** T = ([1,2],[4,5,6]) t[0].append(3)

lst = [[1,2,3], [5,6,7]]

t = (lst, [8,9])

print(t)

t[1].append(10)

print(t)

t[0][0].append(4)

print(t)

output:

([[1, 2, 3], [5, 6, 7]], [8, 9])

([[1, 2, 3], [5, 6, 7]], [8, 9, 10])

([[1, 2, 3, 4], [5, 6, 7]], [8, 9, 10])

Shared reference is, more than one variable pointing to the same object in the memory.

With mutable objects, python mem manager will not create shared memory references.

Everything in python is passed by reference.

Variable equality in two ways. Memory address and internal data.

To check address equality, use identity operator ‘**is’**. To check data equality, use ‘**==’**

a = 10

b = 10

c = None

d = None

print (a is b)

print(a == b)

print (c is d)

print(c == d)

output: True

True

True

True

**Interning:** Reusing objects on demand. At startup, CPython preloads(caches) a global list of integers into memory(-5 to 256)

**Singleton objects** are classes that are instantiated only once.

When the python code is compiled, identifiers are interned. Function, class, variable names etc. it should share with an \_,letter and can include only an \_.

Sys.intern(string)

import sys

a = sys.intern("hello")

b = sys.intern("hello")

c = a

a is b

b is c

output: true true

peephole optimization happens at compile time.

**Peephole optimization** is a type of Code **Optimization** performed on a small part of the code. It is performed on the very small set of instructions in a segment of code. The small set of instructions or small part of code on which **peephole optimization** is performed is known as **peephole** or window.

Membership tests: mutables are replaced by immutables.

In Python, the int objects use a variable number of bits according to the size of the integer number.

Sys.getsizof(<number>) to know how many bits are used to store the value.

The floor operation ‘//’ is not the same as truncation when deals with negative ‘-‘ numbers.

Math.floor

a = 13

b =3

print(a == b \* (a // b) + a % b )

output: True

the default base of an integer is 10 // changing a number from base 10 to another type. Bin(10), oct(25), hex(34)

Rational numbers are fractions of integers. Any real number with a finite number of digits after the decimal point is also a rational number. Square root of 2 and pi are irrational numbers.

From fractions import Fraction

format(0.23, '.12f')

in bankers rounding, selects ‘even’ least significant digit

decimal.getcontext() // decimal.localcontext()

#tuple passed in Decimal constructor. the first argument in the tuple is the sign

from decimal import Decimal

t = (0,(3,4,5,6,7), -4)

a = Decimal(t)

print(a)

output: 3.4567

Not all functions available in math module is not in Decimal class.

Div and mod operations are not allowed in complex numbers. Math functions will not work with complex numbers; instead use cmath.

Complex class is built-in, which uses a rectangular coordinates as a constructor. A = complex(1,2)

a = complex(3,5)

b = 2 + 4j

c = a + b

d = a \* b

print(c)

print(d)

output: (5+9j) // (-14+22j)

--

bool class is definded PEP285. Bool class is a sub class of integers.

a = issubclass(bool, int)

print(a)

output: True.

True and False are singleton objects of type bool.

All objects in Python has an associated truth value.

Classes define their truth values by defining a special instance method. \_\_bool\_\_.If its not defined, use \_\_len\_\_.

class A():

pass

# def \_\_bool\_\_(self):

# return self != 0

def \_\_len\_\_(self):

return self != 0

a = A()

print(bool(a))

--

Output: True

Boolean operators: not, and, or

Use ‘return’ only when inside a function

Chained comparison: a>b>c>d<e!=f

If a positional parameter is defined with a default value, every positional parameter after that also has to be given with a default value.

‘Named’ arguments are given when the function is called.

Once use a named argument, all arguments follows should also be named.

‘packed values’ are bundled together in some way. Any iterable is considered a packed value.

Dictionaries and sets are unordered types. They can be iterated, but no guarantee of order.

# iterates through all the items in the list

a = ['a','b','c',[10,20,30],[1,2,3]]

for i in a:

for x in i:

print(x)

in an assignment operation, Python evaluates the ‘right handside’ first. So swapping a,b = b,a works.

Slicing works only with iterable sequences.

a = [1,2,3,4,5,6,7,8,9]

#b,c = a[0],a[1:]

#b,c = a[0:3], a[4:]

b, c, \*d = a

b, c, d

output: (1, 2, [3, 4, 5, 6, 7, 8, 9])

In [ ]:

Always unpack into a list

To unpack the key and value of a dictionary, use double star. \*\* on the lhs

\*args return a tuple of positional arguments.

def func(a, b):

return a and b

# return 0 as 'false' becase a is 0. Else it will return 20 from the rhs - when its returning from an 'and'

func(0,20)

#unpacking iterables in to \*args

l = [10,20,30]

def func(a,b,\*args):

print(a)

print(b)

print(args[0],args[2])

func(1,2,\*l)

#unpacking iterables in to \*args

l = [10,20,30,40,50]

def func(a,b,\*args):

print(a)

print(b)

#\*args starts at l[2]

print(args[0],args[2])

#func(1,2,\*l)

func(\*l)

output: 10

20

30 50

Positional argument following \*args are called keyword arguments.

Bare ‘\*’ indicates end of positional arguments.

Positional arguments cannot follow named arguments

\*\* scoops up the keyword arguments. Stores arguments in a dictionary.

Generator expression use parenthesis. ()

Higher order functions are that takes a function as argument or returns a function.

First string in the body of a function is the docstring. Its stored in function.\_\_doc\_\_ property

Annotations are stored in function.\_\_annotations\_\_ property

Type hints are advanced version of annotation

def func(a: 'f name' ="Jaison", b:'l name' = "Jacob") -> 'full name':

"concatenates a name"

return a + ' '+ b

func()

print(func.\_\_doc\_\_)

print(func.\_\_annotations\_\_)

the lambda expression returns a function object

No annotations in lambda.

func = lambda x: x\*\*2 if x > 10 else x

a = func(5)

print(a)

output: 5

--

Sorted uses stable sort

lst = ['N','n','K','J','c','r']

srt = sorted(lst, key=lambda x: x.upper())

print(srt)

output: ['c', 'J', 'K', 'N', 'n', 'r'

---- (review1) how to unpack a dictionary in to variable? Or right handside unpacking. Use {\*\*dic}. a = {\*\*dic}

a = dict(a=10,b=20,c=30)

c = {\*a}

b = {\*\*a}

print(c)

print(a)

Functions are first class objects, that have properties, attributes and methods.

**Dir()**is a builtin function, given an object, will return all the attributes of that object.

def func(a=10,\*, kw=1):

c = 10

m = func;

m.new\_var = "Hello world"

print(dir(func))

print(m.\_\_name\_\_)

print(m.\_\_defaults\_\_)

print(func.\_\_code\_\_.co\_varnames)

Objects and classes have attributes bound to the object. An attribute that is callable is called a method.

Import inspect.ismethod, isfunction, isroutine, getsource, getmodule, getcomments, signature,

Getcomments(func) – will return the comments given just above the function definition.

Divmod(x,y,/) – here the ‘/’ says that the preceding parameters are positional only.

Callable(print) – to check if an object is callable. All callables return a value.

A higher order function is one that takes in a function as a parameter and returns a function. Sorted, maps (list comprehension), filter(generator expression

Map(func,\*iterables) returns an iterator that calculates the function applied to each element of the iterables.

Fileter(func, iterable) – if fuction returns true, it will retain the iterable element, if it false, will throw out the element. Returns an iterator, for which all the elements are returned ‘truthy’.

Zip(\*iterables) – takes more than one iterable and returns a tuple zipping up all the iterables

#map

l1 = [10,21,30]

l2 = [5,15,25]

it = map(lambda x, y: x + y, l1,l2)

print(list(it))

#filter

flt = filter(lambda x: x % 2 == 0, l1)

print(list(flt))

#zip

zp = zip(l1,l2)

print(list(zp))

output: [15, 36, 55]

[10, 30]

[(10, 5), (21, 15), (30, 25)]

List comprehension [ xpression1 for variable in iterable if expression2]

l1 = [10,20,30]

l2 = [2,4,6]

#cobining filter with map

lst = filter(lambda x: x > 15, map(lambda n,m: n + m, l1,l2))

print(list(lst))

#list expression

a = list([x + y for x,y in zip(l1,l2) if x+y > 20])

#generator expression

gen = (x + y for x,y in zip(l1,l2) if x+y > 20)

print(a)

for i in gen:

print(i)

output: [24, 36]

[24, 36]

24

36

Reducing functions are functions that recombine an iterable recursively and end up return a single value. They are often called aggregators, accumulators or folding functions.

#find maximum of a number from a list

ls = 10,20,30,4,5,7,19,33,55,1

result = ls[0]

for r in ls:

if r > result:

result = r

print(result)

#reduce

from functools import reduce

result = reduce(lambda x,y : x + " " + y, ("Jaiosn", "Jacob", "Valayil"))

print(result)

output: Jaiosn Jacob Valayil

--

Min,max,any,sum,all are builtin reducing functions.

#partial

from functools import partial

def func\_add(a,b,c):

return a + b + c

#->

par = partial(func\_add, 10)

result = par(20,30)

print(result)

output:60

#partial function using function

def func1(a,b,\*args,d,\*\*kwargs):

print(a,b,args,d,kwargs)

def func2(\*args,s,m,\*\*kwargs):

return func1(10,\*args,d=s,\*\*kwargs)

func2(20,30,s=10,m=15,k=19)

output: 10 20 (30,) 10 {'k': 19}

operator module contains arithmetic operators, comparison, sequence/mapping – contains(s,val), concat, countOf(s,val), getitem, setitem,delitem. Itemgetter

#itemgetter sets an index for a sequence and returns a callable

from operator import itemgetter

lst = [1,23,3,4,5,6,7]

f = itemgetter(3,4,5)

print(f(lst))

--

#attrgetter sets an index for properties of an object and returns a callable

from operator import attrgetter

def func():

pass

#setting function attributes

func.a = 10

func.b= 20

res = attrgetter('a','b')(func) #callable is called with a ()

print(res)

#attrgetter calling the callable and another way using methodcaller

from operator import attrgetter, methodcaller

s = 'monte python'

# attrgetter('upper')(s)()

f = attrgetter('upper')

f(s)() # calling 'upper' callabe

x = methodcaller('upper')(s)

print(x)

methodcaller calls the named attribute and calls it aswell.

**Scopes: Global, local, non local, nested**

The portion of the code, where the name/binding is defined is called the lexical scope of the variable. These bindings are stored in namespaces.

Global scope is module scope, spans a single file.

Some built-in’s are available across the modules: True, False, None, dict, print

Built-in has a namespace. Each module loaded has its own namespace.

If a label couldn’t find in a namespace, python look above(upward) for it, not below it.

Masking is redfining a builtin or global variable inside the local scope.

Variables defined inside a function is local to that function. Every time the function is called, the scope is recreated.

There is no ‘code block’ variables in Python.

Inner function modifying the outer function variable has to redefine it as ‘nonlocal’ variable.

def out():

x = 10

print(f"outer {x}")

def out1():

#local x

x = 2

print(f"outer1 {x}")

def out2():

nonlocal x

x = 3

print(f"outer2 {x}")

out2()

out1()

out()

output: outer 10

outer1 2

outer2 3

nonlocal variables referenced are called ‘free’ variables.

Closure is a function plus the extended scope that contains the free variable.

def outer():

x = "Python"

def inner():

print(f"inner {x}")

return inner

fn = outer()

fn()

‘cell’ references the variable in outer scope

Fn.\_\_closure\_\_

Shared free variables can be accessed in the inner functions and can be modified.

def outer():

x = 10

def inner():

nonlocal x

x += 1

print(f"inner {x}")

def inner1():

nonlocal x

x += 1

print(f"inner1 {x}")

return inner, inner1

f1, f2 = outer()

f1()

f2()

lambda expressions become closures only if they have a free variable.

--

A decorator takes a function as argument

Returns a closure

Closure takes any combination of parameters

Sun some function in the inner code (closure)

Closure function calls the original function passing the arguments passed to the closure

Returns whatever is returned by that function call.

#decorator

from functools import wraps

def decor(fn):

print("decor")

@wraps(fn)

def inner(\*args,\*\*kwargs):

print("inner")

return fn(\*args,\*\*kwargs)

#inner = wraps(fn)(inner)

return inner

def add(a,b,\*,d):

"function add"

return a + b + d

add = decor(add)

res = add(10,20,d=5)

print(res)

help(add)

#stacked decorator:

def auth(fn):

def inner():

res = fn()

return res

return inner

def logged(fn):

def inner():

res = fn()

return res

return inner

@auth

@logged

Def add():

Pass

add = auth(logged(add))

maxsize of lru\_cache is 128

when arguments has to be passed to the parameterized decorator, define a decorator factory outer to the decorator and return the decorator.

def dec\_factory(num):

def dec(fn):

from functools import wraps

@wraps(fn)

def inner(\*args,\*\*kwargs):

print(f"number is {num}")

return fn(\*args,\*\*kwargs)

return inner

return dec

@dec\_factory(3)

def add(a,b,\*,c):

return (a + b) \* c

add = add(10,15,c=2)

print(add)

output: number is 3

50

Add = dec\_factory(3)(add)

--

Decorator class uses \_\_call\_\_ to define the inner function inside – same like decorator factory function

#Decorator class

class Decor\_Class:

def \_\_init\_\_(self,a,b):

self.a = a

self.b = b

def \_\_call\_\_(self,fn):

def inner(\*args, \*\*kwargs):

print(f"inner called, a {self.a} b {self.b}")

return fn(\*args, \*\*kwargs)

return inner

@Decor\_Class(10,20)

def hello(c):

print(c)

hello("Hello")

# obj = Decor\_Class(5,7)

# hello = obj(hello)

# hello("world")

Decorate a class (not decorator class)

from fractions import Fraction

def dec\_speak(cls):

cls.speak = lambda self, msg: f"class is {self.\_\_class\_\_.\_\_name\_\_}, msg is {msg}"

return cls

decim = dec\_speak(Fraction)

f = decim(100,2)

f.speak("All the very best")

#applying the decorator function on another class

class Person:

pass

p = dec\_speak(Person)

p1 =p()

p1.speak("Hi there")

@total\_ordering // functools singledispatch

**Tuples are read only lists.**

In named tuples, in addition to the value of a position, we give names as well.

Tuples lists strings

Containers containers containers

Order matters order matters order matters

Hetrogeneous/homogeneous hetro/homo(more homo) homo

Indexable indexable indexable

Iterable iterable iterable

**Immutable mutable immutable**

**Fixed length length can change fixed length**

**Fixed order order of lements can fixed order**

**Change – can do in-place**

**Sorts, reversals**

Record = (djia, 2019, 03, 24, 1000,9000,9500,300,100,150)

Symbol, year, month, day, high, low, avg, **\*\_**, d\_avg = record (pythonistic)

#A list of tuple

N\_Y = "US","NA",1000000

TOKYO = "JAPAN","ASIA",1200000

JOHANNESBURG = "SA","AFRICA",800000

SANJOSE = "COSTA RICA", "LA", 500000

cities = [N\_Y,TOKYO,JOHANNESBURG,SANJOSE]

population = sum(city[2] for city in cities)

#population

total = 0

for city in cities:

total+=city[2]

total

from collections import namedtuple

namedtuple is a function, which is a class factory, creates a class which inherits from tuple. Also provides named properties to access the elements of the tuple.

# named tuple

from collections import namedtuple

Point2D = namedtuple('Point2D', 'x y')

pt = Point2D(10,20)

print(pt.\_asdict)

print(pt.\_fields)

OrderedDict = Dictionaries that guarantee the ‘key order’.

List.extend(values) – used to append more than a value into the list.

from collections import namedtuple

Person = namedtuple('Person', 'fname lname age')

p = Person("Jaison", "Jacob", 45)

print(p)

#update a value

p = p.\_replace(fname="Esther", lname="Sarah")

print(p.age)

print(hex(id(p)))

#modify a named tuple

values = p[:2]

#a = Person(\*values, 46)

a = Person.\_make(values + (100,))

print(a)

#create a new class from existing class by adding another field (extend a named tuple)

PersonExt = namedtuple('PersonExt', Person.\_fields + ('state',))

print(PersonExt.\_fields)

\_\_defaults\_\_ of function allows to provide default values for a function.

Class\_\_new\_\_.\_\_defaults\_\_(defaults) is right aligned.

Class.\_replace(defaults) is prototype.

--

from collections import namedtuple

Person = namedtuple('Person', 'fname lname age')

p = Person("Jaison", "Jacob", 45)

#set a docstring

Person.\_\_doc\_\_ = "Person class"

Person.fname.\_\_doc\_\_ = "First Name"

print(Person.fname.\_\_doc\_\_)

#defaults prototype

proto1 = Person(fname="FirstName", lname = "LastName", age=10)

pr1 = proto1.\_replace(age=15)

print(pr1)

#setting defaults through constructor

Person.\_\_new\_\_.\_\_defaults\_\_ = ("Jacob",23)

p1 = Person("Emil")

print(p1)

--

from random import randint, random

from collections import namedtuple

Color = namedtuple('Color', 'red green blue orange')

def the\_color():

red = randint(0,255)

green = randint(0,255)

blue = randint(0,255)

orange = round(random(), 2)

return Color(red, green, blue, orange)

color = the\_color()

print(color)

dictionary can be used to create fields in namedtuples.

#Alternative to dictionaries

from collections import namedtuple

data\_list = [{'key1':100, 'key2':200, 'key3' : 300}, {'key3' : 10, 'key1' : 20, 'key2' : 15}, {'key2' : 11, 'key3':23}]

print(data\_list)

def tuplify\_dict(dict\_):

keys = {key for dic in data\_list for key in dic.keys()}

Struct = namedtuple('Struct', sorted(keys))

Struct.\_\_new\_\_.\_\_defaults\_\_ = (None, ) \* len(Struct.\_fields)

return [Struct(\*\*dick) for dick in dict\_]

d\_tpl = tuplify\_dict(data\_list)

print(d\_tpl)

--

Modules are objects of type ModuleType

Globals() – gives all the labels defined in the global / module namespace

Namespaces are dictionaries.

Locals() – provides all labels inside local namespaces (eg – function)

Builtin modules in python are written in ‘c’. Standard library most of it is written in ‘python’ and a few in ‘c’.

When a module is loaded, its reference is stored in a cache in sys.modules and in globals.

Module.\_\_dict\_\_ will contain all the attributes of the module.

Module.\_\_spec\_\_ gives metadata about the module.

Dir(module) gives ‘keys’ from the module.

Namespaces are dictionaries.

Where is python installed? Look at sys.prefix

Where are compiled c binaries located? Look at sys.exec\_prefix

Where does python look for imports? Sys.path

Import importlb

Importlib.import\_module(modulename) will import the module.

‘pyc’ is a compiled python file.

Finders, loaders, finders + loaders = importer

Loaders can be found at sys.meta\_path

---

import importlib

importlib.util.find\_spec("fractions")

dir(importlib)

#importlib.import\_module("fractions")

import fractions, sys

fractions.\_\_spec\_\_

sys.meta\_path

when importing a module, first it adds to sys.modules dictionary. Then adds to module.globals(). If a function is imported from a module, module is loaded in to sys.modules and function is entered in module.global().

--

from cmath import exp

import sys

print('cmath' in sys.modules)

print('cmath' in globals())

print('exp' in globals())

--

Importlib.reload(modulename)

Add files to a zip fle: python –m zipfile –c my\_app \_\_main\_\_.py timeit.py

List the zip file contents: python –m zipfile –l my\_app

Frozenimporter is a finder + loader of self contained application. It includes files, libraries, runtime env all required to run the application.

Packages are modules that can contain modules. Package.\_\_path\_\_

\_\_file\_\_ for the module points to the code for the module.

\_\_file\_\_ for the package points to the \_\_init\_\_.py file

No path for the modules.

Package has \_\_package\_\_, \_\_file\_\_, \_\_path\_\_

File has only \_\_package\_\_ and \_\_file\_\_. No \_\_path\_\_

Arranging classes and function – developers perspective and users perspective.

Use \_\_init\_\_.py to import classes and functions in users perspective for easy access.

Packages has the \_\_dict\_\_ property. It’s the namespace

If a function label has an underscore \_ in front, it will not get exported if imported through ‘import \*’

To specify export, use bare **\_\_all\_\_**= [‘symbol’, ‘symbol’,] in the module. Only these symbols will be exported from the module. Every package has a package.\_\_all\_\_ property.

Copy the globals in for loop: for k in dict(globals()).keys(): print(k)

Implicit namespace package doesn’t contain \_\_init\_\_.py. they are package like.

Type of namespace package is ‘module’.

Single namespace package can live in multiple directories

Cannot monkey around the path of the namespace package in import

To import from package.zip, just append the package to os.path.append(‘package.zip’)

Pypi has 129k packages as of now.

PEP8 – style guide // PEP 20 – zen of python // pep 484 – type hints

Books: Learning python by Mark Luts pub: O’reilly // Fluent python – Luciano Ramalho – O’reilly \*\*high rating

Python cookbook – David Beazley & Brian K Jones – O’reilly

Effective Python: 59 specific ways to write better Python – Brett Slatkin

Python in a nutshell: Alex Martelli, Anna – O’reilly

Twitter: @raymonh – Raymond Hettinger co developer of Python and implemented the latest dictionary.

#zip() with \*argument is great for transposing 2d-data

m = [(1,2,3),(4,5,6)]

print(list(zip(\*m)))

output: [(1, 4), (2, 5), (3, 6)]

Yoututbe: PyCon videos, GvR, RaymondHettinger, AlexMartelli

Blog: <http://planetpython.org>

<https://Stackoverflow.com>

python3.6 new items: Dictionaries maintain sort/key order, preserving order of \*\*kwargs in function arguments, underscores in numeric literals, fstrings, type annotations, asynchronous enhancements.

a = 1000000232

b = 2344

c = f"numerator {a} / denominator {b} = {a/b:0.7f}"

print(c)

ide: mypy, pycharm.

From 3.6, ordereddict is not required.

#update/merge dictionaries preserve order and remove duplicates from the updated dictionary

d = {"a":10,"b":20,"c":30}

e = {"d":40, "e":50, "a":100}

d.update(e)

print(d)

output: {'a': 100, 'b': 20, 'c': 30, 'd': 40, 'e': 50}

ordereddict methods: move\_to\_end(key, last=true), popitem(last=true), reversed()

#move\_to\_end

d = {"a":10,"b":20,"c":30,"d":40,"e":50,"f":60}

for i in range(len(d)-1):

key = next(iter(d.keys()))

d[key] = d.pop(key)

print(d)

#popfirst

key = next(iter(d.keys()))

d.pop(key)

print(d)

#poplast

d.popitem()

print(d)

output:

{'f': 60, 'a': 10, 'b': 20, 'c': 30, 'd': 40, 'e': 50}

{'a': 10, 'b': 20, 'c': 30, 'd': 40, 'e': 50}

{'a': 10, 'b': 20, 'c': 30, 'd': 40}

dictionary implementation: <http://code.activestate.com/recipes/578375/>

#namedtuple factory method

from collections import namedtuple

def named\_dict\_factory(class\_name, \*\*field\_names):

Struct = namedtuple("Struct", field\_names.keys())

Struct.\_\_new\_\_.\_\_defaults\_\_ = tuple(field\_names.values())

return Struct

Person = named\_dict\_factory("Person", name="Jaison",state="Kerala",country="India",age=45)

p = Person("Jacob")

print(p)

Struct(name='Jacob', state='Kerala', country='India', age=45)

In [ ]:

#format number with underscore

a = "format {:\_}".format(1000000)

print(a)

output: format 1\_000\_000

#lambda expression inside print statement.

a = 10

print(f"{(lambda x: x \*\* 2)(a) if a >5 else a}")

by default, system time is set for the random seed.

Random.see(0), random.randint(10,20), random.random(), random.shuffle(“abcd”), random.randrange(5), random.choice(sequence), random.choices(), random.sample()

randoms = []

l = [10,20,30,40]

import random

randoms = [random.choice(l) for \_ in range(10)]

print(randoms)

from timeit import timeit

sys.argv for commandline arguments

--

import argparse

parser = argparse.ArgumentParser(description="Add two numbers")

parser.add\_argument("a", help="Enter a number", type=int)

parser.add\_argument("b", help="Enter a number", type=int)

p = parser.parse\_args()

a = p.a

b = p.b

print(f"a + b = {a + b} ")

--

#like switch statement

def dow\_switch(dow):

dow\_dict = {

1: lambda: print("Sunday"),

2: lambda: print("Monday"),

3: lambda: print("Tuesday"),

4: lambda: print("Wednesday"),

5: lambda: print("Thursday"),

6: lambda: print("Friday"),

7: lambda: print("Saturday"),

'default': lambda: print("Incorrect day entered")

}

return dow\_dict.get(dow, dow\_dict['default'])()

dow\_switch(55)

1st Apr ’20 – Iteration, Generators

Protocols: sequence protocol, iterator protocol, iterable protocol, generator protocol

<https://github/fbaptise/python-deepdive>

a sequence of ordered elements. Position is important. Index start at 0

mutable sequence types: lists, byte arrays

immutable sequence types: tuple, strings, frozen sets, range, bytes. Tuples are also data structures.

Other standard types from standard library: namedtuple, deque, array module (array object)

Any sequence type is iterable. A ‘set’ is an iterable, but not a sequence. Set is not ordered.

Common sequence methods: a in s, a not in s, a + s (its concatenation), s \* n (repetition – n is an integer), len(s), min(s), max(s), s.index(x), s.index(x, i), s.index(x, I, j)

S[i], s[i:j], s[i:j:k] – k is the step. Slicing will return the same container type.

Mutable sequence types does NOT support hashing. Hash(s)

#when a mutable sequence is concatenated (applies to repletion also), its memory address is referenced in the copied object

a = [[10,20]]

b = a + a

print(b)

print(hex(id(b[0][0])))

print(hex(id(b[1][0])))

b[0][0] = 100

print(b)

a = {1:10,2:"a",3:"b","c":"c"}

'c' in a

Output: True

Cannot mix types in concatenation.

--

a = ":::".join(["a","1","@","b"]) // output: 'a:::1:::@:::b'

enumerate a sequence / s= list(enumerate(s))

s = list(enumerate((10,20,30))) output: [(0, 10), (1, 20), (2, 30)]

--

# slice from end.

s = "python"

s[5:2:-1] // output : noh

---

Slicing always creates a new object

#repetition problem of a new object in mutable object. Shallow copy and deep copy can get rid of this problem

a = [[10,20]]

b = a \* 2

a[0][0] = 100

print(b) // output: [[100, 20], [100, 20]]

concatenation does not mutate objects. It just creates another object for the variable to point to.

Mutating an object means changing the objects state without changing the object itself.

#mutating a list element

a = ["Jaiosn","Jacob"]

print(hex(id(a)))

print(hex(id(a[0])))

a[0] = 100

a.append("Valayil")

print("---------\n",hex(id(a)))

print(hex(id(a[0])))

output:

0x165de9d3048

0x165deb356b0

---------

0x165de9d3048

0x7ffea80dadf0

Mutable sequence mthods:

s.clear() // s.append(x), s.insert(i,x), extend(iterable), pop(i), remove(x), s.reverse() – inplace reversal of the elements, s.copy() – shallow copy

a=[2,3,4,5,6]

a[1:3] = {'a','b','c','d','e','f'}

print(a)

output: [2, 'c', 'a', 'd', 'f', 'b', 'e', 5, 6]

--

Constant folding is the process of recognizing and evaluating constant expression at compile time rather than computing them at runtime.

#disassemble

from dis import dis

dis(compile("(1,2,3,4,'a')", 'string', 'eval'))

compiling a list takes more effort than a tuple, because tuples are immutable

for any sequence s, the index range is given by: 0 based: 0 <= n < len(s)

shallow copy uses the same memory references in the new list from the copied list.

Deep copies require a recursive approach. Deepcopy is applicable to all objects in general.

Standard library ‘copy’ module provides generic copy and deep copy operations

Pythonic way of copying an iterable is to use list comprehension

lst = list([1,2,3,4])

cp =[e for e in lst]

print(cp)

--

Slicing only works with sequences. Slicing rely on indexing. With mutable sequence types, can assign data.

S = slice(0,2) object

lst = list([1,2,3,4,5,6,7,8,9])

s = slice(0,4)

a = lst[s]

print(a)

step value of -1 will begin from end

slice.indices()

lst = list([1,2,3,4,5,6,7,8,9])

s = slice(0,3).indices(9)

lst[-3:-9:-1] # k of step -1 iterate back. j of -9 is the 8th element from the end. i of -3 is the 3rd element from end.

Output: [7, 6, 5, 4, 3, 2]

--

xcross