

En lo profundo de Python: Una aventura a otro nivel

^aFernando Enrique Acevedo; ^bGastón Eduardo Acosta; ^cBruno Ulises Fernández;
^dKhalil Abdul Nasir; ^eLeandro Ismael Rojas;

^aUTN, “B”, facevedo326@gmail.com

^bUTN, “A”, gastoneacosta@outlook.com

^cUTN, “B”, fernandezbruno7@gmail.com

^dUTN, “A”, kanasir@hotmail.com

^eUTN, “A”, rojasleandroismael617@gmail.com

Resumen

Durante este trabajo se mostrará la estructura de funcionamiento y particularidades del lenguaje de programación Python. Lo primero que investigamos fue su forma de funcionamiento, descubriendo que es un lenguaje completamente orientado a objetos y muy dinámico, en el cual sus variables no necesitan si quiera ser definidas, lo que ayuda al programador a reducir tiempos y costos durante su trabajo, aparte de no preocuparse por si necesita cambiar los tipos de datos de sus variables, ya que es el mismo lenguaje el que se encarga de todo esto. Otros detalles que se mostrarán en este resumen son, la sobrecarga, el polimorfismo, y su forma de encarar el código, como así también, sus ventajas y desventajas.

1. Introducción

Python es considerado un lenguaje muy particular puesto que tiene características que pocos lenguajes poseen, una de las principales es la herencia multiple. Es un lenguaje 100% orientado a objetos. Soporta programación imperativa, orientación a objetos, y, en menor medida, programación funcional.

2. Desarrollo

Tipos de datos

Todos los elementos en Python son objetos, y cada uno de estos es de un tipo específico, el cual determina las operaciones que podrán ser aplicadas a ese tipo de datos en concreto.

En Python, los tipos de datos que nos podemos encontrar son

Enteros: Números que pertenecen al conjunto de los números enteros

```
1 x = 5
2 print (type(x))
3                                     <class 'int'>
4 y = 120
5 print (type(x)) >>>               <class 'int'>
```

Flotantes: Números pertenecientes al conjunto de los números reales.

```

1  x = 5.5
2  print (type(x))
3                                     <class 'float'>
4  y = 120.2                         <class 'float'>
5  print (type(x)) >>>

```

Complejos: Números pertenecientes al conjunto de números complejos.

```

1  x = 5J
2  print (type(x))
3                                     <class 'complex'>
4  y = 120J                         <class 'complex'>
5  print (type(x)) >>>

```

Cadenas de texto: Cadenas de caracteres.

```

1  x = 'Hola profe!'
2  print (type(x))
3                                     <class 'str'>
4  y = 'Esto es un ejemplo'         <class 'str'>
5  print (type(x)) >>>

```

Booleanos: Variables que pueden tomar solo 2 valores (Binario).

```

1  x = True
2  print (type(x))
3                                     <class 'bool'>
4  y = False                       <class 'bool'>
5  print (type(x)) >>>

```

Listas: Conjunto de elementos cualesquiera, pueden ser de diferentes tipos.

```

1  x = ["D", 2, 1.2, -10]
2  print (type(x))
3                                     <class 'list'>
4  y = [72, 5.3, 'HoLa', -10]      <class 'list'>
5  print (type(x)) >>>

```

Tuplas: Conjunto ordenado e inmutable de elementos del mismo o diferente tipo.

```
1 x = ("N", 2, 1.1, -5)
2 print (type(x))
3
4 y = (5.4, 2, -1, 'Jose')
5 print (type(x))
```

<class 'tuple'>
<class 'tuple'>
>>>

Diccionarios: Estructuras de datos que permite guardar un conjunto no ordenado de pares.

```
1 x = {"Ronaldo":6.75, "Fortnite":1.2}
2 print (type(x))
```

<class 'dict'>
>>>

None: Representa el valor vacío.

Equivalencia de tipos

En Python no es necesario que el usuario defina a que tipo pertenece una determinada variable, sino que su tipo se determina en tiempo de ejecución según el tipo de valor al que se asigne, debido a que Python es un lenguaje de tipo dinámico, las variables se autodefinen.

Por ejemplo, si definimos:

```
1 x = 'Hola Profe'
2 print (type(x))
```

Nunca aclaramos que tipo de dato es, sino que el intérprete ya lo asume viendo qué tipo de dato necesita al realizar alguna operación.

Si asignamos otro tipo de dato a la misma variable, que ya tenía un valor asignado con un determinado tipo, no veremos error, debido a que Python es el que define el tipo de datos de las variables cuando las usa, no deja “estático” su tipo, por ejemplo, podemos asignar 2 valores de tipos diferentes a la misma variable, y no ocurrirá nada malo, por ejemplo:

```
1 x = 'Buen día doña Gertrudis'
2 print (type(x))
3 x = 5.5
4 print (type(x))
5
```

Al evaluar el tipo de datos de la variable x, aunque sea la misma variable, tendrá distintos tipos, porque el mismo programa decidió que así fuese.

```
<class 'str'>
<class 'float'>
>>>
```

Sistemas de tipos

Python, conocido también ser de tipado dinámico lo cual presenta flexibilidad y facilidad para implementación y para su aprendizaje.

Con respecto al tipado dinámico, Python logra inferir los tipos de los objetos que no se han definido, así mismo, es necesario considerar que este tipado dinámico puede ocasionar problemas en caso de que los objetos puedan adquirir tipos que no sean los correctos.

Haciendo un ejemplo de esa idea mencionada anteriormente, se muestra la siguiente porción de código:

A screenshot of a code editor window titled 'Prueba2.py'. The code defines a function named 'saludo' that takes a parameter 'nombre' and returns a string formatted as 'hola' followed by the value of 'nombre'. The function is defined on two lines: line 1 is 'def saludo(nombre):' and line 2 is 'return 'hola'.format(nombre)'. Line 3 is empty.

```
1 def saludo(nombre):
2     return 'hola'.format(nombre)
3
```

Esta función debería imprimir por pantalla con la frase “hola” seguido del nombre. Pero en ningún momento se hace referencia al tipo de datos que puede alojar esa variable “nombre”, es decir, no hace un control de tipo.

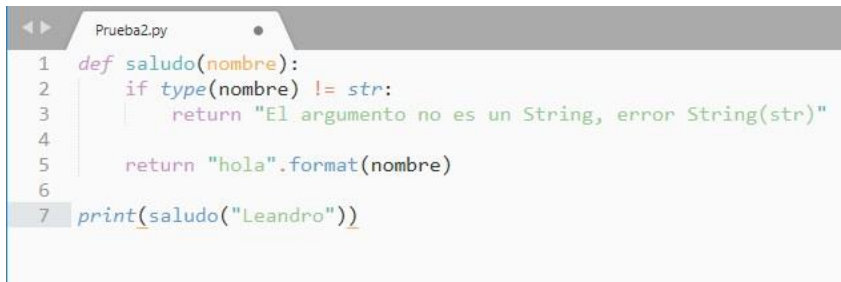
Otro caso valido seria el siguiente a continuación, pero no es el uso o respuesta que se espera.

A screenshot of a code editor window titled 'Prueba2.py'. The code defines a function 'saludo' and then calls it twice. Line 1 is 'def saludo(nombre):', line 2 is 'return 'hola'.format(nombre)', line 3 is empty, line 4 is 'print(saludo("Leandro"))', and line 5 is 'print(saludo(1))'.

```
1 def saludo(nombre):
2     return 'hola'.format(nombre)
3
4 print(saludo("Leandro"))
5 print(saludo(1))
```

Ambos contenidos son acertados, pero no cumple con la función para la cual se escribió la sentencia, es decir, devolver un string.

Considerando otro caso con un grado de complejidad más alto, podría considerarse la situación que se defina el tipo de dato que la variable “nombre” debe tener. Para este caso vamos a definir que ésta variable contenga un String



```

1 def saludo(nombre):
2     if type(nombre) != str:
3         return "El argumento no es un String, error String(str)"
4
5     return "hola".format(nombre)
6
7 print(saludo("Leandro"))

```

Polimorfismo

El polimorfismo es la propiedad por la que es posible enviar mensajes sintácticamente iguales a objetos de tipos distintos, es decir que podrán obtenerse distintos resultados según la clase del objeto.

En un lenguaje de programación orientada a objetos, que cuenta con un sistema de tipos dinámico, se denomina polimorfismo a la capacidad que tienen los objetos de una clase de responder al mismo mensaje o evento en función de los parámetros utilizados durante su invocación. Un objeto polimórfico es una entidad que puede contener valores de diferentes tipos durante la ejecución del programa. se requiere que los objetos que se utilizan de modo polimórfico sean parte de una jerarquía de clases.

En simples palabras el polimorfismo es cuando dos o más objetos entienden los mismos mensajes.

Polimorfismo en Python

Es una característica potente utilizar ya que es un lenguaje de tipado dinámico y sencillo, es decir que no tenemos que especificar de qué tipo es un objeto al ahora que se debe usar, simplificando, así las cosas

En Python es posible recorrer cualquier tipo de secuencia (ya sea una lista, una tupla, un diccionario, un archivo o cualquier otro tipo de secuencia) utilizando la misma estructura de código, como por ejemplo (for elemento in secuencia).

Un bloque de código será polimórfico cuando dentro de ese código se realicen llamadas a métodos que puedan estar redefinidos en distintas clases.

Ejemplo:

Una fábrica ensambladora de electrodomésticos posee varios métodos de ensamblaje para sus electrodomésticos. Como, por ejemplo, heladera, microondas y televisor.

En la Fig. N.º 1, las clases heladera, microondas y televisor tendrán su manera de “ensamblar_partes ()” que es el método que les corresponde.

```

class heladera():

    def ensamblar_partes(self):
        print("Ensamblando partes de heladera\n")

class microondas():

    def ensamblar_partes(self):
        print("Ensamblando partes de microondas\n")

class televisor():

    def ensamblar_partes(self):
        print("Ensamblando partes de televisor\n")

```

Fig. N.º 1

En la Fig. N.º 2, muestra un menú que facilitara el uso del programa.

```
cond=True
while cond == True:
    miElectrodomestico = input("Ingrese un electrodomestico a ensamblar\n")
    if miElectrodomestico == 'heladera':
        ensamblar_electrodomestico(heladera())
    elif miElectrodomestico == 'microondas':
        ensamblar_electrodomestico(telefono())
    elif miElectrodomestico == 'televisor':
        ensamblar_electrodomestico(televisor())
    elif miElectrodomestico == 'salir':
        cond=False
        print("Salir")
    else:
        print("Elección inválida. Ingrese nuevamente el nombre del electrodomestico")
```

Fig. N.º 2

```
def ensamblar_electrodomestico(electrodomestico):
    electrodomestico.ensamblar_partes()
```

Fig. N.º 3

Por otro lado, la función “ensamblar_electrodomestico(electrodomestico)” va a recibir un objeto por parámetro “electrodomestico” que es de tipo genérico, puede ser los electrodomésticos ya mencionados u otros que estén previamente declarados, donde utilizara ese objeto que recibe como parámetro para llamar al método “ensamblar_partes ()”.

Como un objeto, puede tener la capacidad de cambiar de forma y de comportarse de diferente forma, dependiendo el contexto, va a saber a qué método ensamblar_parte () tiene que llamar.

El polimorfismo ocurre en la Fig. N.º 4 en el parámetro de la función “ensamblar_electrodomestico (electrodomestico)” cuando al llamar a este método y pasarle por parámetro el objeto, en este caso la clase “heladera ()”, este se almacena dentro de “electrodomestico” que mediante el polimorfismo hace que se transforme en un objeto de tipo “heladera ()” en lo cual llama al método ensamblar_partes () de la clase mencionada.

```

class heladera():
    def ensamblar_partes(self):
        print("Ensamblando partes de heladera\n")

class microondas():
    def ensamblar_partes(self):
        print("Ensamblando partes de microondas\n")

class televisor():
    def ensamblar_partes(self):
        print("Ensamblando partes de televisor\n")

def ensamblar_electrodomestico(electrodomestico):
    electrodomestico.ensamblar_partes()

cond=True
while cond == True:
    miElectrodomestico = input("Ingrese un electrodomestico a ensamblar\n")
    if miElectrodomestico == 'heladera':
        ensamblar_electrodomestico(heladera())
    elif miElectrodomestico == 'microondas':
        ensamblar_electrodomestico(microondas())
    elif miElectrodomestico == 'televisor':
        ensamblar_electrodomestico(televisor())
    elif miElectrodomestico == 'salir':
        cond=False
        print("Salir")
    else:
        print("Elección inválida. Ingrese nuevamente el nombre del electrodomestico")

```

Fig. N.º 4

Al ejecutar el programa ingresando los objetos que queremos ensamblar sus partes podemos ver en la Fig. N.º 5 que el programa sabe que método utilizar dependiendo el objeto que ingrese el usuario y lo muestra en pantalla.

```

>>>
RESTART: D:\Compu anterior\Ingenieria en Sistemas de Informacion\Paradigmas\Python\paper\Ejemplo_polimorfismo.py
Ingrese un electrodomestico a ensamblar
microondas
Ensamblando partes de microondas

Ingrese un electrodomestico a ensamblar
heladera
Ensamblando partes de heladera

Ingrese un electrodomestico a ensamblar
microondas
Ensamblando partes de microondas

Ingrese un electrodomestico a ensamblar
televisor
Ensamblando partes de televisor

Ingrese un electrodomestico a ensamblar
salir
Salir
>>> |

```

Fig. N.º 5

Herencia

Para hablar de herencia, en principio es conveniente definirla de una manera sencilla. Es una propiedad que permite que los objetos sean creados a partir de otros ya existentes, obteniendo características (métodos y atributos) similares a los ya existentes. Es la relación entre una clase general y otra clase más específica. Es un mecanismo que nos permite crear clases derivadas a partir de clase base. Nos permite compartir automáticamente métodos y datos entre clases subclases y objetos.

Una vez definida brevemente, hay que hablar del tipo de herencia que soporta el lenguaje Python. Existen dos tipos de herencias, la herencia simple y la múltiple. Éste lenguaje posee un sistema de herencia múltiple. La herencia múltiple se refiere a que una clase hija o también llamada subclase, puede heredar atributos y métodos de dos o más clases padres o superclases.

Éste lenguaje, como se dijo anteriormente, posee un sistema de herencia múltiple, pero también se pueden hacer herencias simples. Para demostrar esto, se dejan los siguientes códigos:

Herencia simple

```
Prueba.py x
1 class Vehiculos():
2     def __init__(self, marca, modelo):
3
4         self.marca = marca
5         self.modelo = modelo
6         self.enmarcha = False
7         self.acelera = False
8         self.frena = False
9
10    def arrancar(self):
11        self.enmarcha = True
12
13    def acelerar(self):
14        self.acelera = True
15
16    def frenar(self):
17        self.frena = True
18
19    def estado(self):
20        print("Marca: ", self.marca, "\nModelo: ", self.modelo, "\nEn marcha: ",
21              self.enmarcha, "\nAcelera: ", self.acelera, "\nFrena: ", self.frena)
22
23    class Moto(Vehiculos):
24        pass
25
26    miMoto = Moto("Honda", "CBR 1000")
27
28    miMoto.estado()
29
30
31
32
33
34
35
36
37
38
39
40
41
42
43
44
45
46
47
48
49
50
51
52
53
54
55
56
57
58
59
60
61
62
63
64
65
66
67
68
69
70
71
72
73
74
75
76
77
78
79
80
81
82
83
84
85
86
87
88
89
90
91
92
93
94
95
96
97
98
99
100
101
102
103
104
105
106
107
108
109
110
111
112
113
114
115
116
117
118
119
120
121
122
123
124
125
126
127
128
129
130
131
132
133
134
135
136
137
138
139
140
141
142
143
144
145
146
147
148
149
150
151
152
153
154
155
156
157
158
159
160
161
162
163
164
165
166
167
168
169
170
171
172
173
174
175
176
177
178
179
180
181
182
183
184
185
186
187
188
189
190
191
192
193
194
195
196
197
198
199
200
201
202
203
204
205
206
207
208
209
210
211
212
213
214
215
216
217
218
219
220
221
222
223
224
225
226
227
228
229
230
231
232
233
234
235
236
237
238
239
240
241
242
243
244
245
246
247
248
249
250
251
252
253
254
255
256
257
258
259
260
261
262
263
264
265
266
267
268
269
270
271
272
273
274
275
276
277
278
279
280
281
282
283
284
285
286
287
288
289
290
291
292
293
294
295
296
297
298
299
300
301
302
303
304
305
306
307
308
309
310
311
312
313
314
315
316
317
318
319
320
321
322
323
324
325
326
327
328
329
330
331
332
333
334
335
336
337
338
339
340
341
342
343
344
345
346
347
348
349
350
351
352
353
354
355
356
357
358
359
360
361
362
363
364
365
366
367
368
369
370
371
372
373
374
375
376
377
378
379
380
381
382
383
384
385
386
387
388
389
390
391
392
393
394
395
396
397
398
399
400
401
402
403
404
405
406
407
408
409
410
411
412
413
414
415
416
417
418
419
420
421
422
423
424
425
426
427
428
429
430
431
432
433
434
435
436
437
438
439
440
441
442
443
444
445
446
447
448
449
450
451
452
453
454
455
456
457
458
459
460
461
462
463
464
465
466
467
468
469
470
471
472
473
474
475
476
477
478
479
480
481
482
483
484
485
486
487
488
489
490
491
492
493
494
495
496
497
498
499
500
501
502
503
504
505
506
507
508
509
510
511
512
513
514
515
516
517
518
519
520
521
522
523
524
525
526
527
528
529
530
531
532
533
534
535
536
537
538
539
540
541
542
543
544
545
546
547
548
549
550
551
552
553
554
555
556
557
558
559
560
561
562
563
564
565
566
567
568
569
570
571
572
573
574
575
576
577
578
579
580
581
582
583
584
585
586
587
588
589
590
591
592
593
594
595
596
597
598
599
600
601
602
603
604
605
606
607
608
609
610
611
612
613
614
615
616
617
618
619
620
621
622
623
624
625
626
627
628
629
630
631
632
633
634
635
636
637
638
639
640
641
642
643
644
645
646
647
648
649
650
651
652
653
654
655
656
657
658
659
660
661
662
663
664
665
666
667
668
669
670
671
672
673
674
675
676
677
678
679
680
681
682
683
684
685
686
687
688
689
690
691
692
693
694
695
696
697
698
699
700
701
702
703
704
705
706
707
708
709
710
711
712
713
714
715
716
717
718
719
720
721
722
723
724
725
726
727
728
729
730
731
732
733
734
735
736
737
738
739
740
741
742
743
744
745
746
747
748
749
750
751
752
753
754
755
756
757
758
759
760
761
762
763
764
765
766
767
768
769
770
771
772
773
774
775
776
777
778
779
780
781
782
783
784
785
786
787
788
789
790
791
792
793
794
795
796
797
798
799
800
801
802
803
804
805
806
807
808
809
810
811
812
813
814
815
816
817
818
819
820
821
822
823
824
825
826
827
828
829
830
831
832
833
834
835
836
837
838
839
840
841
842
843
844
845
846
847
848
849
850
851
852
853
854
855
856
857
858
859
860
861
862
863
864
865
866
867
868
869
870
871
872
873
874
875
876
877
878
879
880
881
882
883
884
885
886
887
888
889
890
891
892
893
894
895
896
897
898
899
900
901
902
903
904
905
906
907
908
909
910
911
912
913
914
915
916
917
918
919
920
921
922
923
924
925
926
927
928
929
930
931
932
933
934
935
936
937
938
939
940
941
942
943
944
945
946
947
948
949
950
951
952
953
954
955
956
957
958
959
960
961
962
963
964
965
966
967
968
969
970
971
972
973
974
975
976
977
978
979
980
981
982
983
984
985
986
987
988
989
990
991
992
993
994
995
996
997
998
999
1000
```

```
Marca: Honda
Modelo: CBR 1000
En marcha: False
Acelera: False
Frena: False
[Finished in 0.1s]
```

En ésta porción de código se puede observar que hay dos clases, una que se llama “Vehículos” y otra “Moto”, donde podemos ver que ésta última tiene entre paréntesis el nombre de la clase definida primeramente. Ésta es la sintaxis que posee el lenguaje para Python para declarar una herencia, donde “Moto” está heredando todos los atributos y métodos de “Vehículos”.

Se puede observar también, que la clase “Vehículos” posee 5 métodos y 5 atributos, donde el primer método es el constructor, que tiene una sintaxis un poco diferente a los demás, siempre comienza con dos guiones bajos la palabra init y termina nuevamente con dos guiones bajos.

En la clase “Moto” podemos ver que hay una instrucción “pass”, ésta es una instrucción nula y se utiliza cuando se requiere por sintaxis una declaración, pero no se quiere ejecutar ningún comando o código. Como ya heredamos todo lo necesario de la súper clase, no necesitábamos agregar más nada. Claramente, éste es un ejemplo bastante reducido, ya que demasiado sentido no tendría hacer esto, simplemente podría instanciarse la clase “Vehículos” y ya, pero se lo hizo de esta manera para explicar de manera sencilla el concepto. “Moto” tendría sus propios atributos y métodos, que son exclusivamente de ésta.

Herencia múltiple

Lo explicado anteriormente, simplemente fue para introducirnos en el concepto de herencia y ver un ejemplo en concreto. Pero realmente la potencia de este lenguaje en cuanto a herencia, es justamente el sistema de herencia múltiple que posee. Éste es uno de los pocos lenguajes que poseen éste sistema, puesto que suelen producirse ambigüedades y esto genera complicaciones a la hora de la compilación o interpretación, y por supuesto también, a la hora de poder comprender con facilidad un código extenso.

A continuación, se muestra un ejemplo en concreto donde se podrá visualizar el uso de éste sistema de herencia

```
Prueba.py x
1 class Vehiculos():
2     def __init__(self, marca, modelo):
3
4         self.marca = marca
5         self.modelo = modelo
6         self.enmarcha = False
7         self.acelera = False
8         self.frena = False
9
10    def arrancar(self):
11        self.enmarcha = True
12
13    def acelerar(self):
14        self.acelera = True
15
16    def frenar(self):
17        self.frena = True
18
19    def estado(self):
20        print("Marca: ", self.marca, "\nModelo: ", self.modelo, "\nEn marcha: ",
21              self.enmarcha, "\nAcelera: ", self.acelera, "\nFrena: ", self.frena)
22
23    class VElectricos():
24        def __init__(self, autonomia):
25            self.autonomia = autonomia
26
27        cargando = False
28        def cargaEnergia():
29            self.cargando = True
30
31    class BiciElectrica(VElectricos, Vehiculos):
32        hwilly = False
33        def willy(self):
34            self.hwilly = True
35        def estado(self):
36            print("Autonomia: ", self.autonomia, "\nCargando: ", self.cargando,
37                  "\nHace Willy: ", self.hwilly)
38
39    miBici = BiciElectrica(150)
40    miBici.willy()
41    miBici.estado()
42
43
Autonomia: 150
Cargando: False
Hace Willy: True
[Finished in 0.2s]
```

En éste código se puede apreciar que hay dos clases padres “Vehículos” y “VElétricos”, y una clase hija o subclase “BiciElectrica” la cual hereda los atributos y métodos de ambas clases padres. Hay algo muy importante a resaltar, la pregunta que se puede llegar a plantear en esta situación es: ¿Qué constructor hereda la clase hija? O si hay dos métodos con el mismo nombre en ambas clases padres ¿Cuál se ejecutaría al momento de ser llamado?, Python resuelve esta problemática de una manera muy sencilla, lo hace dándole toda la prioridad a la clase que se pase primero como parámetro, en este caso particular, “VElectricos”. Es por esto, que a la hora de instanciar la clase hija, se le pasa como argumento “150”, porque en el constructor de “VElectricos” solo necesita un solo parámetro.

A continuación, se muestra un ejemplo de que ocurriría si se intentaran pasar más argumentos como parámetros a la hora de instanciar la subclase, es decir, si se interpretara que la subclase heredara el constructor de la clase que se haya definido primero en el código, en este caso, “Vehículos”.

```
Prueba.py x
1 class Vehiculos():
2     def __init__(self, marca, modelo):
3
4         self.marca = marca
5         self.modelo = modelo
6         self.enmarcha = False
7         self.acelera = False
8         self.frena = False
9
10    def arrancar(self):
11        self.enmarcha = True
12
13    def acelerar(self):
14        self.acelera = True
15
16    def frenar(self):
17        self.frena = True
18
19    def estado(self):
20        print("Marca: ", self.marca, "\nModelo: ", self.modelo, "\nEn marcha: ",
21              self.enmarcha, "\nAcelera: ", self.acelera, "\nFrena: ", self.frena)
22
23 class VElectricos():
24     def __init__(self, autonomia):
25         self.autonomia = autonomia
26
27     def cargaEnergia():
28         self.cargando = True
29
30 class Biciflectrica(VElectricos, Vehiculos):
31     pass
32
33 miBici = Biciflectrica("Orbea", "HJ50")
34
File "C:\Users\Khalil\Desktop\Python\Prueba.py", line 33, in <module>
Traceback (most recent call last):
  File "C:\Users\Khalil\Desktop\Python\Prueba.py", line 33, in <module>
    miBici = Biciflectrica("Orbea", "HJ50")
TypeError: __init__() takes 2 positional arguments but 3 were given
[Finished in 0.1s with exit code 1]
[shell_cmd: python -u "C:\Users\Khalil\Desktop\Python\Prueba.py"]
[dir: C:\Users\Khalil\Desktop\Python]
[path: C:\Program Files (x86)\Razer Chroma SDK\bin;C:\Program Files\Razer Chroma SDK\bin;C:\Windows\system32;C:\Windows;C:\Windows\System32\Wbem;C:\Windows\System32\WindowsPowerShell\v1.0\;C:\Windows\System32\OpenSSH\;C:\Program Files (x86)\NVIDIA Corporation\PhysX\Common;C:\Users\Khalil\AppData\Local\Microsoft\WindowsApps;C:\Users\Khalil\AppData\Local\Programs\Python\Python37-32\Scripts\;C:\Users\Khalil\AppData\Local\Programs\Python\Python37-32\;C:\Users\Khalil\AppData\Local\Microsoft\WindowsApps;]
```

Claramente se puede notar que da un error, y como se dijo anteriormente, esto se debe a que la primera clase pasada como parámetro tiene prioridad, por ende, hereda su constructor.

Alcance

Para introducirnos en lo que es el alcance, primeramente, hay que definirlo de una manera sencilla y clara. El alcance es la porción del programa sobre el cuál una declaración de variable es efectiva, es decir, se puede acceder, modificar y/o usar su valor. Si se quisiera utilizar su valor y se trata de acceder a ésta desde un lugar diferente al cual fue declarado, el compilador devolvería un error, puesto que, para él, ésta no es alcanzable.

A continuación, se mostrará una porción muy sencilla de código en Python y explicaremos algunos conceptos.

```
Prueba1.py x
1 def evaluacion(nota):
2     valoracion = "Aprobado"
3     if nota < 6:
4         valoracion = "Desaprobado"
5     return valoracion
6
7
8 print(evaluacion(8))
9
```

```
Aprobado
[Finished in 0.1s]
```

Se puede decir que la porción de código de la función “evaluación” es un ámbito, pero a su vez, la porción de código del condicional “if” también es un ámbito dentro del ámbito más grande de la función que lo contiene. Algo particular ocurre en estos ámbitos de Python, y esto es, que si se declara una variable dentro del ámbito de la función, ésta puede ser alcanzada dentro del condicional, pero si ésta es definida dentro del ámbito del condicional, la variable no puede ser

alcanzada dentro de la función y fuera del ámbito del condicional. Porque se dice que ésta variable no es alcanzable en éste ámbito. Para entenderlo mejor, se muestra el siguiente ejemplo:



```
1 def evaluacion(nota):
2     valoracion = "Aprobado"
3     if nota < 6:
4         valoracion = "Desaprobado"
5         recursar = 1
6
7     if recursar = 1:
8         return "Recurso"
9     return valoracion
10
11
12 print(evaluacion(8))
13
```

File "C:\Users\Khalil\Desktop\Python\Prueba1.py", line 7

File "C:\Users\Khalil\Desktop\Python\Prueba1.py", line 7
if recursar = 1:
 ^
SyntaxError: invalid syntax
[Finished in 0.1s]

En esta porción de código, a diferencia de la anterior, se puede ver que se definió una variable llamada “recursar” dentro del ámbito del condicional. Y también se agregó otro condicional, que también posee su propio ambiente. Dentro del ambiente del segundo condicional, se intentó utilizar su valor para realizar una comparación, es en éste momento, donde el compilador tira un error, en donde indica que ésta variable no es alcanzable. Es como se dijo anteriormente, las variables en Python solo pueden ser alcanzadas dentro del ámbito donde fue definida, en cualquier otro ámbito, que no sea interno del ámbito mayor, no puede ser alcanzada.

Paso de parámetros

Los parámetros son variables locales a los que se les asigna un valor antes de comenzar la ejecución del cuerpo de una función. Su ámbito de validez, por tanto, es el propio cuerpo de la función. El mecanismo de paso de parámetros a las funciones es fundamental para comprender el comportamiento de los programas en Python.

Por lo general, en los diferentes lenguajes de programación, se habla del paso de parámetros por valor y por referencia. En el paso de parámetros por copia/valor, una función no puede cambiar el valor de las variables que recibe por fuera de su ejecución: al intentar hacerlo meramente se alteraría las copias locales de dichas variables. Por el contrario, al pasar por referencia, una función obtiene acceso directo a las variables originales, permitiendo así su modificación.

En Python, en cambio, no se concibe la lógica de paso por valor/referencia, debido a que el lenguaje no trabaja con el concepto de variables, sino con objetos y referencias. Al realizar la asignación $N = 1$ no se dice que “N contiene el valor 1” sino que “N referencia a 1”. Una variable es efectivamente solo un nombre, un identificador, que está asociado a la referencia de un objeto en memoria, el cual sirve solo para acceder a él. Un mismo objeto puede tener diversos nombres asociados a él. Así, en comparación con otros lenguajes, podría decirse que en Python los parámetros siempre se pasan por referencia.

Es importante tener en cuenta el concepto de mutabilidad de los objetos al momento de estudiar el comportamiento de las funciones en este lenguaje, a partir del tipo de objeto que tengamos (mutable o inmutable) se puede distinguir la siguiente clasificación:

```

1 def f(x):
2     print("Id del argumento x antes de modificar: ", id(x))
3     x += [7]
4     print("Id del argumento x después de modificar: ", id(x))
5
6 l = [6]
7 print("Id de variable l: ", id(l))
8 print("contenido: ", l)
9 f(l)
10 print("contenido: ", l)

```

```

Id de variable l: 9848272
contenido: [6]
Id del argumento x antes de modificar: 9848272
Id del argumento x después de modificar: 9848272
contenido: [6, 7]
[Finished in 0.1s]

```

Paso de parámetros con objeto inmutable.

Los números enteros, los de coma flotante, las cadenas y otros objetos son inmutables. Es decir, una vez creados, su

valor no puede ser modificado. Desde la perspectiva del lenguaje, si se realiza la asignación $N=1$ y luego $N=2$, no se está cambiando el valor de N , de 1 a 2, sino que esto implica en realidad siempre crear un nuevo objeto y una asignación de la nueva referencia a la variable. Por lo tanto, en el momento que dentro de la función se realice algo como $N=2$, siendo N un entero pasado como argumento, el nombre N pasa a estar asociado a un nuevo objeto en memoria.

En el presente ejemplo, se puede observar que al ejecutarlo $N1, N2, N3 = 4, 5, 6$ está creando nuevas referencias para los números 4, 5 y 6 dentro de la función f con los nombres $N1, N2$ y $N3$. Esto queda demostrado al usar la función id , la cual nos permite acceder al número de identificador de los objetos (dentro y fuera de la ejecución de la función), gracias al cual podemos observar que los respectivos no coinciden. Es decir, no se está referenciando al mismo objeto.

Objetos inmutables

Bool, bytes, complex, decimal, int, float, frozenset, str/Unicode, tuple, rango.

Paso de parámetros con objeto mutable:

Las listas son objetos mutables, esto implica que las listas que se modifiquen dentro de una función son los mismos objetos que los que se pasan como argumento. Es decir que dentro de la función no se crea un nuevo objeto, sino que es modificado directamente.

Al realizar la inspección del siguiente código, se puede verificar lo planteado anteriormente. Efectivamente, la lista original es modificada dentro la función. Se puede observar que los identificadores, dentro y fuera, son los mismos, lo cual confirma el hecho de que no se creen nuevos objetos, sino que se modifican directamente.

```

1 def f(N1, N2, N3):
2     N1, N2, N3 = 4, 5, 6
3     print(N1, N2, N3)
4     print(id(N1), id(N2), id(N3))
5     # No altera los objetos originales.
6
7
8 N1, N2, N3 = 1, 2, 3
9 f(N1, N2, N3)
10 print('-----')
11 print(N1, N2, N3)
12 print(id(N1), id(N2), id(N3))

```

```

4 5 5
1701103792 1701103808 1701103824
-----
1 2 3
1701103744 1701103760 1701103776
[Finished in 0.1s]

```

En el caso de no querer que se modifique la lista original, lo que se procede a realizar es: pasar a la función una copia de esta, o crearla dentro la propia función.

Objetos mutables:

Bytearray, dict, list, set.

Orden de evaluacion

El orden de evaluación se refiere exactamente al momento en que cada parámetro actual se evalúa cuando se llama a una abstracción. Existen básicamente dos posibilidades:

- Evaluar el parámetro actual en el punto de la llamada.
- Retardar su evaluación hasta que el argumento realmente se usa.

El primer orden de evaluación se denomina “Evaluación Impaciente” (o evaluación en orden aplicativo). Se evalúa el parámetro actual una sola vez, y en efecto se sustituye el resultado en cada ocurrencia del parámetro formal.

El segundo orden de evaluación se denomina “Evaluación en orden normal”. No se evalúa inmediatamente el parámetro actual, sino que se sustituye el parámetro actual por cada ocurrencia del parámetro formal.

Python utiliza el primer orden de evaluación (Evaluación impaciente), lo cual está comprobado a continuación:

En el siguiente ejemplo se detalla una función, la cual toma como parámetro a una función sin fin, la cual se devuelve un mensaje. Dicha función *bucle* no es utilizada para ningún fin en la función *f*, por la cual no es necesaria la llamada a la misma. Al realizar la evaluación impaciente, y acceder a la función *bucle*, termina por no evaluar la función *f*, mostrando así al ejecutar constantemente el mismo mensaje.



```
1 def bucle(a):
2     while a==0:
3         print("evaluacion impaciente")
4 def f(func):
5     print("evaluacion normal/peresosa")
6
7
8
9 f(bucle(0))
```

evaluacion impaciente
evaluacion impaciente
evaluacion impaciente
evaluacion impaciente
evaluacion impaciente
evaluacion impaciente

3. Conclusiones

A la conclusión que se puede llegar es que Python es un lenguaje de programación simple y claro, que busca diferenciarse de los demás lenguajes. El hecho de que obligue al programador a escribir con bloques logra que estos adquieran la costumbre de escribir códigos claros y reutilizables. Es un lenguaje de programación de libre distribución. Soporta programación imperativa, orientación a objetos, y, en menor medida, programación funcional. Tiene tipos dinámicos, conversión de tipos explícita, maneja excepciones y es multiplataforma.

Lo expuesto en el presente informe, es una síntesis de la gran capacidad que posee este lenguaje, por ende, se anima a los lectores a investigar en páginas webs, libros de programación y otros medios de información acerca de las funciones que posee Python, para así lograr una mayor comprensión de la gran capacidad de este lenguaje de programación

Bibliografia

- [1] - <https://entrenamiento-python-basico.readthedocs.io/es/latest/leccion3/>
- [2] - https://librosweb.es/libro/algoritmos_python/capitulo_15/polimorfismo.html
- [3] - https://librosweb.es/libro/algoritmos_python/capitulo_15/herencia.html
- [4] - <http://docs.python.org.ar/tutorial/2/classes.html>