Computação Bioinspirada

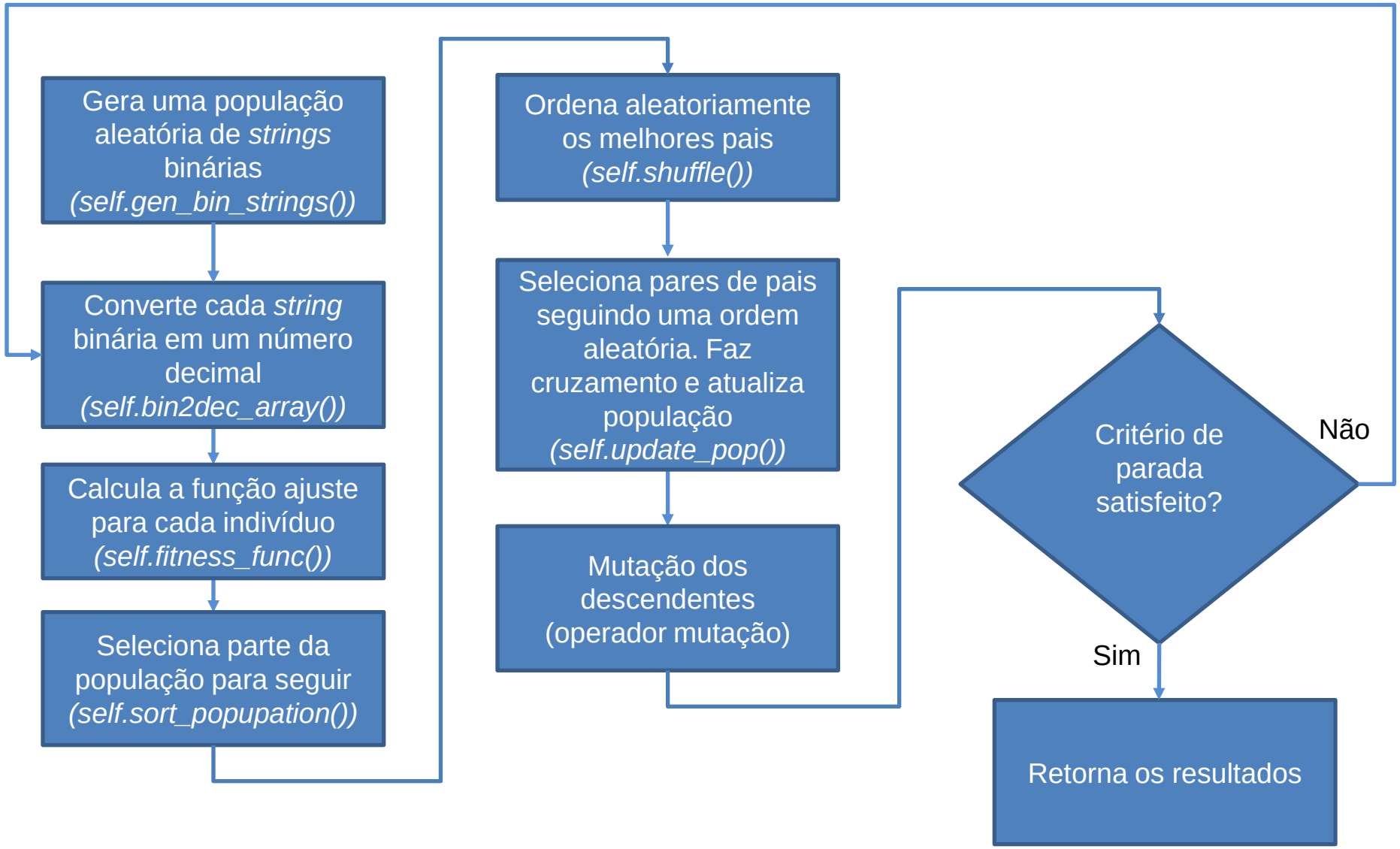
Aula 04

azevedolab.net

Vimos anteriormente um algoritmo genético simples para resolução de um problema de maximização. O algoritmo visa encontrar o maior número entre 0 e 4095 que maximiza a função x^2. No próximo slide temos o fluxograma do programa *sg01.py.*

String binária	Decimal
0 0 0 0 0 0 0 0 0 0 0 0	0
0 0 0 0 0 0 0 0 0 0 0 1	1
0 0 0 0 0 0 0 0 0 0 1 0	2
0 0 0 0 0 0 0 0 0 0 1 1	3
0 0 0 0 0 0 0 0 0 1 0 0	4
0 0 0 0 0 0 0 0 0 1 0 1	5
0 0 0 0 0 0 0 0 0 1 1 0	6
0 0 0 0 0 0 0 0 0 1 1 1	7
0 0 0 0 0 0 0 0 1 0 0 0	8
.....	.....
1 1 1 1 1 1 1 1 1 1 1 0	4094
1 1 1 1 1 1 1 1 1 1 1 1	4095

python™
www.python.org



Veremos hoje a implementação em Python do algoritmo 1 relatado por Coley 1999. Usaremos o paradigma de programação orientada a objeto. Abaixo temos a função *main()*. Os atributos são previamente definidos e indicados em vermelho abaixo. Após a definição dos parâmetros, criamos um objeto da classe *Simple_GA01()* com a linha *p1 = Simple_GA01(pop_size,max_iter,p_cross,len_str,p_mut,2).*

```
def main():
        # Set up initial values for GA
        pop_size = 8 # Population size
        max_iter = 60 # Number of iterations
        p_cross = 0.8 # Probability of cross-over
        len_str = 12 # Length of strings
        p_mut = 0.02 # Probability of mutation_Coley_algo_01

        # Instantiating an object of the Simple_GA01() class and assigns it to p1
        p1 = Simple_GA01(pop_size,max_iter,p_cross,len_str,p_mut,2)
        # Invoking the plot_ga_evolution() method
        p1.coley_algo_01()
        # Invoking the coley_algo_01() method
        p1.plot_ga_evolution("Generation","Fitness function, f","Maximum value of
f","Mean value of f","ga01.png")
main()
```

Após a criação do objeto, o método com a implementação do algoritmo 1 de Coley é invocado com a linha de código *p1.coley_algo_01()*. Por último, invocamos o método *plot_ga_evolution()* para gerarmos o gráfico com a evolução da média e do valor máximo da função ajuste para cada geração.

```
def main():
        # Set up initial values for GA
        pop_size = 8 # Population size
        max_iter = 60 # Number of iterations
        p_cross = 0.8 # Probability of cross-over
        len_str = 12 # Length of strings
        p_mut = 0.02 # Probability of mutation_Coley_algo_01

        # Instantiating an object of the Simple_GA01() class and assigns it to p1
        p1 = Simple_GA01(pop_size,max_iter,p_cross,len_str,p_mut,2)
        # Invoking the plot_ga_evolution() method
        p1.coley_algo_01()
        # Invoking the coley_algo_01() method
        p1.plot_ga_evolution("Generation","Fitness function, f","Maximum value of
f","Mean value of f","ga01.png")
main()
```

Abaixo temos o código inicial da classe *Simple_GA01()*, nele vemos o método construtor onde definimos os atributos para tamanho da população (*self.pop_size*), número máximo de iterações (*self.max_iter*), probabilidade de cross-over (*self.p_cross*), tamanho de cada string binária (*self.len_str*), probabilidade de mutação (*self.p_mut*) e divisor que indica que parte da população é selecionada (*self.div_in*). No último atributo (*self.div_in*) o valor 2 indica que pegamos a metade da população. Também definimos como atributos *arrays* para os valores máximo (*self.max_array*) e médio (*self.mean_array*) da função ajuste em cada geração.

```
class Simple_GA01(object):
        """A class to implement a very simple GA"""
        # Constructor method
        def __init__(self,pop_size,max_iter,p_cross,len_str,p_mut,div_in):
                # Import library
                import numpy as np
                # Define atributes
                self.pop_size = pop_size # Population size
                self.max_iter = max_iter # Number of iterations
                self.p_cross = p_cross # Probability of crossover
                self.len_str = len_str # String length
                self.p_mut = p_mut # Probability of mutation_Coley_algo_01
                self.div_in = div_in
                # Set up arrays of zeros
                self.max_array = np.zeros(self.max_iter)
                self.mean_array = np.zeros(self.max_iter)
```

Em seguida temos o método *__str__()* que define uma string com a descrição da classe.

```
        # __str__ method
        def __str__(self):
                info = "Simple_GA01 class\n"
                info += "Class to generate a model based on a very simple genetic
algorithm.\n"""
                info += "It is a OOP implementation of algorithm 1 described by Coley,
1999.\n"
                info += "Reference:\n"
                info += " Coley, David A. An Introduction to Genetic Algorithms for
Scientists and Engineers.\n"
                info += "Singapore: World Scientific Publishing Co. Pte. Ltd., 1999. 227 pp."
                return info
```

O principal método da classe *Simple_GA01()* é o *coley_algo_01(self)*, mostrado abaixo.

```
        # Definition of coley_algo_01 method
        def coley_algo_01(self):
                """Method to implement algorithm 1 from Coley 1999"""
                # Call gen_bin_strings()
                self.gen_bin_strings()
                # Call show_pop()
                self.show_pop("\nInitial Population",self.current_pop)
                # Looping through the number of iteractions
                for i in range(self.max_iter):
                        # Call selection_Coley_algo_01()
                        self.selection_Coley_algo_01()
                        # Call cross_over_Coley_algo_01() method
                        self.cross_over_Coley_algo_01()
                        # Call mutation_Coley_algo_01 method
                        self.mutation_Coley_algo_01()
                        # Call show_pop()
                        self.show_pop("\nPopulation for generation
"+str(i),self.current_pop)
                        # Call bin2dec_array()
                        d = self.bin2dec_array(self.current_pop)
                        # Call fitness_func()
                        d_2 = self.fitness_func(d)
                        # Call get_max()
                        self.max_array[i] = self.get_max(d_2)
                        # Call get_mean()
                        self.mean_array[i] = self.get_mean(d_2)
                # Call show_pop()
                self.show_pop("\nFinal Population",self.current_pop)
```

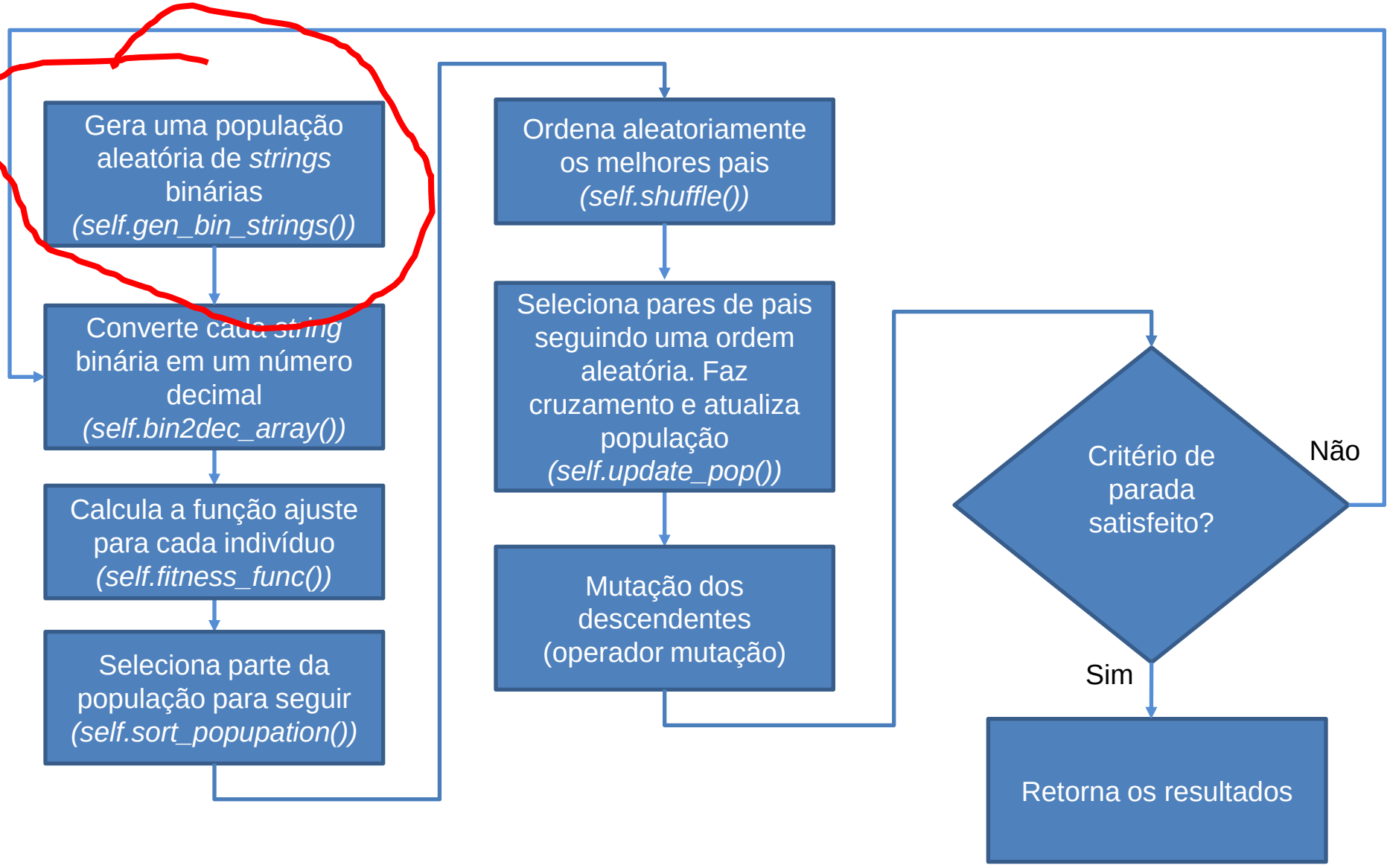
Inicialmente geramos uma população aleatória de strings binárias com o método *self.gen_bin_strings()*. Usamos o *self.* para invocarmos um método na classe.

```
        # Definition of coley_algo_01 method
        def coley_algo_01(self):
                """Method to implement algorithm 1 from Coley 1999"""
                # Call gen_bin_strings()
                self.gen_bin_strings()
                # Call show_pop()
                self.show_pop("\nInitial Population",self.current_pop)
                # Looping through the number of iteractions
                for i in range(self.max_iter):
                        # Call selection_Coley_algo_01()
                        self.selection_Coley_algo_01()
                        # Call cross_over_Coley_algo_01() method
                        self.cross_over_Coley_algo_01()
                        # Call mutation_Coley_algo_01 method
                        self.mutation_Coley_algo_01()
                        # Call show_pop()
                        self.show_pop("\nPopulation for generation
"+str(i),self.current_pop)
                        # Call bin2dec_array()
                        d = self.bin2dec_array(self.current_pop)
                        # Call fitness_func()
                        d_2 = self.fitness_func(d)
                        # Call get_max()
                        self.max_array[i] = self.get_max(d_2)
                        # Call get_mean()
                        self.mean_array[i] = self.get_mean(d_2)
                # Call show_pop()
                self.show_pop("\nFinal Population",self.current_pop)
```

No próximo slide temos o fluxograma com destaque para o método *self.gen_bin_strings().*

```
        # Definition of coley_algo_01 method
        def coley_algo_01(self):
                """Method to implement algorithm 1 from Coley 1999"""
                # Call gen_bin_strings()
                self.gen_bin_strings()
                # Call show_pop()
                self.show_pop("\nInitial Population",self.current_pop)
                # Looping through the number of iteractions
                for i in range(self.max_iter):
                        # Call selection_Coley_algo_01()
                        self.selection_Coley_algo_01()
                        # Call cross_over_Coley_algo_01() method
                        self.cross_over_Coley_algo_01()
                        # Call mutation_Coley_algo_01 method
                        self.mutation_Coley_algo_01()
                        # Call show_pop()
                        self.show_pop("\nPopulation for generation
"+str(i),self.current_pop)
                        # Call bin2dec_array()
                        d = self.bin2dec_array(self.current_pop)
                        # Call fitness_func()
                        d_2 = self.fitness_func(d)
                        # Call get_max()
                        self.max_array[i] = self.get_max(d_2)
                        # Call get_mean()
                        self.mean_array[i] = self.get_mean(d_2)
                # Call show_pop()
                self.show_pop("\nFinal Population",self.current_pop)
```

python™
www.python.org



Depois mostramos a população na tela com o método *self.show_pop()*.

```
        # Definition of coley_algo_01 method
        def coley_algo_01(self):
                """Method to implement algorithm 1 from Coley 1999"""
                # Call gen_bin_strings()
                self.gen_bin_strings()
                # Call show_pop()
                self.show_pop("\nInitial Population",self.current_pop)
                # Looping through the number of iteractions
                for i in range(self.max_iter):
                        # Call selection_Coley_algo_01()
                        self.selection_Coley_algo_01()
                        # Call cross_over_Coley_algo_01() method
                        self.cross_over_Coley_algo_01()
                        # Call mutation_Coley_algo_01 method
                        self.mutation_Coley_algo_01()
                        # Call show_pop()
                        self.show_pop("\nPopulation for generation
"+str(i),self.current_pop)
                        # Call bin2dec_array()
                        d = self.bin2dec_array(self.current_pop)
                        # Call fitness_func()
                        d_2 = self.fitness_func(d)
                        # Call get_max()
                        self.max_array[i] = self.get_max(d_2)
                        # Call get_mean()
                        self.mean_array[i] = self.get_mean(d_2)
                # Call show_pop()
                self.show_pop("\nFinal Population",self.current_pop)
```

Em seguida temos o loop *for* que gera a sequência de operadores seleção, cross-over e mutação. O loop se repete o número de iterações definido no atributo *self.max_iter*.

```
        # Definition of coley_algo_01 method
        def coley_algo_01(self):
                """Method to implement algorithm 1 from Coley 1999"""
                # Call gen_bin_strings()
                self.gen_bin_strings()
                # Call show_pop()
                self.show_pop("\nInitial Population",self.current_pop)
                # Looping through the number of iteractions
                for i in range(self.max_iter):
                        # Call selection_Coley_algo_01()
                        self.selection_Coley_algo_01()
                        # Call cross_over_Coley_algo_01() method
                        self.cross_over_Coley_algo_01()
                        # Call mutation_Coley_algo_01 method
                        self.mutation_Coley_algo_01()
                        # Call show_pop()
                        self.show_pop("\nPopulation for generation
"+str(i),self.current_pop)
                        # Call bin2dec_array()
                        d = self.bin2dec_array(self.current_pop)
                        # Call fitness_func()
                        d_2 = self.fitness_func(d)
                        # Call get_max()
                        self.max_array[i] = self.get_max(d_2)
                        # Call get_mean()
                        self.mean_array[i] = self.get_mean(d_2)
                # Call show_pop()
                self.show_pop("\nFinal Population",self.current_pop)
```

As populações de cada iteração são mostradas na tela. Por último, mostramos a população final. Veremos a seguir os métodos dos operadores genéticos.

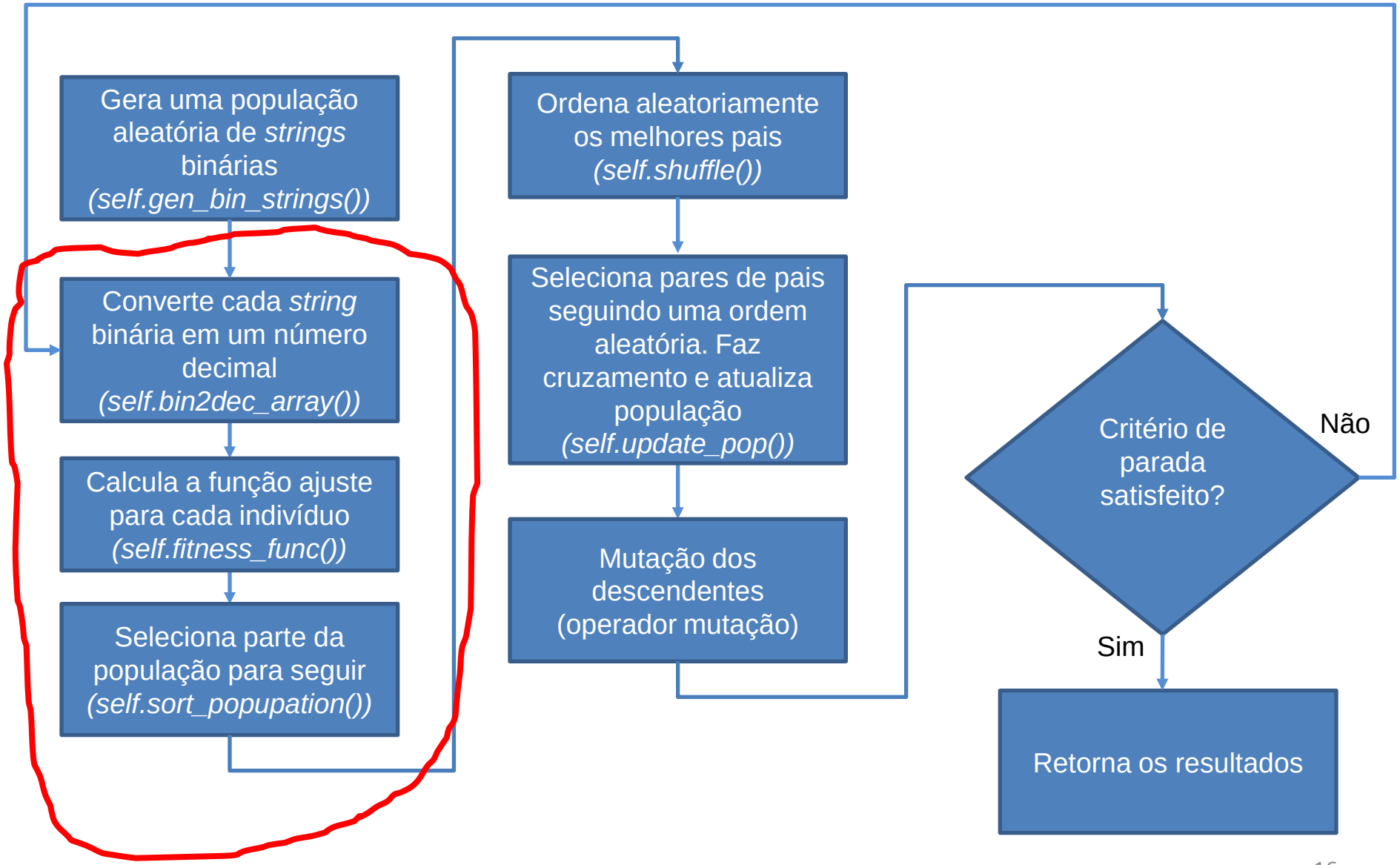
```
        # Definition of coley_algo_01 method
        def coley_algo_01(self):
                """Method to implement algorithm 1 from Coley 1999"""
                # Call gen_bin_strings()
                self.gen_bin_strings()
                # Call show_pop()
                self.show_pop("\nInitial Population",self.current_pop)
                # Looping through the number of iteractions
                for i in range(self.max_iter):
                        # Call selection_Coley_algo_01()
                        self.selection_Coley_algo_01()
                        # Call cross_over_Coley_algo_01() method
                        self.cross_over_Coley_algo_01()
                        # Call mutation_Coley_algo_01 method
                        self.mutation_Coley_algo_01()
                        # Call show_pop()
                        self.show_pop("\nPopulation for generation
"+str(i),self.current_pop)
                        # Call bin2dec_array()
                        d = self.bin2dec_array(self.current_pop)
                        # Call fitness_func()
                        d_2 = self.fitness_func(d)
                        # Call get_max()
                        self.max_array[i] = self.get_max(d_2)
                        # Call get_mean()
                        self.mean_array[i] = self.get_mean(d_2)
                # Call show_pop()
                self.show_pop("\nFinal Population",self.current_pop)
```

O método *selection_Coley_algo_01()* inicialmente converte para decimal cada elemento da lista de strings binárias do atributo *self.current_pop*. Em seguida, é calculada a função ajuste para cada decimal com o método *self.fitness_func().* Depois, o método *self.sort_popupation(self.current_pop)* classifica em ordem decrescente de valores da função ajuste os elementos da população. Assim, o primeiro elemento da lista é aquele de maior valor da função ajuste. Na parte seguinte, varremos a metade da população (*self.div_in = 2),* que é atribuída à variável *new_pop*. Por último, atribuímos a nova população (*new_pop*) ao atributo *self.current_pop.* No slide seguinte temos o fluxograma do algoritmo 1 com destaque para o operador seleção.

```
# Definition of selection_Coley_algo_01() (selection operator)
    def selection_Coley_algo_01(self):
        """Method for selection operator using aklgorithm 1 from Coley 1999"""
        # Call bin2dec_array()
        self.x = self.bin2dec_array(self.current_pop)
        # Call fitness_func()
        self.fit = self.fitness_func(self.x)
        # Sort population
        self.current_pop = self.sort_popupation(self.current_pop)
        # Set up empty list for temporary population
        new_pop = []
        # Looping throguh population (get half of the population if self.div_in = 2)
        for i in range( int( self.pop_size/self.div_in)  ):
            new_pop.append(self.current_pop[i])

        # Update self.current_pop with best indivudual
        self.current_pop = new_pop
```

python™
www.python.org

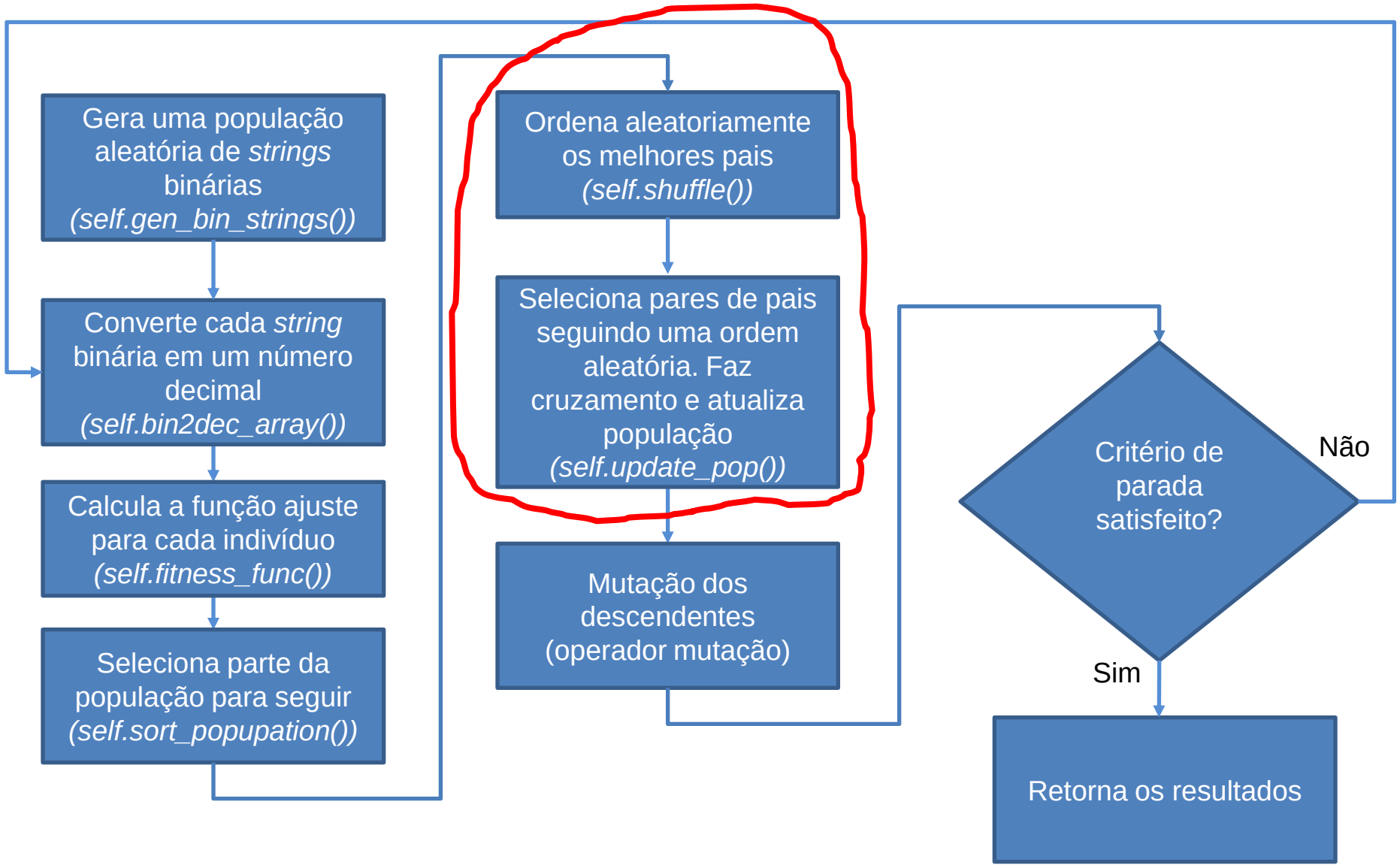


No método *cross_over_Coley_algo_01()* temos a implementação do operador cross-over como proposto no algoritmo 1 de Coley, 1999. Inicialmente a lista com a população corrente é embaralhada com o método *self.shuffle()*. O último método a ser invocado é o *self.update_pop(),* que pega pares aleatórios de strings pais e gera um par de strings filhas para cada par de pais. É usada a probabilidade de cross-over. Assim, nem todos os pares de pais geram pares de filhas, para probabilidades de cross-over menores que 1. No slide seguinte temos o fluxograma do algoritmo 1 com destaque para o operador cross-over.

```
# Definition of cross_over
def cross_over_Coley_algo_01(self):
    """Method to apply cross-over operator to a population"""

    # Call shuffle()
    self.current_pop = self.shuffle(self.current_pop)

    # Call update_pop
    self.update_pop(self.current_pop)
```


Gera uma população aleatória de *strings* binárias *(self.gen_bin_strings())*
Converte cada *string* binária em um número decimal *(self.bin2dec_array())*
Calcula a função ajuste para cada indivíduo *(self.fitness_func())*
Seleciona parte da população para seguir *(self.sort_popupation())*
Ordena aleatoriamente os melhores pais *(self.shuffle())*
Seleciona pares de pais seguindo uma ordem aleatória. Faz cruzamento e atualiza população *(self.update_pop())*
Mutação dos descendentes (operador mutação)
Critério de parada satisfeito?
Não
Sim
Retorna os resultados

No método *mutation_Coley_algo_01()* implementamos o operador mutação. Inicialmente transformamos todas as strings numa única string.

```
# Definition of mutation_Coley_algo_01 method
def mutation_Coley_algo_01(self):
        """Method to apply mutation_Coley_algo_01 operator"""
        # Import library
        import random
        # Determine total
        total = self.pop_size*self.len_str
        # Some editing list-> string
        str_pop = "".join(self.current_pop)
        str_pop = str_pop.replace("\n","")
        # Set up empty string
        str_pop_out = ""
        # Carry out mutation
        for i in range(total):
                if i == int(1/self.p_mut):
                        if str_pop[i] == 1:
                                str_pop_out+="0"
                        else:
                                str_pop_out+="1"
                else:
                        str_pop_out+=str_pop[i]
        # Back to string. Set up empty list
        pop = []
        # Loop to recover population after mutation_Coley_algo_01
        for i in range(self.pop_size):
                aux = ""
                for j in range(self.len_str):
                        aux += str_pop_out[i+j]
                pop.append(aux)
        self.current_pop = pop
```

Em seguida varremos a string única criada e trocamos cada bit na posição determinada a partir da probabilidade de mutação (1/*self.p_mut*).

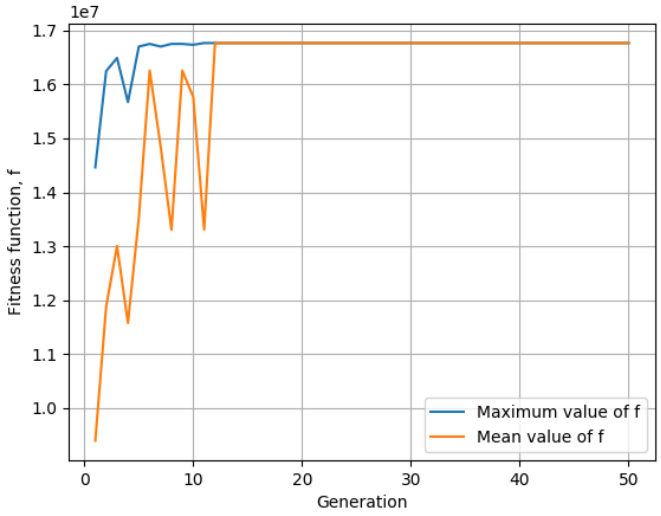
```
# Definition of mutation_Coley_algo_01 method
def mutation_Coley_algo_01(self):
        """Method to apply mutation_Coley_algo_01 operator"""
        # Import library
        import random
        # Determine total
        total = self.pop_size*self.len_str
        # Some editing list-> string
        str_pop = "".join(self.current_pop)
        str_pop = str_pop.replace("\n","")
        # Set up empty string
        str_pop_out = ""
        # Carry out mutation
        for i in range(total):
                if i == int(1/self.p_mut):
                        if str_pop[i] == 1:
                                str_pop_out+="0"
                        else:
                                str_pop_out+="1"
                else:
                        str_pop_out+=str_pop[i]
        # Back to string. Set up empty list
        pop = []
        # Loop to recover population after mutation_Coley_algo_01
        for i in range(self.pop_size):
                aux = ""
                for j in range(self.len_str):
                        aux += str_pop_out[i+j]
                pop.append(aux)
        self.current_pop = pop
```

No loop seguinte recuperamos as strings binárias. O operador mutação é o último operador genético do loop dor método *coley_algo_01().*

```
# Definition of mutation_Coley_algo_01 method
def mutation_Coley_algo_01(self):
        """Method to apply mutation_Coley_algo_01 operator"""
        # Import library
        import random
        # Determine total
        total = self.pop_size*self.len_str
        # Some editing list-> string
        str_pop = "".join(self.current_pop)
        str_pop = str_pop.replace("\n","")
        # Set up empty string
        str_pop_out = ""
        # Carry out mutation
        for i in range(total):
                if i == int(1/self.p_mut):
                        if str_pop[i] == 1:
                                str_pop_out+="0"
                        else:
                                str_pop_out+="1"
                else:
                        str_pop_out+=str_pop[i]
        # Back to string. Set up empty list
        pop = []
        # Loop to recover population after mutation_Coley_algo_01
        for i in range(self.pop_size):
                aux = ""
                for j in range(self.len_str):
                        aux += str_pop_out[i+j]
                pop.append(aux)
        self.current_pop = pop
```

Abaixo temos resultado da execução do código *sga01.py*.

```
Final population
111111111111
111111111111
111111111111
111111111111
111111111111
111111111111
111111111111
111111111111
```



Modifique os parâmetros para os valores mostrados abaixo.

```
# Set up initial values for GA
pop_size = 8 # Population size
max_iter = 50 # Number of iterations
p_cross = 0.8 # Probability of cross-over
len_str = 12 # Length of strings
p_mut = 0.17 # Probability of mutation_Coley_algo_01
```

1) No método *coley_algo_01()* modifique o código de forma que não seja aplicado o operador cross-over. Dica: coloque o símbolo de comentário (#) no início da linha:

```
#self.cross_over_Coley_algo_01().
```

2) Repita o processo para o operador mutação.

3) Repita o processo para os dois operadores.

4) O programa acha a reposta certa?

O programa *tartaruga01.py* invoca a classe *touche()* para desenhar uma tartaruga. Este programa desenha uma tartaruga a partir de parâmetros definidos.

```
# Program to import touch class and draw turtles
# Import libraries
import touche as tart
import hexadec as hx
import numpy as np
# Instantiating an object of the Hexa() class and assigns it to the variable h1
h1 = hx.Hexa() # String for hexadecimal
# Invoking the gen_hex() method
h1.gen_hex()
hex = h1.my_hex_string # Assign hexadecimal string to hex
# Define parameters for a turtle
x = 0 # x coordinate in pixels
y = 0 # y coordinate in pixels
length = 3.2 # Length for the turtle
width = 3.2 # Width for the turtle
rotate = np.random.randint(0,8) # Rotation angle 8*45 degrees
# Instantiating an object of the Touche() class and assigns it to the variable t1
t1 = tart.Touche(x, y, length, width, rotate, "#"+hex)
# Invoking the draw() method
t1.draw()
# Invoking the bye method
t1.bye()
```

Inicialmente o programa importa bibliotecas. Além da nossa conhecida *NumPy*, o programa importa duas bibliotecas. A *touche* é usada para desenhar uma tartaruga. A *hexadec* gera uma string com um número hexadecimal.

```
# Program to import touch class and draw turtles
# Import libraries
import touche as tart
import hexadec as hx
import numpy as np
# Instantiating an object of the Hexa() class and assigns it to the variable h1
h1 = hx.Hexa() # String for hexadecimal
# Invoking the gen_hex() method
h1.gen_hex()
hex = h1.my_hex_string # Assign hexadecimal string to hex
# Define parameters for a turtle
x = 0 # x coordinate in pixels
y = 0 # y coordinate in pixels
length = 3.2 # Length for the turtle
width = 3.2 # Width for the turtle
rotate = np.random.randint(0,8) # Rotation angle 8*45 degrees
# Instantiating an object of the Touche() class and assigns it to the variable t1
t1 = tart.Touche(x, y, length, width, rotate, "#"+hex)
# Invoking the draw() method
t1.draw()
# Invoking the bye method
t1.bye()
```

Os códigos das bibliotecas t*ouche* e *hexadec* devem estar no mesmo diretório do código *tartaruga01.py*. Essas bibliotecas são módulos em Python *(touche.py* e *hexadec.py*).

```
# Program to import touch class and draw turtles
# Import libraries
import touche as tart
import hexadec as hx
import numpy as np
# Instantiating an object of the Hexa() class and assigns it to the variable h1
h1 = hx.Hexa() # String for hexadecimal
# Invoking the gen_hex() method
h1.gen_hex()
hex = h1.my_hex_string # Assign hexadecimal string to hex
# Define parameters for a turtle
x = 0 # x coordinate in pixels
y = 0 # y coordinate in pixels
length = 3.2 # Length for the turtle
width = 3.2 # Width for the turtle
rotate = np.random.randint(0,8) # Rotation angle 8*45 degrees
# Instantiating an object of the Touche() class and assigns it to the variable t1
t1 = tart.Touche(x, y, length, width, rotate, "#"+hex)
# Invoking the draw() method
t1.draw()
# Invoking the bye method
t1.bye()
```

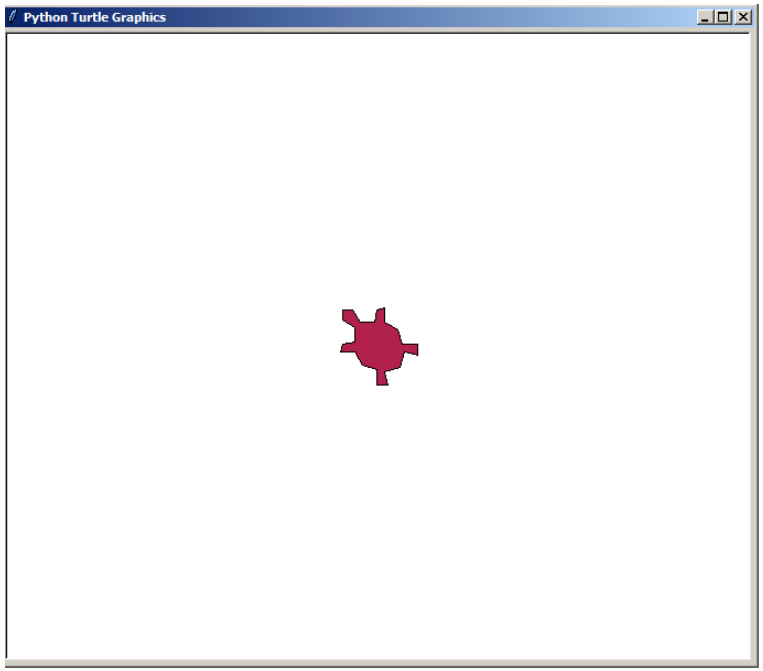
Agora criamos um objeto da classe *Hexa().* Em seguida invocamos o método *gen_hex().* Para termos a string, acessamos o atributo do objeto com a linha *hex = h1.my_hex_string.*

```
# Program to import touch class and draw turtles
# Import libraries
import touche as tart
import hexadec as hx
import numpy as np
# Instantiating an object of the Hexa() class and assigns it to the variable h1
h1 = hx.Hexa() # String for hexadecimal
# Invoking the gen_hex() method
h1.gen_hex()
hex = h1.my_hex_string # Assign hexadecimal string to hex
# Define parameters for a turtle
x = 0 # x coordinate in pixels
y = 0 # y coordinate in pixels
length = 3.2 # Length for the turtle
width = 3.2 # Width for the turtle
rotate = np.random.randint(0,8) # Rotation angle 8*45 degrees
# Instantiating an object of the Touche() class and assigns it to the variable t1
t1 = tart.Touche(x, y, length, width, rotate, "#"+hex)
# Invoking the draw() method
t1.draw()
# Invoking the bye method
t1.bye()
```

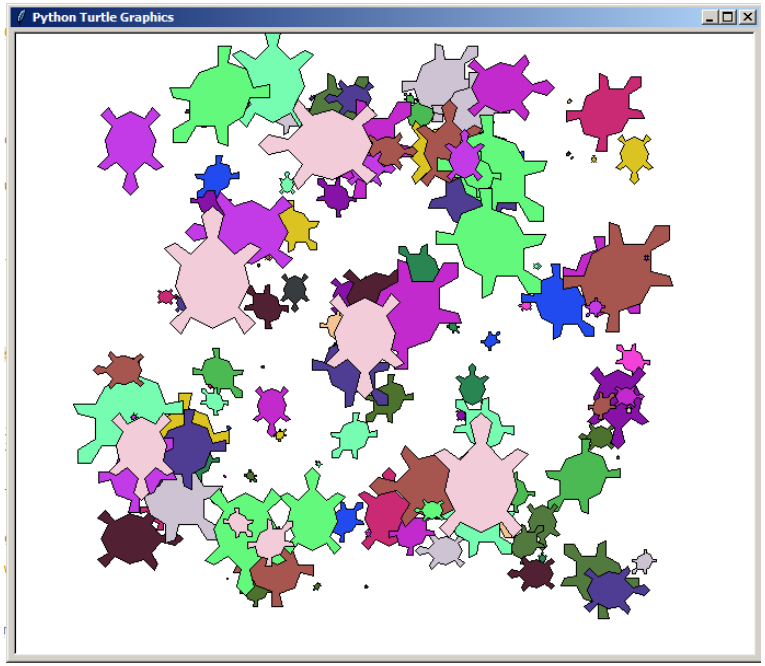
Para desenharmos a tartaruga com o módulo *touche* precisamos definir os parâmetros para o desenho. Depois criamos um objeto da classe *Touche()* e invocamos os métodos *draw()* e *bye()*, como destacado abaixo.

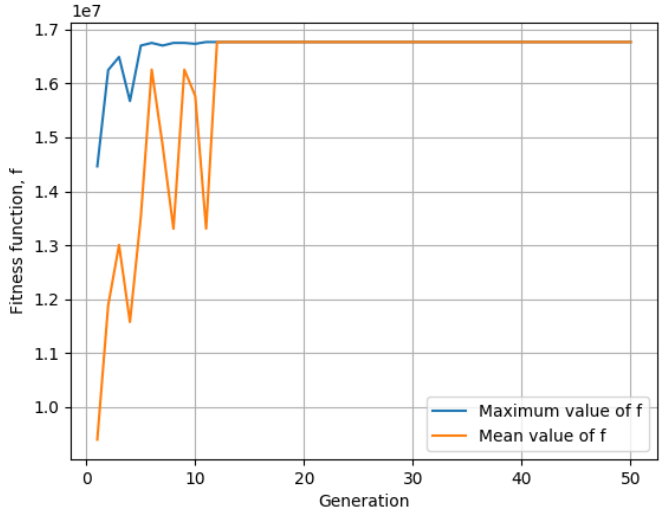
```
# Program to import touch class and draw turtles
# Import libraries
import touche as tart
import hexadec as hx
import numpy as np
# Instantiating an object of the Hexa() class and assigns it to the variable h1
h1 = hx.Hexa() # String for hexadecimal
# Invoking the gen_hex() method
h1.gen_hex()
hex = h1.my_hex_string # Assign hexadecimal string to hex
# Define parameters for a turtle
x = 0 # x coordinate in pixels
y = 0 # y coordinate in pixels
length = 3.2 # Length for the turtle
width = 3.2 # Width for the turtle
rotate = np.random.randint(0,8) # Rotation angle 8*45 degrees
# Instantiating an object of the Touche() class and assigns it to the variable t1
t1 = tart.Touche(x, y, length, width, rotate, "#"+hex)
# Invoking the draw() method
t1.draw()
# Invoking the bye method
t1.bye()
```

Abaixo temos resultado da execução do código *tartaruga01.py*.



Usaremos os módulos *touche* e *hexadec* para modificarmos o programa *sga01.py,* de forma que a evolução da função ajuste seja expressa pelo crescimento das tartarugas em cada iteração do algoritmo genético. O programa usa os atributos de largura (*width*) e comprimento (*length*) da classe *Touche()* para expressar o tamanho que será proporcional ao valor da função ajuste de cada indivíduo da população. O programa mostra os indivíduos da população em cada geração. A posição e as cores serão aleatórias, só o tamanho será proporcional ao valor da função ajuste. O novo programa será chamado *sga02.py.*





As modificações no código ficam restritas ao método *coley_algo_01()*. Como o código do referido método é extenso, mostraremos as modificações por etapas. Inicialmente importamos os módulo *touche* e *hexadec*. A biblioteca *NumPy* também é importada, ela será usada para geração de números aleatórios.

```
    # Definition of coley_algo_01 method
    def coley_algo_01(self):
        """Method to implement algorithm 1 from Coley 1999"""
         # Import libraries
        import numpy as np
        import touche as tart
        import hexadec as hx

        # General parameters for the turtles
        x_min = -240          # Minimum x coordinate in pixels
        x_max =  240          # Maximum x coordinate in pixels
        y_min = -240          # Minimum y coordinate in pixels
        y_max =  240          # Maximum y coordinate in pixels
        max_rotate = 8        # Maximum rotation angle 8*45 degrees
        conv_f = 3e-7         # Conversion factor (fitness function to dimensions)

        # Call gen_bin_strings()
        self.gen_bin_strings()

        # Call show_pop()
        self.show_pop("\nInitial Population",self.current_pop)
```

Em seguida definimos parâmetros gerais que serão usados para desenhar as tartarugas. Temos quatro parâmetros que definem a faixa onde serão desenhadas as tartarugas (*x_min*, *x_max*, *y_min*, e *y_maxl).* Os valores estão em pixels. O parâmetro *max_rotate* define o número máximo de rotações em múltiplos de 45°.

```
    # Definition of coley_algo_01 method
    def coley_algo_01(self):
        """Method to implement algorithm 1 from Coley 1999"""
         # Import libraries
        import numpy as np
        import touche as tart
        import hexadec as hx

        # General parameters for the turtles
        x_min = -240          # Minimum x coordinate in pixels
        x_max =  240          # Maximum x coordinate in pixels
        y_min = -240          # Minimum y coordinate in pixels
        y_max =  240          # Maximum y coordinate in pixels
        max_rotate = 8        # Maximum rotation angle 8*45 degrees
        conv_f = 3e-7         # Conversion factor (fitness function to dimensions)

        # Call gen_bin_strings()
        self.gen_bin_strings()

        # Call show_pop()
        self.show_pop("\nInitial Population",self.current_pop)
```

O fator de conversão do valor da função ajuste para as dimensões das tartarugas foi atribuído à variável *conv_f*. Este valor é razoável para o desenho das tartarugas, sabendo-se que o máximo valor da função ajuste é 16769025.

```
    # Definition of coley_algo_01 method
    def coley_algo_01(self):
        """Method to implement algorithm 1 from Coley 1999"""
         # Import libraries
        import numpy as np
        import touche as tart
        import hexadec as hx

        # General parameters for the turtles
        x_min = -240          # Minimum x coordinate in pixels
        x_max =  240          # Maximum x coordinate in pixels
        y_min = -240          # Minimum y coordinate in pixels
        y_max =  240          # Maximum y coordinate in pixels
        max_rotate = 8        # Maximum rotation angle 8*45 degrees
        conv_f = 3e-7         # Conversion factor (fitness function to dimensions)

        # Call gen_bin_strings()
        self.gen_bin_strings()

        # Call show_pop()
        self.show_pop("\nInitial Population",self.current_pop)
```

Em seguida criamos um objeto da classe *Hexa()* e invocamos o método *gen_hex().* Acessamos o atributo *my_hex_string* e o atribuímos à variável *hex*. Esta string hexadecimal será usada para definir a cor das tartarugas de uma dada geração.

```
# Looping through the number of iteractions
   for i in range(self.max_iter):
        # Call selection_Coley_algo_01()
        self.selection_Coley_algo_01()
        # Call cross_over_Coley_algo_01() method
        self.cross_over_Coley_algo_01()
        # Call mutation_Coley_algo_01 method
        self.mutation_Coley_algo_01()
        # Call show_pop()
        self.show_pop("\nPopulation for generation "+str(i),self.current_pop)
        # Call bin2dec_array()
        d = self.bin2dec_array(self.current_pop)
        # Call fitness_func()
        d_2 = self.fitness_func(d)
        # Instantiating an object of the Hexa() class and assigns it to the h1
        h1 = hx.Hexa()                                  # String for hexadecimal
        # Invoking the method gen_hex()
        h1.gen_hex()
        hex = h1.my_hex_string
```

Dentro to loop *for* das iterações temos um loop *for* para varrer os indivíduos da população de uma dada geração, em vermelho abaixo. As coordenadas *x,y* de cada tartaruga a ser desenhada são definidas por números aleatórios dentro das faixas previamente estabelecidas.

```
        # Looping through all elements
        for j in range(len(d_2)):
            # Define parameters for each turtle
            x = np.random.randint(x_min,x_max)          # x coordinate in pixels
            y = np.random.randint(y_min,y_max)          # y coordinate in pixels
            length  = d_2[j]*conv_f                     # Length for the turtle
            width = d_2[j]*conv_f                       # Width for the turtle
            rotate = np.random.randint(0,max_rotate)    # Rotation angle 8*45 def
            # Instantiating an object of the Touche() class and assigns it to t
            t = tart.Touche(x, y, length, width, rotate, "#"+hex)
            # Invoking the method draw()
            t.draw()
        # Call get_max()
        self.max_array[i] = self.get_max(d_2)
        # Call get_mean()
        self.mean_array[i] = self.get_mean(d_2)
# Invoking the method bye()
t.bye()
# Call show_pop()
self.show_pop("\nFinal Population",self.current_pop)
```

As dimensões das tartarugas (*length* e *width*) são obtidas a partir da multiplicação do valor da função ajuste de cada indivíduo (*d_2[j]*) pelo fator de conversão (*conv_f*).

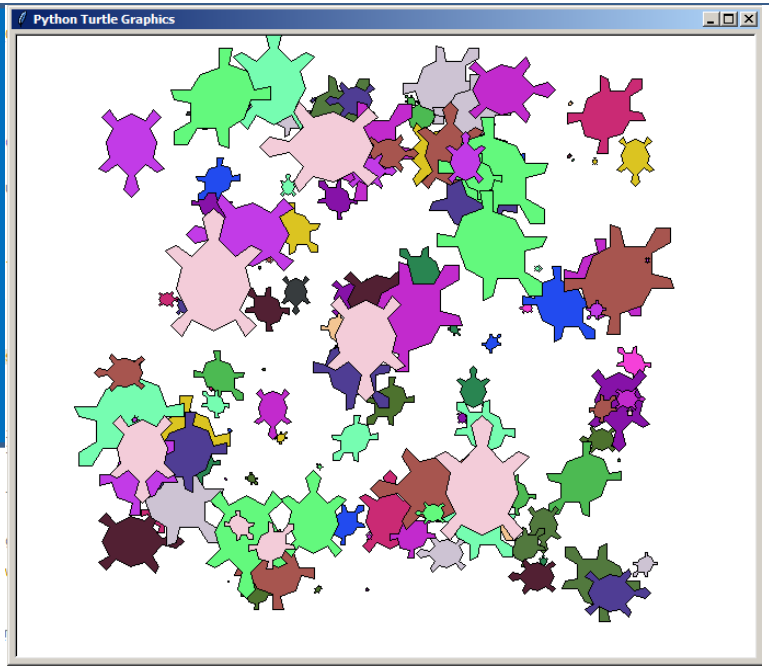
```
        # Looping through all elements
        for j in range(len(d_2)):
            # Define parameters for each turtle
            x = np.random.randint(x_min,x_max)          # x coordinate in pixels
            y = np.random.randint(y_min,y_max)          # y coordinate in pixels
            length  = d_2[j]*conv_f                     # Length for the turtle
            width = d_2[j]*conv_f                       # Width for the turtle
            rotate = np.random.randint(0,max_rotate)    # Rotation angle 8*45 def
            # Instantiating an object of the Touche() class and assigns it to t
            t = tart.Touche(x, y, length, width, rotate, "#"+hex)
            # Invoking the method draw()
            t.draw()
        # Call get_max()
        self.max_array[i] = self.get_max(d_2)
        # Call get_mean()
        self.mean_array[i] = self.get_mean(d_2)
    # Invoking the method bye()
    t.bye()
    # Call show_pop()
    self.show_pop("\nFinal Population",self.current_pop)
```

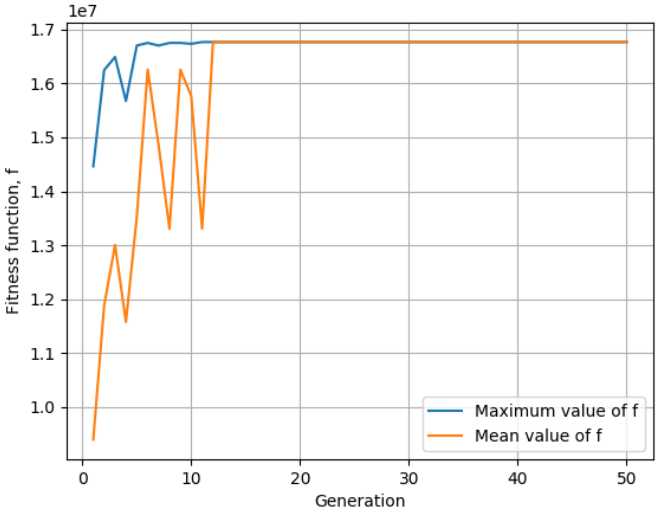
Em seguida, criamos um objeto da classe *Touche()*, usando-se como atributos do objeto os parâmetros previamente definidos. Depois invocamos o método *draw()*. Por último, fora dos loops *for*, invocamos o método *bye()*. O restante do código é idêntico ao do programa *sga01.py*.

```
        # Looping through all elements
        for j in range(len(d_2)):
            # Define parameters for each turtle
            x = np.random.randint(x_min,x_max)          # x coordinate in pixels
            y = np.random.randint(y_min,y_max)          # y coordinate in pixels
            length  = d_2[j]*conv_f                     # Length for the turtle
            width = d_2[j]*conv_f                       # Width for the turtle
            rotate = np.random.randint(0,max_rotate)    # Rotation angle 8*45 def
            # Instantiating an object of the Touche() class and assigns it to t
            t = tart.Touche(x, y, length, width, rotate, "#"+hex)
            # Invoking the method draw()
            t.draw()
        # Call get_max()
        self.max_array[i] = self.get_max(d_2)
        # Call get_mean()
        self.mean_array[i] = self.get_mean(d_2)
# Invoking the method bye()
t.bye()
# Call show_pop()
self.show_pop("\nFinal Population",self.current_pop)
```

Abaixo temos resultado da execução do código *sga02.py*.

```
Final population
11111111111
11111111111
11111111111
11111111111
11111111111
11111111111
11111111111
11111111111
```





A seguir temos a descrição de dois programas a serem desenvolvidos, usando-se o conteúdo discutido na presente aula.
Lista de programas:

tartaruga02.py
sga03.py

Desenho de Tartarugas de Cores, Dimensões e Posições Aleatórias

Programa: *tartaruga02.py*

Resumo
Modifique o código do programa *tartaruga01.py,* de forma que sejam geradas tartarugas com posições, cores e dimensões aleatórias.

Algoritmo Genético Simples (Versão 3)

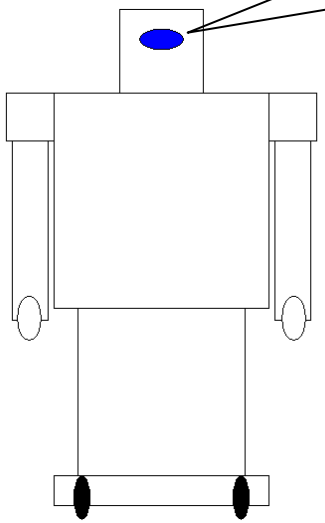
Programa: *sga03.py*

Resumo
Modifique o código do programa *sga02.py,* de forma que o novo programa ache o valor mínimo da função x^2.

A seguir temos a lista de programas da presente aula.
Lista de programas:

sga01.py
sga02.py
tartaruga01.py
touche.py
hexadec.py

This text was produced in a HP Pavillon dm4 notebook with 6GB of memory, a 500 GB hard disk, and an Intel® Core® i7 M620 CPU @ 2.67 GHz running Windows 7 Professional. Text and layout were generated using PowerPoint 2007. Turtles drawings on slides 29, 30, and 38 were generated with Turtle library. Plots on slides 22, 30, and 38 were generated by Matplotlib. The image on the first slide was taken from http://briandcolwell.com/2017/02/artificial-intelligence-the-big-list-of-things-you-should-know/.html on March 26th 2017. This text uses Arial font.



-BRESSERT, Eli. **SciPy and NumPy**. Sebastopol: O'Reilly Media, Inc., 2013. 57 p. Link

-DAWSON, Michael. **Python Programming, for the Absolute Beginner.** 3ed. Boston: Course Technology, 2010. 455 p.

-COLEY, David A. **An Introduction to Genetic Algorithms for Scientists and Engineers.** Singapore: World Scientific Publishing Co. Pte. Ltd., 1999. 227 p.

-EIBEN, A. E., SMITH, J. E**. Introduction to Evolutionary Computing. Natural Computing Series**. 2nd ed. Berlin: Springer-Verlag, 2015. 287 p. Link

-HACKELING G. **Mastering Machine Learning with scikit-learn**. Birmingham: Packt Publishing Ltd., 2014. 221 p. Link

-HETLAND, Magnus Lie. **Python Algorithms. Mastering Basic Algorithms in the Python Language.** 2ed. Nova York: Springer Science+Business Media LLC, 2010. 294 p. Link

-IDRIS, Ivan. **NumPy 1.5. An action-packed guide dor the easy-to-use, high performance, Python based free open source NumPy mathematical library using real-world examples. Beginner's Guide.** Birmingham: Packt Publishing Ltd., 2011. 212 p. Link

-LUTZ, Mark. **Programming Python.** 4ed. Sebastopol: O'Reilly Media, Inc., 2010. 1584 p. Link

-TOSI, Sandro. **Matplotlib for Python Developers.** Birmingham: Packt Publishing Ltd., 2009. 293 p. Link

Última atualização: 28 de maio de 2017.