Research Questions

CIST-005B
Bertrand B. Blanc
03/13/2024

Why is using "while True" not a good idea? Is there any situation in that we have to use it?

“while True” means the loop is infinite without a termination. If the designer truly had in mind to design a “forever” loop, it would make sense to get it implemented via a “while True” loop. However, in most of the cases the designer does expect to have the while-loop terminating. Hence, as a good coding practice, the termination condition should be used. Besides, having a termination condition may prevent entering in an unwanted infinite loop.

That conditional expression works well when the body of the while-loop is small enough, when the control-flow is easily executed from the first line of the body to the last line, when an index or iterator is involved, or an event expected to happen. The termination condition heavily depends on the setting of a variable within the body: forgetting this setting like “flag = False” or “idx += 1” will also end up with an infinite loop.

Besides, when the control flow becomes more cumbersome with a collection of nested conditional statements, it may be very easy to lose track of all the different control flowS. A compiled language may have some capabilities to find infinite loop at static time, however for Python, as an interpreted language it may not be that easy: the way to find such infinite loops is when exercising dynamically these paths in the code: finding all these paths to exercise all of them is a business industry on its own.

Then, I’m a proponent to “go with the flow”: writing linear simple code, embedding termination points inside the different control flow paths. Unlike primitive languages which basically only had “goto” to break a loop, modern languages have more: return, break and raise. These 3 keywords are part of the language: then let’s use them.

I start wondering whether this coding rule to avoid “while True” is not more historical than rooted in modern language semantics.

Why is using the "eval**" and "**exec**" functions a bad practice and may cause a security issue?**

Both of these Python built-in commands basically take a string as parameter and evaluate that string as a python statement. The main security issue is code injection like in SQL queries or AI user prompts. An input from a user is a string, an ill-intentioned hacker may have knowledge an eval/exec is used on such inputs and take advantage of such vulnerability to give as input some real code which will be executed. Python does implement the introspection, meaning the internal code of a program can be dynamically modified: such code injection may be quite damageable for a company.

That said, the coder may have the need to dynamically generate some code based on string computation, hence the usage of exec & eval to address that need. The introspection is part of the semantics of the language, and so are eval and exec when used wisely.

What is pdb (the Python Debugger)?

"assert" statements are native Python commands aimed at debussing the code by embedding these statements inside the code for debug purposes. They can automatically be stripped from the code using -O option or the environmental variable PYTHONOPTIMIZE=1.
Another ad-hoc custom is the usage of "print" statements or stopping the execution of some parts of the code. Such methodology disrupts the code and may have dramatic side-effects like forgetting some unwanted print statements, forgetting to reenable some parts of the code, or deleting by mistake some code.

Debuggers are tools which allow and help debugging a program. PDB is a Python library which helps instrumenting the code with "breakpoint" operator to allow inline dynamic debugging richer than mere "assert" statements. When the breakpoint is reached, the IDE prompts dynamically an "ipde" prompt requiring the coder to enter pdb code introspection commands at that specific breakpoint to investigate, for example, the content of variables.

```
Import pdb
def foo(x):
  breakpoint()
  return sum(x)

print(foo((2,3)))
```

```
>>> <ipython-input-7-784826e93a44>(3)foo()
      1 def foo(x):
      2   breakpoint()
----> 3   return sum(x)
      4
      5 print(foo((2,3)))

ipdb> display x
display x: (2, 3)
ipdb> x = (2,3,4)
ipdb> c
9
```

Documentation: https://docs.python.org/3/library/pdb.html

Why do we use "_" before the name of a variable?

The single leading underscore in front of any identifier (constant, function, class, variable) means the object is intended to be private. Such attribute is deemed private to the class or such variable is private to the module and won't be exported into other module.

Within a class, such attributes are expected to be accessed through getters/accessors and setters/modifiers. That is more of a convention since the concept of "private" doesn't exist in Python, hence still allows a coder to directly access & modify these variables from outside the class.

Linters like PyLint or other tools also interpret such identifiers as "private", hence may enforce some rules or loosen others such as not enforcing docstrings, or not reporting such identifiers via the "help" command.

What are Dunder/Magic Methods in Python? Explain with an example.

Dunder or magic methods are the special methods in Python which start and end with double-underscores aka "dunder." As examples: __init__ is the magic method which is called when an object is created from a class. __add__ is another magic prefixed method which is called thru the infix "+" sign.
Coders are prohibited to create their own identifiers looking like magic methods. Some of these magic methods are expected to be implemented by the coder when they create a new class.
These methods are not expected to be used as such, except maybe when needed like in functools.reduce.

What is the "self" in your code? Why do we use it in a method's header?

"self" as the first argument to all methods of a class, is a convention to highlight that all methods take a mandatory first argument which refers to the object itself. That argument may actually be called as we please like "this", however PEP 8 standardized the name to be "self" for class instance objects, and "cls" for class objects.

```
class Foo():
  nb_instances = 0

  @classmethod
  @property
  def num(cls):
    return cls.nb_instances

  def __init__(self, val):
    self._val = val
    Foo.nb_instances += 1

  @property
```

```
  def val(self):
    return self._val


print(Foo.num)
foo1 = Foo(2)
print(foo1.val)
print(Foo.num)
```

0
2
1

What is the difference between a list **and a** tuple**? When do we use them?**

List and tuple are both ordered collections. The content of a list can be changed whereas the content of a tuple cannot. Basically both have the same behavior and methods except for the modifiers: only the list has them.
Lists are used when the content if expected to be modified. Tuple are used when the content is not expected to be modified (which also allows under the hood some optimizations since there's no expectation to see the tuple dynamically increasing or decreasing in size).
Although the content of tuple cannot change, since the object are references, it's always possible to change the content of these references, same applies to the lists actually.

What is encapsulation**? Why do we use it?**

Encapsulation is an OOP concept aiming at bundling together all data and behavior of an object into a class, hiding the intricacies of such data aka attributes, and behaviors aka methods.
We use it as a divide-to-conquer methodology working on smaller objects and start building bigger and bigger objects interacting with the objects through their interfaces.

What is a constructor in a class and how can we define it?

A constructor in a class is the magic method __init__ which is called every time the instance of an object is created. It allows to define the initial state of that instance upon creation such as the instance attributes.

What are **Overriding** and **Overloading**?

Overriding a function or method means creating multiple functions with the same name and different signatures. Overriding is prohibited by construction in Python: it is not possible to have two separate functions with the same name and different signatures (except for standardized setters

and getters using specific decorators). That concept may be emulated thru the signature using default parameters or *args/**kargs packing.

Overloading a method is an OOP concept which means a child class may redefine or implement the content of a method previously declared or defined in a parent class. The binding of the correct method will happen at run-time to select the method from the proper class or instance class object.

```
class Foo():
  def __str__(self):
    return "I'm Foo"

class Bar(Foo):
  def __str__(self): # magic method overriding
    return f"I'm Bar and also {super().__str__()}"

print(Foo())
print(Bar())
>>> I'm Foo
>>> I'm Bar and also I'm Foo
```

What is Polymorphism**? Explain it with examples.**

Polymorphism is another OOP concept aiming at defining a collection of objects based on common features usually implemented via abstract classes. Such generic objects are later specialized according to their specifics usually implemented via inheritance: such objects are more refined, they can be concrete classes ready to be instantiated in client code, or still abstract classes which need further refinements. Hence the term polymorphism from the Greek meaning “poly” (many) and “morphism” (forms).

```
class Shape(ABC):
  @abstractmethod
  def perimeter(self):
    pass
  @abstractmethod
  def area(self):
    pass

class Quadrilateral(Shape):
  pass

class Square(Quadrilateral):
  def __init__(self, side):
    self.side = side
  def perimeter(self):
    return 4*self.side
```

```
  def area(self):
    return self.side**2

class Rectangle(Quadrilateral):
  def __init__(self, L, l):
    self.sides = (L,l)
  def perimeter(self):
    return 2*sum(self.sides)
  def area(self):
    return reduce(int.__mul__,self.sides)
```

What are the differences between a list **and a** set**?**

Both list and set are mutable collections. The list is ordered whereas the set is unordered: lists are subscriptable whereas sets are not. Lists may have many instances of the same object located at different positions whereas the set can only have a unique instance if the same object, a nifty way to uniquify the elements of a list. To enforce their difference in nature, semantical equivalent mutator have different names such as list.append for the list to add an element at the end of the list and set.add to add an element in a set.

What is the difference between the Python identity **operator (**is/is not**) and the** equality **operator (==/!=)? Where to use them and where not?**

The "is/is not" operators compare the references of objects i.e. the physical memory locations, whereas the "=/!=" operators compare the content of the object hence requiring the objects to be comparable.

A is B is equivalent to id(A) == id(B).

What is the difference between OOP and Functional programming?

Object Oriented Programming is a coding style involving objects and the semantics associated to objects such as encapsulation, attributes, methods, abstraction, polymorphism, inheritance. The instances of objects interact between each other via their interface exchanging messages or transactions. A few major Python keywords related to OOP are "class", "__init__", "@abstractmethod", "@classmethod", "def".

Functional programming is a different coding style with a different semantics. It consists in having a flow of functions whose return values, which can also be functions, are used as inputs to other functions. A few major Python keywords related to functional programming are "def", "lambda", "maps", "reduce", list comprehension. Functional programming is also called ML standing for

“meta language”, which may be confusing in the rise of “Machine Learning” which is a totally different and unrelated domain which would cover GAI as another programming paradigm.

Imperative programming is also called procedural programming. Although procedures and functions may be similarly implemented, the semantics is different and should not be confused with functional programming. Procedural programming consists in a flow of controlled sequential steps to be performed in a specific order to accomplish a purpose. A few major Python keywords related to imperative programming are “if/elif/else”, “while”, “for”, “def”.

We can notice that “def” is used as part of these 3 programming styles to implement 3 different behaviors: methods for OOP, functions for functional programming, procedures for imperative programming.

```
class Email():
  def __init__(self, user, domain):
    self.user = user
    self.domain = domain
  @property
  def email(self):
    return f'{self.user}@{self.domain}'

email=lambda user,domain:f'{user}@{domain}'

print(Email("bb", "gmail.com").email) # OOP
print(email("bb", "gmail.com")) # functional
```

What are iterators, generators, and decorators in Python?

Iterators are objects which allow iterating on themselves i.e. scanning their content sequentially. Iterators implement the magic method __iter__ and are called via the built-in Python function “iter”. Objects called in Python for-loop statements are typical examples of iterators.

Generators are objects or functions which produce a new object every time the built-in function “next” is called upon them. Generator objects shall also implement the dunder method __next__. Iterators are typical examples of generators since each of the new values received for each iteration is produced from that iterator thru a call to the “next” command.
Generators can generate objects indefinitely or terminate when they don’t have any other object to produce or raise the exception StopIteration. The statement “yield” is the blocking statement in a generator which pauses the execution of the code until resumed via the next call to “next”.

The example below highlights the underlying usage of an iterator coupled with a generator as part of a for-loop statement, hence showing the “iter”, “next” and StopIteration exception using a while-loop.

```
seq = (1, 2, 3, 4) # a basic tuple
for i in seq:
  print(i)

it = iter(seq)
while True: # no expectation for the flow to stop
  try:
    i = next(it)
    print(i)
  except StopIteration:
    break # the flow stops here
```

Decorators are shallow functions or objects wrapping other functions or methods to slightly enhance their basic behavior. Decorators are called in Python via the keyword "@" preceding the name of the decorator located just before the function to wrap. Examples of decorators are @property, @abstractmethod, @staticmethod, @x.setter. Follows an example of the creation of a decorator.

```
import functools
import time
def howlong(f):
  @functools.wraps(f)
  def wrapper(*args, **kargs):
    start = time.time()
    back = f(*args, **kargs)
    print(f'{f.__name__} was executed in {round(time.time()-start,2)}
seconds')
    return back
  return wrapper

@howlong
def foo(s):
  time.sleep(2)
  print(s)

foo('hello')
```

>>> hello
>>> foo was executed in 2.0 seconds

What are passing by value and passing by reference? Explain it with an example.

Both statements refer to the way the arguments are provided to the functions. A value is typically the content of a memory location which cannot be modified like an integer or a string. A reference is a memory location which can therefore be modified as a side-effect. The difference is similar to the question related to "==/!=" operator on values versus "is/is not" operator on references. Python passes most of the variable by reference which may cause unbecoming side-effects.

In the following example, the variable "v" is passed by reference to the function "foo" since its content can be modified. I'm not aware of any Python construct to enforce the variable to be passed by value to prevent unwanted side-effects.

```
def foo(argv, side_effect=False):
  if side_effect:
    for i,x in enumerate(argv):
      argv[i] = x*10+12
    print("inside:", argv)
    return

  argv = [x for x in range(5)]
  print("inside:", argv)
```

```
v = [0, 3]
print("value of v:", v)
foo(v)
print("value of v (after calling foo):", v)
```

```
>>> value of v: [0, 3]
>>> inside: [0, 1, 2, 3, 4]
>>> value of v (after calling foo): [0, 3]
```

```
v = [0, 3]
print("value of v:", v)
foo(v, True)
print("value of v (after calling foo):", v)
```

```
>>> value of v: [0, 3]
>>> inside: [12, 42]
>>> value of v (after calling foo): [12, 42]
```

What is a round-off **error in python? Explain it with an example.**

A round-off error in Python is actually not related to Python by itself but how the floating point arithmetic values are represented in memory with bits preventing to have a mathematically precise values. That "error" may be fixed by rounding the float with a precision based on the specifications of the problem to solve, hence the term of round-off.

```
f'{0.1:.30f}'
>>> 0.100000000000000005551115123126
```

Hence, in order to obtain a meaningful value, floats need to be used with “round” function mentioning how many digits are significant for a round-off. In the example above, the precision is up to 17 decimals.

Bitcoin crypto currency is defined with 8 decimals which correspond to its smallest unit: the Satoshi. The ETH has a precision of 18 decimals. We understand now from a banking balance sheet perspective the importance of the floating point precision.

```
print(f'0.1 BTC = {0.1:.8f}')
>>> 0.1 BTC = 0.10000000 # all good
print(f'0.1 ETH = {0.1:.18f}')
>>> 0.1 ETH = 0.100000000000000006  # not that good anymore
```

Same concerns with parameters computation in AI models. The values are kept small with the floating point precision being an important factor, to avoid parameter corruption or vanishing gradient like it used to be the case with the usage of the sigmoid functions at the time when ReLU activation function had not been rediscovered yet in 2012.

The good news is Python environment allows the setting of the floating point precision as well as provides the Decimal library to be used in lieu of native floats.

Explain map(), filter(), and reduce() in Python with Examples.

They are part of functional programming style, applying a function on the elements of an iterable object. Map and filter are generators applying a 1-variable user-defined function on each elements of the list separately. Reduce produces a final result which is the reduction of the elements of the list applied to a 2-variable user-defined function, the first element being the accumulator for the previous elements of the list.

```
from functools import reduce
seq = [x for x in range(5)]

result = 0
for x in map(lambda x:2*x, seq):
  result += x
print(result)
>>> 20

print(2*reduce(lambda x,y:x+y, seq))
>>> 20
```

Notice the Python operator “sum” was not used here to highlight the nature of the iterator “map”. “sum” may also be written with a “reduce” operator.

```
mysum = lambda L:reduce(int.__add__,L)
print(sum(list(map(lambda x:2*x, seq))))
>>> 20
print(mysum(list(map(lambda x:2*x, seq))))
>>> 20
```