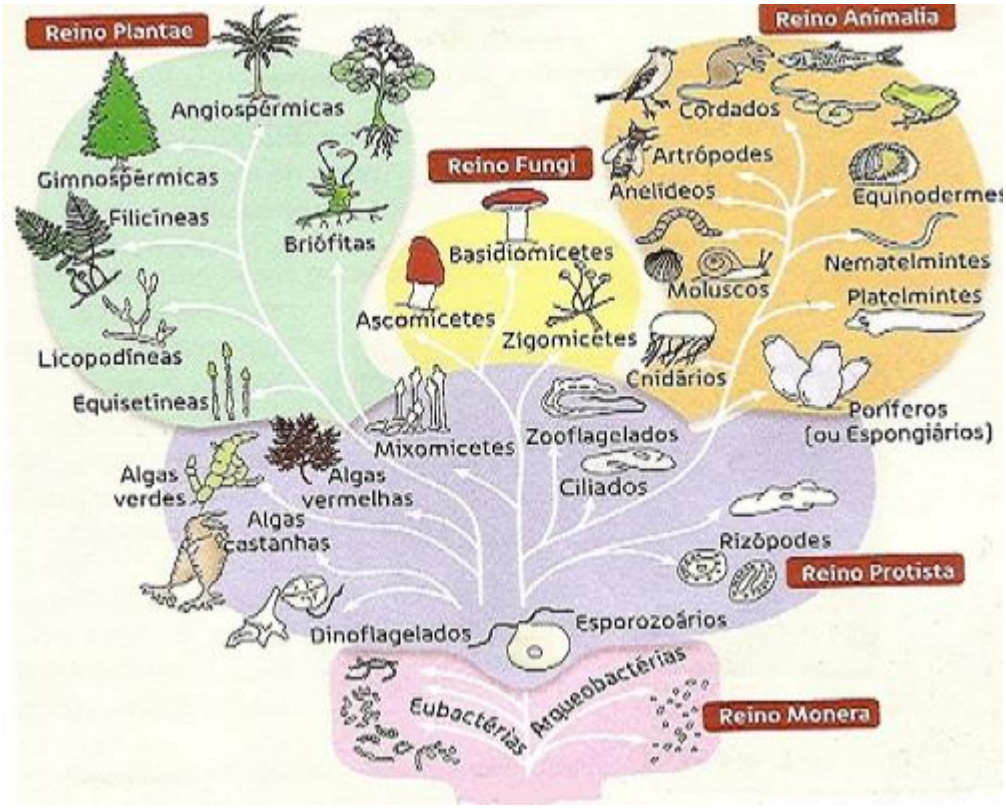
Capítulo 0

Introdução à programação orientada à objetos em Python



Taxonomia: A classificação dos seres vivos, disponível em https://oncomciencias.wordpress.com/2012/05/13/taxonomia-a-classificacao-dos-seres-vivos/ , acessado em 03/04/2014

Unidade 1 - Introdução à programação orientada à objetos em Python

Introdução

A programação em Python é baseada no paradigma de orientação à objetos, isto é, os dados utilizados no Python são considerados objetos. Para a manipulação destes objetos necessitamos das funções e procedimentos que serão chamados de métodos e os tipos de objetos serão as classes.

A Programação Orientada a Objetos possui quatro pilares: Abstração, Encapsulamento, Herança e Polimorfismo.

Abstração

A abstração consiste em um dos pontos mais importantes na POO. Como representar um objeto do mundo real, por exemplo, um número complexo em um programa? O primeiro fator é a identidade do objeto, isto é, dentro do programa um determinado objeto tem que ser único. O segundo fator são os atributos ou características que este objeto possui. Por exemplo, se o número é complexo ele possui a parte real e a parte imaginária. O terceiro fator são as ações que serão executadas com este objeto. No caso do objeto número complexo, podemos ter como ações: módulo, simétrico e conjugado.

Encapsulamento

O *encapsulamento* é uma característica fundamental da POO. O objeto é visto como se fosse uma caixa preta, isto é, o programa que utiliza o objeto não enxerga como é feito as manipulações internas dentro deste objeto. No exemplo anterior, considerando o objeto um número complexo, o programa que utiliza um determinado número complexo, quando se calcula o módulo desse complexo, não enxerga como é calculado o módulo desse complexo.

Herança

A Herança é uma das principais características da POO. Na Física, a impedância elétrica é um número complexo onde a parte real corresponde a resistência do fio (R) e a parte imaginária é a reatância (Z). Nesse caso, podemos observar que a impedância herda todas as propriedades definidas do objeto número complexo. Esta característica é definida como herança.

Polimorfismo

O polimorfismo é a características de objetos que possui uma hierarquia de herança. Por exemplo, o objeto impedância elétrica possui uma característica que sobrepõe a característica de seu ancestral, objeto número complexo. A parte imaginária da impedância elétrica a reatância, pode ser capacitiva ou indutiva. Logo, quando se trabalha com objetos do tipo impedância elétrica, o tratamento da parte imaginária deve ser modificado contemplando essa característica. Ou seja, se um objeto é criado como sendo uma impedância elétrica, ele se **molda** nas características e especificidades do objeto impedância sobrepondo as características do número complexo.

Neste capítulo será apresentado os conceitos de programação orientada à objetos utilizando a linguagem Python como exemplificação dos conceitos.

Tipos de dados Primitivos

Os dados primitivos em Python podem ser do tipo: **boolean** (lógico), **char** (caractere), **String** (cadeia de caracteres), **int** (inteiro) , **float** (real ponto flutuante) e **complex** (número complexo)

- int - para números inteiros
- str - para conjunto de caracteres
- bool - armazena True ou False
- float - para números reais em ponto flutuante
- complex – para números complexos

```
"""
*   Tipos Primitivos de Dados em Python
*
"""
b = 2         # Inteiro
f=3.1415      #  Real ponto flutuante
flag = True   # lógico
c = 'A'       #  Caractere
s  = 'Dados'  #  String
z = 3 - 4j    # Complexo
```

Comentário

Em Python o comentário pode ser de duas formas: a primeira, comentário que possui mais de uma linha, inicia-se com “”” e termina o comentário com “””, e a segunda, comentário de uma linha, inicia-se com #. A seguir descrevem-se estes dois tipos

```
"""
*   Início do comentário com mais de uma linha
*  Corpo do comentário
"
"""
#   fim do comentário
```

Atribuições

Em Python o operador **=** é utilizado para a atribuição. A seguir temos alguns exemplos de atribuições.

```
valor = 32000   # atribui 32000 à variável valor
flag = True     # atribui o valor true à variável do tipo lógico flag
```

Em alguns casos podemos fazer uma atribuição de um valor fixo para várias variáveis. Por exemplo, para atribuirmos o valor 0 (zero) para as variáveis x, y e z, do tipo inteiro poderíamos escrever

```
x = y = z = 0   # atribui o valor 0 às variáveis x,y e z
```

Forma básica de impressão

Em Python o comando básico de impressão é utilizando o método “**print**” para imprimir e, após a impressão, automaticamente salta para a próxima linha.

Na linha de comando a seguir será impresso *flag* = ***True***, considerando que a variável flag tenha valor igual a ***True***.

```
print('flag = ' + str(flag))
```

O operador **+** neste caso funciona como uma concatenação do String ‘flag = ‘ e o string inicializado com o valor da variável flag.

Operadores Aritméticos

Em Python os principais operadores aritméticos são: **+, -, *, /, %** e ***abs*** que significam respectivamente, os operadores: soma, diferença, produto, divisão, resto da divisão e valor absoluto. Observa-se que para obtermos o valor absoluto temos que importar a biblioteca math e o método abs (float abs). A tabela 1.1, ilustra os operadores o seu significado e um exemplo de utilização de cada operador.

Tabela 1.1 – Descrição dos operadores aritméticos, significado, exemplo e os retornos.

OPERADOR	SIGNIFICADO	EXEMPLO	RETORNO
+	Soma	13+4	17
-	Diferença	8-17	-9
*	Produto	3*5	15
/	Divisão	13/57	0,2280
%	Resto da divisão	23%4	3
abs	Valor absoluto	fabs(-14)	14

O fragmento de código 1.2, ilustra a utilização dos operadores aritméticos em Python. O operador **%**, retorna o resto da divisão inteira entre dois números inteiros e o operador **mat.fabs**, retorna o valor absoluto da variável. Também é importante frisar que o operador **/** é utilizado para a divisão de dois números do tipo **float** e retorna um número do tipo **float**.

```
import math
x = 30
y = 6
pi = 3.1416
fi =  1.6180
print ( "x é " + str(x) +", y é " +str( y) )
print( "x + y = " + str((x + y)) )
print ( "x - y = " + str((x - y)) )
print( "x / y = " + str((x / y)) )
print ( "x % y = " + str(( x % y )) )
print( "valor absoluto de -x = " + str(math.fabs(-x)) )
print ( "pi é  " + str(pi) + ", fi é " + str(fi) )
print ( "pi é  " + str(math.pi) + ", fi é " + str(fi) )
```

Fragmento de Código 1.2. Implementação em Python dos principais operadores aritméticos.

Repare que a última linha de impressão, ao imprimir o valor da variável pi, é chamado o valor da constante pi da biblioteca math.

Os valores impressos após a execução do fragmento de código 1.2 estão listados a seguir

```
x é 30, y é 6
x + y = 36
x - y = 24
x / y = 5.0
x % y = 0
valor absoluto de -x = 30.0
pi é  3.1416, fi é 1.618
pi é  3.141592653589793, fi é 1.618
```

Operador pré fixado

Em Python podemos utilizar a ideia de pré-condição. O operador pré-fixado aplicado à uma variável qualquer, é representada da seguinte forma

variável (operador)= expressão

O Operador pode ser +, -, *, e /

A seguir exemplificam-se a utilização destes operadores.

Atribuição pré fixado

Os operadores de atribuição na forma de pré-condição em Python são: **+=,-=, *= ,** e **/=**. A tabela 1.2 ilustra estes operadores, as expressões correspondentes e os respectivos significados.

Tabela 1.2 – Descrição dos operadores aritméticos na forma de pré-condição, seu significado e exemplos de utilização.

OPERADOR	EXPRESSÂO	SIGNIFICADO
+=	x += y	x = x + y
–=	x –= y	x = x – y
*=	x *= y	x = x * y
/=	x /= y	x = x / y

O fragmento de código 1.3, ilustra a utilização dos operadores aritméticos como forma de atribuição em Python.

```
x = 9
y = 4
x += y
print(x)
x -= y
print(x)
x *= y
print(x)
x /= y
print(x)
print(x != y)
```

Fragmento de Código 1.3. Implementação em Python dos operadores pré-fixados na forma de atribuição.

Os valores impressos após a execução do fragmento de código 1.3 serão:

```
13
9
36
9.0
True
```

Comparações

O Python possui várias expressões para testar a igualdade e magnitude. Todas as expressões retornam um valor booleano (***True*** ou ***False***).

Operadores de comparação

Os operadores de comparação em Python são: ==, !=, <, <=, > e >=. A tabela 1.4 ilustra estes operadores, as expressões correspondentes e os respectivos significados.

Tabela 1.4 – Utilização dos operadores de comparação.

OPERADOR	EXPRESSÂO	SIGNIFICADO
==	x == 3	x é igual a 3
!=	x != 3	x é diferente de 3
<	x < 3	x é menor que 3
<=	x < =3	x é menor ou igual a 3
>	x >3	x é maior que 3
>=	x > =3	x é maior ou igual a 3

Operadores lógicos

Os operadores lógicos em Python são: and, or e not A tabela 1.5 ilustra estes operadores, as expressões correspondentes e os respectivos significados.

Tabela 1.5 – Utilização dos operadores lógicos.

OPERADOR	EXPRESSÂO	SIGNIFICADO
and	x>3 and y<7	Operação lógica E (AND)
or	x>3 or y<7	Operação lógica OU (OR)
not	not flag	Negação lógica

Estruturas de Controle

As estruturas de controle podem ser de duas formas, a seleção simples e a seleção múltipla.

Seleção simples

Em Python a estrutura de seleção simples corresponde a condicional se (**if**). A forma básica do uso da condicional se pode ser

```
if < expressão lógica > :
   # Corpo do então

else:
   # Corpo do senão
```

A figura 1.1 ilustra o fluxograma do comando **if** com os respectivos desvios dependendo se a condição é verdadeira ou falsa.

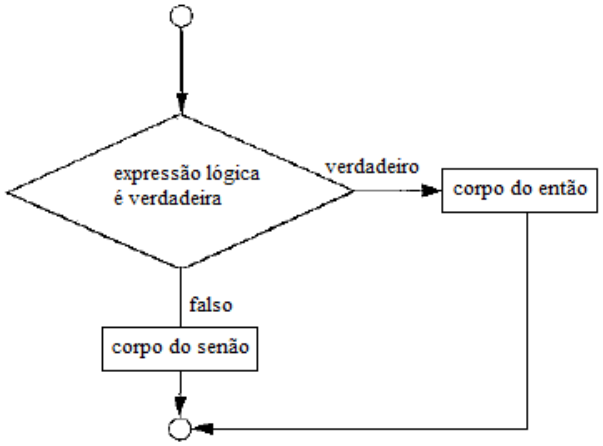


Figura 1.1 – Fluxograma do comando if.

O fragmento de código 1.5 exemplifica a utilização do comando **if**. Considerando por exemplo a variável i seja igual a 5, isto é, menor que 10, o procedimento imprime "O valor de i = 5 é menor que 10" e, supondo i seja igual a 15, isto é, maior que 10, o procedimento imprime "O valor de i = 15 é maior que 10".

```
if  ( i < 10  ):
   print('O valor de i  = '+ str(i) +' é menor que 10 ')
else:
   print('O valor de i = '+ str(i)+ ' é maior ou igual a 10')
```

Fragmento de Código 1.5. Exemplo de Implementação em Python do operador de seleção simples.

Seleção simples como parâmetro de retorno (inline)

Em alguns casos necessitamos de fazer uma comparação simples e dependendo do caso, retornar um determinado valor. Em Python podemos utilizar o comando if na forma dita em linha (*inline)*. A sintaxe da utilização deste operador na forma *inline* é mostrada a seguir

```
return <valor do retorno no caso da condição verdadeira>
       if <condição> else <valor de retorno no caso da condição falsa>
```

O fragmento de código 1.6 exemplifica a utilização do comando if, na forma inline, sendo utilizado na função gt10. Neste caso, a variável **i**, passado por parâmetro, será comparada se é maio que 10 (**i** > 10) . Caso o resultado da comparação seja verdadeiro o valor retornado será igual a True, e caso contrário, símbolo, o valor retornado será False.

```
def gt10(i):
    return True if i > 10 else False
```

Fragmento de Código 1.6. Implementação em Python da utilização do operador if na forma inline.

Seleção múltipla

Em alguns casos necessitamos fazer várias comparações ao mesmo tempo. Por exemplo, seja uma variável **v**, que representa uma vogal qualquer (a, e, i, o, u). Se desejássemos, para cada valor da variável **v**, executarmos uma determinada sequência de comandos, poderíamos utilizar um ninho de **if** ou, utilizarmos uma outra forma denominada de **if.. elif**, que corresponde à seleção múltipla. A sintaxe do operador **if .. elif** pode ser exemplificada da seguinte forma

```
if <expressão 1>:
   # corpo do caso 1
elif <expressão 2>:
   # corpo do caso 2
elif <expressão 3>:
   # corpo do caso 3
else:
   # corpo de outro caso
```

A figura 1.2 ilustra o fluxograma do comando **if ... elif** com os respectivos desvios dependendo se a condição for verdadeira ou falsa.

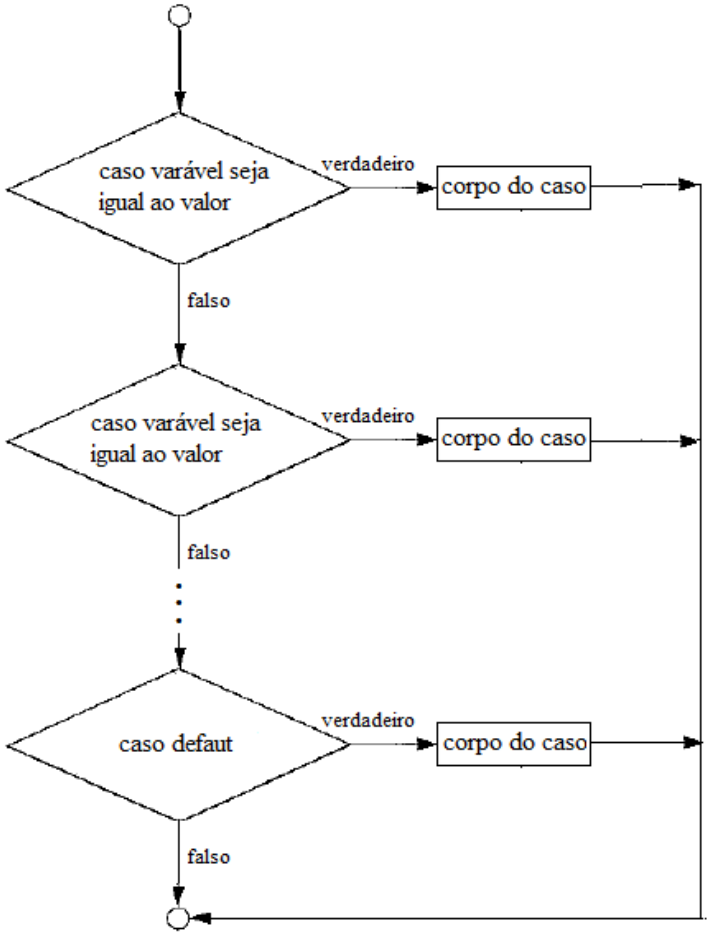


Figura 1.2 – Fluxograma do comando if ..elif.

O fragmento de código 1.7 exemplifica a utilização da seleção múltipla utilizando o comando **switch**. É importante frisar que a variável i é testada em cada caso e se, verdadeiro, executa os comandos que estão no caso especificado.

```
if i == 1:
   print('i é igual a  1 : ',i)
elif i > 1:
   print('i é maior que 1 : ',i)
else.:
   print('i é menor que 1 : ',i)
```

Fragmento de Código 1.7. Implementação em Python do operador de seleção múltipla.

Estruturas de Repetição

As estruturas de repetição podem ser da forma controlada ou indefinida. A forma de repetição indefinida é aquela em que o loop será executado um número indefinido de vezes e, a controlada, é aquela em que o loop será executado um número de vezes pré-estabelecido. Em Python para a repetição indefinida temos a estrutura: enquanto (**while**) e, para a repetição controlada. temos a estrutura: para (**for**).

Repetição indefinida

A estrutura de repetição indefinida em Python é o comando **while.**

Repetição com teste no início do loop

A estrutura de repetição indefinida **while** é uma estrutura de repetição em que o teste é feito sempre antes de executarmos o bloco de comandos da estrutura. A sintaxe é da seguinte forma

```
while < expressão lógica > :
    # Corpo do while
```

A figura 1.4 ilustra o fluxograma do comando **while**.

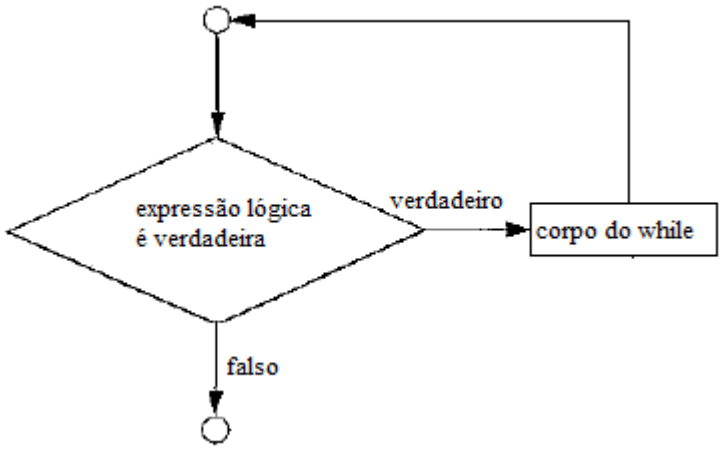


Figura 1.5 – Fluxograma do comando while .

O fragmento de código 1.9 exemplifica a utilização da seleção múltipla utilizando o comando **while**.

```
i = 1
while ( i < 10):
    i = 2*i
    print('O valor de i =  ' + str(i))
```

Fragmento de Código 1.9. Implementação em Python do operador de repetição indefinida **while**.

Os valores impressos após a execução do fragmento de código 1.9 estão listados a seguir

```
O valor de i =  2
O valor de i =  4
O valor de i =  8
O valor de i =  16
```

Repetição com teste no fim do loop

A estrutura de repetição indefinida **while** é uma estrutura de repetição em que o teste é feito sempre antes da execução do bloco de comandos da estrutura, logo, para entrarmos no loop, ignorando o teste de entrada, implementamos o while True, que sempre é verdadeiro, e após a execução dos comandos fazemos o teste de controle. No caso verdadeiro, utilizamos o comando break, para sair do loop. A sintaxe é da seguinte forma

```
while True:
    # Corpo do while
    if < condition> :
       break
```

A figura 1.6 ilustra o fluxograma do comando **while** com o teste no final.

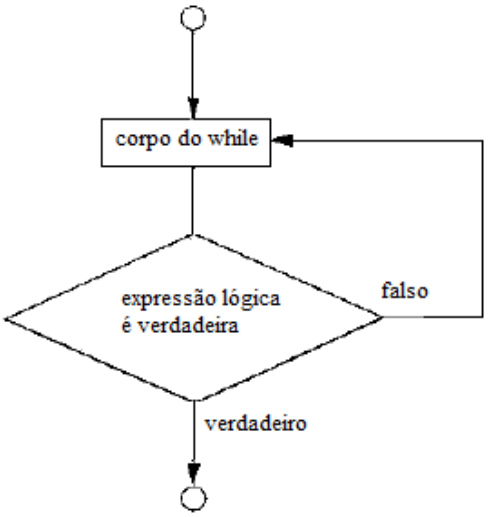


Figura 1.6 – Fluxograma do comando while True.

O fragmento de código 1.10 exemplifica a utilização da seleção múltipla utilizando o comando **while** com o teste no final. A entrada é feita sempre com **while True.**

```
i = 1
while True:
    i = 2*i
    print('O valor de i =  ' + str(i))
    if i > 10:
        break
```

Fragmento de Código 1.10. Implementação em Python do operador de repetição indefinida **while** com o teste no final.

Os valores impressos após a execução do fragmento de código 1.10 estão listados a seguir

```
O valor de i =  2
O valor de i =  4
O valor de i =  8
O valor de i =  16
```

Repetição controlada

A estrutura de repetição controlada em Python é a estrutura **for**. A seguir descreve-se as funcionalidades, sintaxe e exemplos do comando **for**.

Comando for

A estrutura de repetição controlada **for** é uma estrutura de repetição em que o número de iterações é controlado por uma variável. A sintaxe da utilização do comando for é a seguinte

```
for <variável de controle> in range(<valor inícial>,<valor de parada>,
                                    <incremento/decremento>):
   # Corpo do for
```

Inicialmente é feito a inicialização da variável de controle, e em cada iteração é feito o teste da condição de parada e então é feito um incremento ou decremento da variável de controle. A figura 1.7 ilustra o fluxograma do comando **for**.

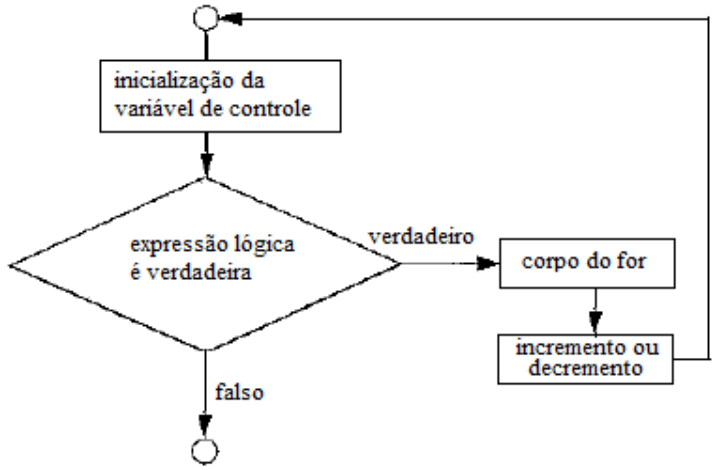


Figura 1.7 – Fluxograma do comando for.

O fragmento de código 1.11 exemplifica a utilização da seleção múltipla utilizando o comando **for**.

```
# for com os três parâmetros
# incremento de 2 unidades
for i in range(0,10,2):
   print('O valor de i =  ' + str(i))

# for com os três parâmetros
# decremento de duas unidades
for i in range(10,4,-2):
   print('O valor de i =  ' + str(i))

# for com dois parâmetros
# (limite inferior, limite superior)
# com incremento default em uma unidade
for i in range(6,10):
   print('O valor de i =  ' + str(i))

for com um parâmetro (limite superior)
# com incremento default em uma unidade
for i in range(5):
   print('O valor de i =  ' + str(i))
```

Fragmento de Código 1.11. Implementação em Python do operador de repetição controlada **for**.

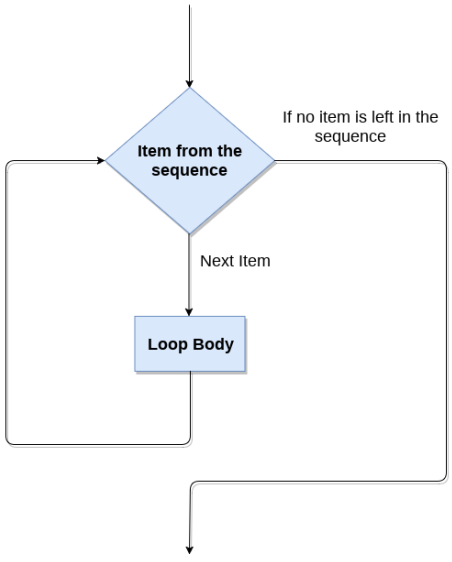
Os valores impressos após a execução do fragmento de código 1.11 estão listados a seguir

```
O valor de i =  0
O valor de i =  2
O valor de i =  4
O valor de i =  6
O valor de i =  8

O valor de i =  10
O valor de i =  8
O valor de i =  6

O valor de i =  6
O valor de i =  7
O valor de i =  8
O valor de i =  9

O valor de i =  0
O valor de i =  1
O valor de i =  2
O valor de i =  3
O valor de i =  4
```

Comando for com limite superior enumerável

Outra forma de utilizarmos a estrutura **for** é quando desejarmos fazer uma iteração com os elementos de um conjunto enumerável qualquer. A sintaxe é feita da seguinte forma

```
conj  = < conjunto enumerável de Objetos >
for element in conj:
   #   corpo do laço
```



O conjunto enumerável pode ser, por exemplo, uma lista. A estrutura lista será estudada no próximo capítulo. No fragmento de código 1.12 ilustra-se a utilização da varredura dos elementos da lista **a**. Inicialmente é feita a inicialização dos elementos da lista, e no comando **for**, a variável value, do tipo inteiro, pois o vetor é do tipo inteiro, recebe os valores de cada variável na sequência de acordo com a inicialização.

```
a = [10,20,30,40,50]
for value in  a:
   print(value)
```

Fragmento de Código 1.12. Implementação em Python do operador de repetição definida por **enumeração**.

Os valores impressos após a execução do fragmento de código 1.12 será

```
10
20
30
40
50
```

O comando continue

Em alguns casos, em um laço qualquer, necessitamos que se uma determinada condição for satisfeita temos que ignorar todos os comandos após esta condição. Neste caso utilizarmos o comando **continue**. Na execução deste comando o controle vai para o início do laço. O fragmento de código 1.114exemplifica a utilização do comando **continue**.

```
i = 0
while i < 10:
   i += 1
   if i % 2 == 0:
      continue
   print('O valor de i =  ' + str(i))
```

Fragmento de Código 1.14. Implementação em Python do operador de repetição indefinida **while** com o operador **continue**.

Os valores impressos após a execução do fragmento de código 1.14 estão listados a seguir

```
o valor de i =  1
o valor de i =  3
o valor de i =  5
o valor de i =  7
o valor de i =  9
```

Tipo Abstrato de Dados

Existem momentos que necessitamos criar um tipo de dados que não é um dado primitivo. Por exemplo, criar uma variável do tipo complexo. Sabemos que um número complexo é do tipo $z = a + bi$, onde $a, b \in \Re$ e $i = \sqrt{-1}$.

Como representar este tipo de abstrato de dados em Python? Neste caso utilizamos o Conceito de Classe que será estudada logo a seguir. Primeiramente, vamos apresentar o conceito de Objeto que será uma instância qualquer de uma classe.

O Python já implementa a classe complexo. Para criarmos uma variável do tipo complexo basta ativar o constructor da classe. Por exemplo:

```
c = 1+3j
```

O Python cria uma variável c, do tipo complexo, em que a parte real é igual a 1 e, a parte imaginária é igual a 3.

```
>>> c.real
    1.0
>>> c.imag
    3.0
```

Objetos

Objeto é um determinado elemento do mundo real. Por exemplo, Grizendi é um objeto do tipo abstrato de dados Pessoa. Este objeto possui determinadas características (denominados de atributos) que desejamos gerenciar, como por exemplo: nome, RG, CPF e endereço. Pode-se destacar também que determinadas ações que este objeto poderá executar como, por exemplo: Obter o nome, armazenar o nome, são ações que incidem nos atributos e denominadas de métodos.

A Figura 1.8 (a) ilustra a representação gráfica de um objeto e suas principais características (nome do objeto, atributos e métodos) e a letra b) ilustra um objeto específico de nome Grizendi com os valores inicializados dos atributos.

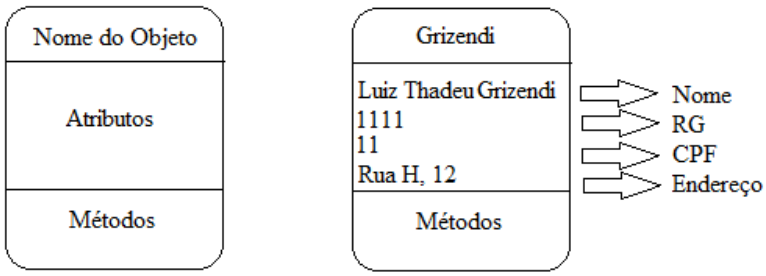


Figura 1.8. Representação gráfica de uma classe e de um exemplo.

Classe

Uma Classe é um repositório de objetos de mesmo tipo. Por exemplo, os objetos do mesmo tipo Pessoa, formam a denominada Classe Pessoa, isto é, os objetos do tipo Pessoa, como por exemplo: Grizendi, Lucas e Gabriel são exemplares da Classe Pessoa, denominadas de **instâncias** da Classe Pessoa. Basicamente, uma classe é composta pelo nome da Classe (Pessoa), os seus atributos (Nome, RG, CPF e Endereço) e, os métodos de manipulação dos Objetos desta referida Classe (getNome(), setNome(),..). O método getNome() é um método que retorna o nome da pessoa, setNome() é um método utilizado para armazenar o nome. A figura 1.9 ilustra graficamente estes elementos.

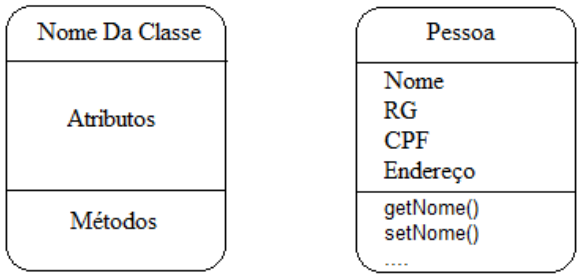


Figura 1.9. Representação gráfica de uma classe e de um exemplo.

O Fragmento de Código 1.15 ilustra a criação da classe Pessoa em Python. Neste exemplo podemos destacar os modificadores do estado de cada atributo que serão abordados a seguir.

```
class Pessoa:
 def __init__(self,nome=None,rg=None,cpf=None,end=None):
      self.nome = nome    # public
      self._rg = rg       # private
      self.__cpf = cpf    # protected
      self.end = end      # public
```

Fragmento de Código 1.15. Implementação em Python de parte Classe Pessoa.

Os modificadores do estado de uma classe, método ou atributo estão descritos a seguir.

Modificadores

São operadores que modificam o estado de uma determinada Classe, método ou atributo. Os modificadores em Python são : **private**, **protected**, **public** e **static**. A seguir descrevem-se estes modificadores e os significados.

private é um modificador que identifica que o atributo, a classe ou o método são do tipo **privado**. Quando o atributo for do tipo **private**, significa que este atributo, classe ou método, só será enxergado dentro da classe que ele foi criado.

private, significa que esta classe só será enxergada dentro da classe que ela foi criada. Finalmente, quando o método for do tipo **private**, significa que o atributo, classe ou método só poderá ser enxergado dentro da classe que ele foi criado. No Python utilizamos o sublinhado simples antes do nome da classe, método ou atributo, (_).

protected é um modificador similar ao modificador private. A diferença é que o atributo, classe ou método são enxergados na execução da própria classe e privados quando classe é importada. No Python utilizamos o sublinhado simples antes do nome da classe, método ou atributo (__)

public é um modificador que identifica que o atributo, a classe ou o método são do tipo **público**, isto é, o atributo, método ou classe podem ser enxergados fora da classe em que foram criados.

static é um modificador que identifica que o método é do tipo estático, isto é, o método pode ser referenciado sem uma inicialização da classe. Deve-se colocar o atributo @staticmethod antes da implementação do método.

Atributos

Os atributos são os campos tipos de dados que pertencem à classe. Por exemplo na classe Pessoa, implementada no fragmento de código 1.15, temos como atributos: nome, RG, CPF e endereço.

```
class Pessoa:
 def __init__(self,nome=None,rg=None,cpf=None,end=None):
      self.nome = nome    # public
      self._rg = rg       # private
      self.__cpf = cpf    # protected
      self.end = end      # public
```

Métodos

Os Métodos são funções e procedimentos que realizam ações com os objetos instanciados das respectivas Classes. O fragmento de código 1.16 ilustra alguns métodos da classe. O método __getRG(), possui o modificador protected, logo esse método não é enxergado por um programa externo à classe. O método _getnome(), possui o modificador private, logo esse método não é importado quando se faz o import da classe. Finalmente temos o método getCPF() com nenhum modificador, logo ele é considerado público, que retornará uma variável protegida da classe.

```
def __getRG(self):    # protected method
     return self._rg

def getCPF(self):     # public method
     return self.__cpf

def _getnome(self):   # private method
     return self.nome
```

Fragmento de Código 1.16. Implementação em Python dos métodos __getRG(), getCPF() e _getNome().

Construtores

Os Construtores são métodos de inicialização de qualquer Objeto de uma Classe. Todo objeto do tipo Classe deve ser inicializado antes de sua manipulação. A sintaxe de um construtor é da seguinte forma

```
def __init__(self, variáveis ):
```

O fragmento de código 1.17 exemplifica o construtor da classe Pessoa. Pode-se observar, as variáveis do parâmetro são inicializadas com None, pois, se na criação da classe, não for passado esse parâmetro ele será considerado igual a None.

```
def __init__(self,nome=None,rg=None,cpf=None,end=None):
    self.nome = nome    # public
    self._rg = rg       # private
    self.__cpf = cpf    # protected
    self.end = end      # public
```

Fragmento de Código 1.17. Implementação em Python do construtor da classe Pessoa.

O operador self

No fragmento de código 1.17 o construtor tem um parâmetro de entrada a variável CPF tem o mesmo nome do atributo CPF da classe Pessoa, neste caso utilizamos o operador **self** para dizer que a variável do atributo da classe CPF recebe a variável de memória CPF.

Manipulando os atributos de uma classe

Se os atributos forem do tipo **private**, necessitamos criar métodos para atribuição e obtenção do atributo (get e set). Geralmente utiliza-se a sintaxe

```
def  getNomedoAtributo(self) :
        # corpo do método getNomedoAtributo

def  setNomeAtributo(self, NomeAtributo):
        # corpo do método setNomedoAtributo
```

O fragmento de código 1.18 exemplifica os métodos get e set do atributo Nome da classe Pessoa.

```
def getnome(self):
    return self.nome

def setnome(self,nome):
    self.nome = nome
```

Fragmento de Código 1.18. Implementação em Python dos métodos getNome() e setNome().

O método main (Principal)

Este método será executado pelo PYTHON através do seu **run-time**, isto é, quando desejarmos executar uma determinada classe, esta classe deverá ter o método **main**. A sintaxe deste método em Python é a seguinte

```
if __name__ == '__main__':
    # compo do método main
    p = Pessoa('Luiz',1,2,'rua h')
    print(p.getCPF())
    print(p.__getRG())
    print(p._getnome())
    print(p.nome)
```

Instanciação

Quando desejarmos criar um objeto de uma determinada classe, devemos, ativar o **constructor** correspondente da classe para a inicialização deste objeto, por exemplo.

```
p = Pessoa('Luiz',1,2,'rua h')
```

Neste instante criou-se em memória um objeto, ou uma instância da classe Pessoa, denominado de p, com os seguintes atributos: nome = ‘Luiz’, RG = 1, CPF =2 e Endereço = ‘rua h’.

A figura 1.10 ilustra graficamente este objeto.

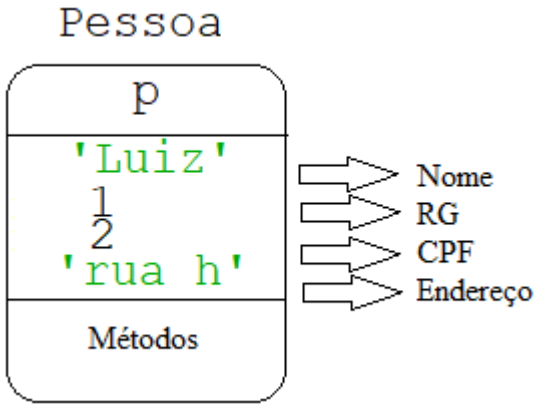


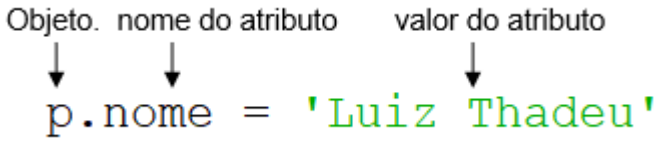
Figura 1.10. Representação gráfica da classe Pessoa.

Manipulando os atributos de uma classe utilizando os métodos ou diretamente

Quando desejarmos manipular os atributos de uma classe externamente, e estes objetos são do tipo **public**, podemos alterar este objeto diretamente da seguinte forma

Objeto.<nome do atributo> = < valor do atributo >;.

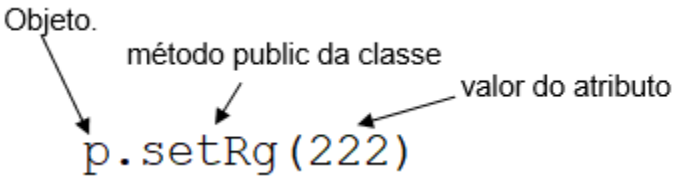
Exemplo, alterar o nome do objeto p, do tipo Pessoa, classe Pessoa, para "Luiz Thadeu", escrevemos,



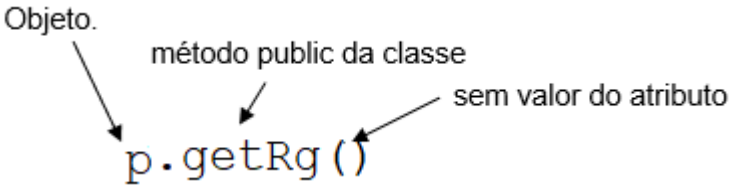
Quando desejarmos manipular os atributos de uma classe externamente, e estes objetos são do tipo **private ou protected**, utilizamos os métodos públicos **get** e **set** da classe para fazer isto.

Objeto.<método public da classe> (< valor do atributo >);

Exemplo, alterar o RG do objeto p, do tipo Pessoa, classe Pessoa, para "222", escrevemos



Para obtermos o RG do objeto p, do tipo Pessoa, escrevemos



Tratamento de Exceções

Ao executarmos uma determinada rotina, esta rotina pode gerar alguma exceção e o programa terminar de forma súbita. Este problema pode ser resolvido fazendo um tratamento das exceções.

Considere por exemplo o fragmento de código

```
def inverso(n):
    return 1/n
```

Se este método fosse executado, passando como parâmetro o valor zero, o programa vai terminar de forma súbita. Então, como tratarmos este problema sem que o problema termine de forma inadequada. Nestes casos, uma forma de tratamento da exceção está em acrescentarmos ao código os operadores **try** e **except**.

Operadores try e except

O operador **try** delimita a parte do código considerando que não tenha uma exceção, O operador **except** (que deve vir sempre atrelado ao **try**, não podendo existir sozinho) define o código a ser executado caso ocorra uma exceção.

A implementação da função inverso com utilização dos operadores try e except está descrita a seguir

```
def inverso(self,n):
    try:
        return 1/n
    except ArithmeticError:
            print('Divisão por zero!')
```

A seguir, na Tabela 1.7, descrevem-se alguns casos comuns de exceções e os seus respectivos significados.

Tabela 1.7 – Exemplos de casos de exceções em Python.

Exceção	**Causa**
AssertionError	Surge quando o comando assert falha
AttributeError	Surge quando uma atribuição ou referência falha
EOFError	Surge quando a função input() chega na condição de fim de arquivo
FloatingPointError	Surge quando uma operação de ponto flutuante falha
GeneratorExit	Surge quando o método close() do gerador é chamado
ImportError	Surge quando o módulo importado não é encontrado
IndexError	Surge quando o índice de uma sequência está fora de alcance
KeyError	Surge quando uma chave não é encontrada no dicionário
KeyboardInterrupt	Surge quando um usuário aperta o botão de interrupção (CTRL + C ou delete)
MemoryError	Surge quando acaba a memória de uma operação
NameError	Surge quando uma variável não é encontrada no escopo local ou global
NotImplementedError	Surge por métodos abstratos
OSError	Surge quando alguma operação de sistema causa algum erro de sistema
OverflowError	Surge quando uma operação aritmética é muito grande para ser representada
ReferenceError	Surge quando um proxy de referência fraco é usado para acessar um garbage collected referente
RuntimeError	Surge quando um erro não se encaixa em nenhuma outra categoria
StopIteration	Surge pela função next() para indicar que não há mais itens para o iterador retornar

Exceção	Causa
SyntaxError	Surge pelo parser quando um erro de sintaxe ocorre
IndentationError	Surge quando não indentamos nosso código
TabError	Surge quando a indentação consiste de espaços e tabs impróprios
SystemError	Surge quando o interpretador detecta um erro interno
SystemExit	Surge pela função sys.exit()
TypeError	Surge quando uma função ou operação é aplicada a um objeto de tipo incorreto
UnboundLocalError	Surge quando uma referência é feita para uma variável local em uma função ou método, porém nenhum valor está preso à variável
UnicodeError	Surge quando existe um erro relacionado a Unicode codificação ou decodificação
UnicodeEncodeError	Surge quando um erro de codificação de Unicode ocorre
UnicodeDecodeError	Surge quando um erro de decodificação de Unicode ocorre
UnicodeTranslateError	Surge quando um erro de tradução de Unicode ocorre
ValueError	Surge quando uma função pega um argumento de tipo correto, porém de valor impróprio
ZeroDivisionError	Surge quando a segunda divisão ou módulo é zero

Operador: raise TypeError

Podemos também tratar o erro usando o comando raise TypeError que chama a diretiva de parada e exibe o tipo de erro. Por exemplo o código da função inverso poderia ser escrita da seguinte forma:

```
def inverso(n):
    if n == 0:
            raise TypeError('Não existe o inverso de '+str(n))
    return 1/n
```

Poliformismo

Algumas vezes é necessário criarmos um molde para determinadas classes. Por exemplo, as classes aritmeticProgression (Progressão Aritmética) e geometricProgression (Progressão Geométrica) são do tipo Progression (progressão). Se não desejarmos criar uma classe Progression, pois esta classe não será instanciada, podemos utilizar a ideia de **interface**. A **interface** é um molde que contém os rótulos (labels) dos métodos que serão utilizados e visíveis pelas classes que implementam este molde.

O fragmento 1.19 ilustra a criação da interface Progression. Primeiramente observa-se, a importação da classe abc (Abstract Base Classes) do Python e, o parâmetro, metaclass = ABCMeta, informando que esta é uma classe abstrata, no caso uma interface, pois nenhum código de métodos está implementado.

```
import abc
class Progression(metaclass=abc.ABCMeta):
    @abc.abstractmethod
    def setFirst(self,first):
         pass
    @abc.abstractmethod
    def setBase(self,base):
         pass
    @abc.abstractmethod
    def getFirst(self):
         pass
    @abc.abstractmethod
    def getBase(self):
         pass
    @abc.abstractmethod
    def sumTerms(self,n):
        pass
    @abc.abstractmethod
    def generalTerm(self,n):
        pass
```

Fragmento de Código 1.19. Implementação em Python da interface Progression.

Classe Abstrata

Algumas vezes é necessário criarmos um repositório de métodos e atributos que várias classes derivadas desta classe possuem em comum. Para que não tenhamos uma duplicação de código, podemos criar uma classe Abstrata que conterá os atributos e métodos que sempre farão parte das suas classes derivadas.

O fragmento de código 1.20 ilustra a classe abstractProgression que implementa a interface Progression. Primeiramente observa-se, a importação da interface Progression.

Podem-se destacar dois atributos: base, que será a razão da progressão e. o first. que indica o primeiro termo desta progressão.

Os métodos getFirst(), setFirst(), getBase() e setBase() são os métodos para a manipulação dos atributos da classe.

Os métodos sumTerms(), para a obter a soma dos termos da progressão, o método generalTerm(), para obter o termo geral da progressão e o método Interpolation() para obtermos uma interpolação de termos entre dois termos informados como parâmetros, não possuem implementação nesta classe abstrata pois eles variam de acordo com as especificidades do tipo de progressão.

Nestes casos, observa-se a utilização do operador **@abc.abstractmethod,** informando que o método é um método abstrato e, será implementado em cada especialização. O comando pass, informa que a implementação será sobrecarregada pela implementação da especialização.

```
class abstractProgression(Progression):
    def __init__(self,base = None, first = None):
         self.base = base  # Razão da Progressão
         self.first = first # primeiro termo da Progressão
    def setFirst(self,first):
         self.first=first
    def setBase(self,base):
         self.base = base
    def getFirst(self):
         return self.first
    def getBase(self):
         return self.base
    @abc.abstractmethod
    def sumTerms(self,n):
        pass
    @abc.abstractmethod
    def generalTerm(self,n):
        pass
    @abc.abstractmethod
    def interpolation(self,f, l, n):
        pass
```

Fragmento de Código 1.20. Implementação em Python da classe abstrata abstractProgression.

Herança

A Herança é uma propriedade muito importante na programação orientada à objetos. Esta propriedade é um mecanismo no qual uma determinada classe herda todas as características, métodos e atributos de outras classes. Este mecanismo faz com que reaproveitamos todos os atributos e métodos definidos anteriormente em uma outra classe hierarquicamente superiora.

Herança simples

Neste caso uma classe mãe pode ser generalizada em outras classes filhos e, sendo assim as classes filhos são especializações da classe mãe.

Por exemplo: a classe abstractProgression é uma generalização das classes aritmeticProgression e geometricProgression, neste caso as classes aritmeticProgression e geometricProgression são especializações da classe abstractProgression que possui a interface Progression A figura 1.11 ilustra esta hierarquia de classes.

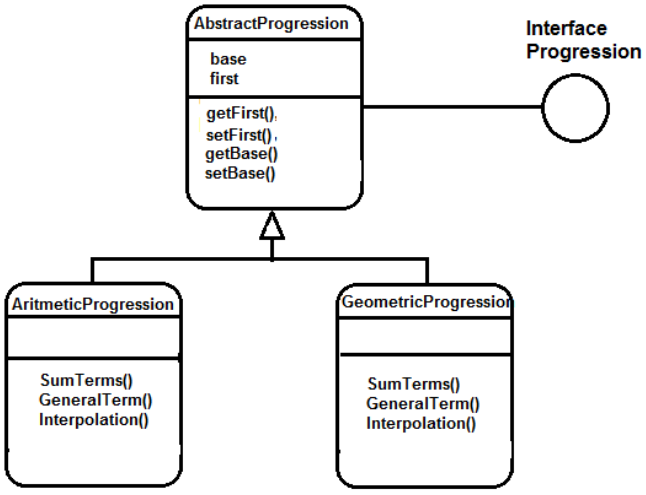


Figura 1.11. Representação gráfica em UML de uma classe e de um exemplo.

O fragmento de código 1.21 ilustra a implementação da classe aritimeticProgression. Neste caso, como ela é uma especialização da classe abstractProgression, logo, deve-se fazer a referência à classe mãe, passando como parâmetro da classe.

O construtor __init__() faz uma chamada ao constructor da classe superiora supper().__init__(), passando os parâmetros de inicialização da classe. lasse abstractProgression, logo, deve-se fazer a referência à classe mãe, passando como parâmetro da classe.

```
class arithmeticProgression(abstractProgression):
    def __init__(self,base = None, first = None):
        super().__init__(base,first)

    def generalTerm(self,n):
        return self.getFirst()+(n-1)*self.getBase()

    def sumTerms(self,n):
        return ((self.getFirst()+self.generalTerm(n))*n/2)

    def interpolation(self, f, l, n):
        b=(l-f)/(n+1)
        l = []
        for i in range(0, n):
            f += b
            l.append(f)
        return l

    @staticmethod
    def get_first_base(i,vi,j,vj):
        i -= 1
        j -= 1
        b=(vj-vi)/((j-i-1)+1)
        return vi- b*(i), b
```

Fragmento de Código 1.21. Implementação em Python da classe arithmeticProgression.

O método get_firt_base(i,vi,i,vj) é um método estativo (@staticmethod) pois ele pode ser executado sem, primeiramente, instanciarmos um objeto da classe aritimeticProgression.

A execução do fragmento de código a seguir ilustra esta característica. Observe que o método para ser executado é da forma arithmeticProgression.get_first_base(), isto é, na forma <classe>.método estático. O primeiro parâmetro, valor 3, é o índice do termo cujo valor é 8 (segundo parâmetro), o terceiro parâmetro, 6, é o índice do termo cujo valor é 17 (quarto parâmetro), Em resumo, a função é entendida como: dado o terceiro termo e o sexto termos, determine o primeiro termo e a razão da PA.

```
f,b = arithmeticProgression.get_first_base(3,8,6,17)
print('a1 = ',f,'r : ',b)
```

Sobrecarga

Os métodos generalTerm(), sumTerms() e interpolarion(), implementados no fragmento de código 1.21, sobrepõem aos métodos definidos na classe mãe, Progression(). Este procedimento é denominado, em orientação à objetos, sobrecarga.

O fragmento de código 1.22 ilustra a implementação da classe geometricProgression. Neste caso também, ela é uma especialização da classe Progression, logo, deve-se fazer a referência à classe mãe

```
import math
class geometricProgression(abstractProgression):
  def __init__(self,base = None, first = None):
       super().__init__(base,first)

  def generalTerm(self,n):
       return self.getFirst()*math.pow(self.getBase(),n-1)

  def sumTerms(self,n):
    if self.getBase() > 0 and  self.getBase() < 1:
         return self.getFirst()/(1-self.getBase())
    else:
         return (self.getFirst()*
                (math.pow(self.getBase(),n)-1)/(self.getBase()-1))

  def interpolation(self, f, l, n):
    b=math.pow(l/f,math.pow(n+1,-1))
    l = []
    for i in range(0, n):
         f *= b
         l.append(f)
    return l
```

Fragmento de Código 1.22. Implementação em Python da classe geometricProgression.