Python_AQA_SpecPoints

June 26, 2020

1 Using Python to Meet the Requirement of the A Level Specifications

Python is a great learning language that is very popular with programmers in a wide range of industries, from web development to engineering and science to application development. It is suitable for rapid prototyping and the basics are easy to learn, with no boilerplate and clean syntax, while being powerful enough for enterprise and academic use.

There are some areas of the specifications though that take some thought to cover without resorting to other languages, but there are no show stoppers and few difficult areas.

1.1 Technical topics

1. arrays
2. switch/case statements
3. databases
4. records (structs)
5. functional programming

6. constants
7. class access modifiers, getters and setters

1.2 1. Arrays

1.2.1 The Array module

The base install of Python includes C-type arrays, initiated with a declaration and sporting static typing. Python arrays are suitable for OCR, but are not necessary for AQA, which only requires the arry-type *item addressing*, so lists are fine.

https://docs.python.org/3.6/library/array.html

[1]:
```
import array

print(array.typecodes)
```

```
bBuhHiIlLqQfd
```

[2]:
```
from array import array

array1 = array('l')
array2 = array('u', 'hello \u2641')
array3 = array('l', [1, 2, 3, 4, 5])
```

[3]:
```
for x in array3:
    print(x, end=", ")
```

```
1, 2, 3, 4, 5,
```

Being Python, arrays are not immutable, and many of the standard list methods apply:

[4]:
```
array4 = array('d', [1.0, 2.0, 3.14])
array4.append(2.75)
print(array4)
print("slicing: array4[2]:", array4[2])
```

```
array('d', [1.0, 2.0, 3.14, 2.75])
slicing: array4[2]: 3.14
```

1.2.2 Python Lists

Native Python lists are usable for most array work, and all of it for AQA which really expects lists to be used.

[5]:
```
myList = list(range(6))
print("myList =          ", myList)

print("myList[2]:        ", myList[2])
print("myList[1:2]:      ", myList[1:2])

myList.sort(reverse=True)
print("Reversed:         ", myList)

nestedList = [[1,2,3]] * 3
print("Nested list:      ", nestedList)
print("nestedList[1][2]:", nestedList[1][2])
```

```
myList =          [0, 1, 2, 3, 4, 5]
myList[2]:        2
myList[1:2]:      [1]
Reversed:         [5, 4, 3, 2, 1, 0]
Nested list:      [[1, 2, 3], [1, 2, 3], [1, 2, 3]]
nestedList[1][2]: 3
```

2 2. Switch/Case

Pseudocode:

```
switch entry:
    case "a":
        print("You selected A")
    case "b":
        print("You selected B")
    case "c":
        print("You selected C")
    case "d":
        print("You selected D")
    default:
        print("Unrecognised selection")
endswitch
```

Switch/case structures are not provided with a specific syntax in Python, but equivalent function can be achieved with if/elif/else

2.1 if / elif / else

[6]:
```
entry = "c"

if entry == "a":
    print("You selected A")
elif entry == "b":
    print("You selected B")
elif entry == "c":
    print("You selected C")
elif entry == "d":
    print("You selected D")
else:
    print("Unrecognised selection")
```

```
You selected C
```

2.2 Dictionaries

Dictionaries work well if a value or function is returned:

[7]:
```
entry = "c"

case = {
    'a': "You selected A",
    'b': "You selected B",
    'c': "You selected C",
```

```
    'd': "You selected D",
    }
if entry in case:
    print(case[entry])
else:
    print("Unrecognised selection")
```

```
You selected C
```

For more advanced students, lambdas allow function calls from within a compact structure:

[8]:
```
entry = "c"

case = (lambda x:
            print("You selected A") if x=='a' else
            print("You selected B") if x=='b' else
            print("You selected C") if x=='c' else
            print("You selected D") if x=='d' else
            print("Unrecognised selection") #default return
        )
case(entry)
```

```
You selected C
```

3 Databases

There are lots of options for accessing databases with Python, but SQLite3 is installed as a base package by default.

This code below will link to a database file in memory. Naming as a file instead of referencing ':memory:' will create a database file instead of one in-memory. (eg. 'testDatabase.db')

[9]:
```
import sqlite3

dbase = sqlite3.connect(':memory:')

# Create a cursor object for interaction
cursor = dbase.cursor()
cursor.execute('''
    CREATE TABLE users(id INTEGER PRIMARY KEY, name TEXT, email TEXT unique,␣
 ↪password TEXT)
                ''')
dbase.commit()

dbase.close()
```

4 Record Data Structures

AQA has in its A Level specification: ”Use records (or equivalent)”, while OCR has the phrase ”Arrays, records, lists, tuples.”

Records should have named fields which may be of different types, and ought to be immutable to enable safe passing around in a program. Im many programming cases plain *tuples* may be a suitable alternative even without the named-fields property, using implicit tuple unpacking.

4.1 Tuples

[10]:
```
catRecord1 = ('Snuffles', 'Tabby', 3, 4.2)
catRecord2 = ('Schneider', 'Black', 4, 3.6)

records = [catRecord1, catRecord2]

for cat in records:
    name, breed, age, weight = cat
    print(f"{name} is a {breed} cat and is {age} years old, weighing {weight}␣
 ↪kilos.")

print()
print('Text representation:', cat)
```

```
Snuffles is a Tabby cat and is 3 years old, weighing 4.2 kilos.
Schneider is a Black cat and is 4 years old, weighing 3.6 kilos.

Text representation: ('Schneider', 'Black', 4, 3.6)
```

4.2 Named Tuples

More useable as traditional records are named tuples, from the *collections* module.

[11]:
```
from collections import namedtuple

Cat = namedtuple('Cat', 'name breed age weight')

catRecord1 = Cat('Snuffles', 'Tabby', 3, 4.2)                    # Use␣
 ↪positional arguments ...
catRecord2 = Cat(name='Irene', age=4, weight=2, breed='Persian') # or named␣
 ↪fields

records = [catRecord1, catRecord2]

name, breed, age, weight = records[0]          # tuple unpacking again
```

```
print(f"{name} is a {breed} cat and is {age} years old, weighing {weight} kilos.
 ↪")

name = catRecord2.name                          # ... or use attribute names
print(f'Cat2\'s name is {name}.')
print()
print("Text represention:", catRecord1)
```

```
Snuffles is a Tabby cat and is 3 years old, weighing 4.2 kilos.
Cat2's name is Irene.

Text represention: Cat(name='Snuffles', breed='Tabby', age=3, weight=4.2)
```

5 Functional Programming

Haskell is a great language for demonstrating the functional programming paradigm, and Python does not have all the same structures, so we should use Haskell for this element of the specificatoin. But you can show some Haskell code re-written in Python to show it in a more familiar context.

5.1 Functions as first class objects

[12]:
```
numbers = (1, 3, 5, 6, 7, 12)
print('The numbers are', *numbers)

choice = input('Menu \n 1: minimum \n 2: maximum \nChoice:')

def func(choice):
    if choice == 1: return min
    else: return max                # returns a function. Note: no '()'

function = func(choice)
print('The result is', function(numbers))
```

```
The numbers are 1 3 5 6 7 12
Menu
 1: minimum
 2: maximum
Choice:2
The result is 12
```

5.2 High order functions

AQA require knowledge of map, filter and reduce or fold.

5.2.1 map

For example, add 5 to each element of a list with a loop, and with map:

[13]:
```
itemList = [1 ,2, 3, 4, 5]
result = []

for item in itemList:
    result.append(item + 5)

print(result)
```

```
[6, 7, 8, 9, 10]
```

[14]:
```
def func(x): return x + 5

list(map(func, [1, 2, 3, 4, 5]))
```

[14]: [6, 7, 8, 9, 10]

Anonymous lambda functions can be used in place of the function, like the Haskell function:

```
map (+5) [1 ,2, 3, 4, 5]
```

'map' returns an interator, so Python needs the 'list' function if you want it as a list. Alternatively, use a list comprehension.

[15]:
```
list(map(lambda x : x + 5, [1, 2, 3, 4, 5]))
```

[15]: [6, 7, 8, 9, 10]

[16]:
```
[x + 5 for x in [1, 2, 3, 4, 5]]    # list comprehension
```

[16]: [6, 7, 8, 9, 10]

5.2.2 Filter

Filters out values using a conditional expression, like the Haskell expression:

```
filter (>3) [1..5]
```

[17]:
```
def test(x): return x > 3

list(filter(test, range(6)))
```

[17]: [4, 5]

[18]:
```
list(filter(lambda x : x > 3, range(6)))
```

[18]: [4, 5]

[19]:
```
[x for x in [1, 2, 3, 4, 5] if x > 3]
```

[19]: [4, 5]

5.2.3 Reduce

reduce is included in the *functools* module, and operates as the Haskell *foldl*:

```
foldl (*) 1 [1, 2, 3, 4, 5, 6]
```

[20]:
```
from functools import reduce

def func(x, y): return x * y

reduce(func, [1, 2, 3, 4, 5, 6], 1)
```

[20]: 720

[21]:
```
reduce(lambda x, y : x * y, [1, 2, 3, 4, 5, 6], 1)
```

[21]: 720

6 6. Constants

Enforcing constants is not something that is normally considered with Python — the standard programming style is to use ALL_CAPS to indicate an identifier that programmers shouldn't reassign to another value. No more needs to be done to satisfy exam boards.

6.1 Defining 'Constants' with ALL CAPS

[22]:
```
MATRIX_SIZE = 3

matrix1 = [ [0]*MATRIX_SIZE for _ in range(MATRIX_SIZE)]
print(matrix1)
```

```
[[0, 0, 0], [0, 0, 0], [0, 0, 0]]
```

6.2 Defining Constants with Enum

It is possible to enforce constants if you wish, using the Enum class, which also tidies away constants into a class definition.

[23]:
```
from enum import Enum

class const(Enum):
    pi = 3.14159
    z = "zed"
    L = [1, 2, 3]

print(*const)
```

```
const.pi const.z const.L
```

The Enum class is iterable while the constants can be accessed using the name.value attributes.

[24]:
```
for C in const:
    print(f"{C} = {C.value}")

radius = 4.0
area = 2 * const.pi.value * radius**2
print(f"\nArea = {area}")

# attempt to reassign member raises an exception, remove the # below to test:
# const.e = 3
```

```
const.pi = 3.14159
const.z = zed
const.L = [1, 2, 3]

Area = 100.53088
```

7 Class Access Modifiers

Like CONSTANTS, class access modifiers do not prevent direct assess of attributes and methods, instead they indicate to programmers what should and shouldn't be accessed. Protected names are written with a single leading underscore (eg `self._data = [ ]`) with private names having double underscores as in `def __private_method(self):`.

Single underscores show in the name that it should be considered protected, while double underscored names are name-mangled to make access from outside the class instances more onerous.

[25]:
```
class Person:
    def __init__(self, name, age=None):
        self.name = name
        self._age = age                    # protected attribute

    def __lower_age(self, by_years=10):    # private method
        self._age -= by_years
```

```
staff1 = Person('Alice', age=40)
print(f"The new employee is {staff1.name}, age {staff1._age}")
```

```
The new employee is Alice, age 40
```

The private method can be accessed from outside the class, following the name mangling rules, adding in the class name, so `staff1.__lower_age()` becomes `staff._person__lower_age()`. In practice, 'protected' is sufficient for most purposes.

[26]:
```
#staff1.__lower_age(11) # will raise an error

staff1._Person__lower_age(11) # works as intended
print(f"Age is now {staff1._age}")
```

```
Age is now 29
```

7.1 Getters and Setters

Getters and setter methods are rare in Python, since direct attribute access is so easy and access cannot be prevented.

[27]:
```
class Person:
    def __init__(self, name, age=None):
        self.name = name
        self.age = age    # note, this is a public attribute as part of the␣
 ↪interface

# instantiate without an age, adding it later:

staff2 = Person('Bob')
staff2.age = 37
print(f"{staff2.name} is {staff2.age} years old")
```

```
Bob is 37 years old
```

A setter method would let you validate the age before assigning it, so setters (and getters) can be written without having to change how you access the attribute. Use the `property` function to associate these with the attribute, and mark the original attribute as protected in the init method.

When direct access is attempted, the call is intercepted and passed to the getter or setter functions. This allows you to use the simple direct assess initially, then adding the complexity of the getter/setter methods later when necessary, without breaking the common direct attribute access used elsewhere in the code base.

[28]:
```
class Person:
    def __init__(self, name, age=None):
        self.name = name
        self._age = age          # note the underscore has been added
```

```
    def get_age(self):
        print('Getter called')
        return self._age

    def set_age(self, age):
        print('Setter called')
        if age and not 0 < age < 130:
            raise ValueError("Age out of range: 0, 130 years")
        self._age = int(age)

    age = property(fget=get_age, fset=set_age)

print('Instantiating...  ', end='')
staff2 = Person('Bob')
print(f"Current age is '{staff2.age}'")

staff2.age = 37
print(f"Current age is '{staff2.age}'")
```

```
Instantiating…  Getter called
Current age is 'None'
Setter called
Getter called
Current age is '37'
```

There is some syntactic sugar to make this code easier to read so you can, instead of calling the function directly, mark the getter and setter with `@property` and `@<attribute>.setter` decorators.

[29]:
```
class Person:
    def __init__(self, name, age=None):
        self.name = name
        self._age = age         # note the underscore now

    @property
    def age(self):              # both the setter and getter functions have
        print('Getter called')  # the same name as the previously public␣
 ↪attribute
        return self._age

    @age.setter
    def age(self, age):
        print('Setter called')
        if age and not 0 < age < 130:
            raise ValueError("Age out of range: 0, 130 years")
        self._age = int(age)

print('Instantiating...  ', end='')
staff2 = Person('Bob')
```

```
print(f"Current age is '{staff2.age}'")

staff2.age = 37
print(f"Current age is '{staff2.age}'")
```

```
Instantiating…  Getter called
Current age is 'None'
Setter called
Getter called
Current age is '37'
```