Inteligência Artificial

José Mendes 107188

2023/2024

1 Noções de Programação Declarativa

1.1 Programação Declarativa

A Programação Declarativa abstrai-se da implementação, focando-se apenas na descrição do que se pretende fazer, enquanto faz uso de dois paradigmas:

- **Programação Funcional**, baseado em funções/calculo-lambda, a entidade central é a função;
- **Programação em Lógica**, baseado em lógica de primeira ordem, a entidade central é o predicado;

1.2 Paradigma Imperativo

O fluxo de operações é especialmente sequenciado de operações com foco na forma como as tarefas são executadas (**instrução**). Podemos **alterar o conteúdo em memória** e ainda (instruções de afetação/atribuição) e ainda **realizar análise de casos** (if-then-else, switch/case, ...), **processamento iterativo** (while, repeat, for, ...) e ter associados **sub-programas** (procedimentos, funções).

Exemplo: SQL é uma linguagem declarativa, uma vez que nos seus comandos apenas descrevemos o que queremos obter. Assim, o programador é abstraído da forma como as operações são executadas na prática, ficando essa tarefa a cargo do compilador.

1.3 Paradigma Declarativo

	Funcional	Lógico
Fundamentos	Lambda calculus	Lógica de primeira ordem
Conceito central	Função	Predicado
Mecanismos	Aplicação de funções Unificação uni-direccional Estruturas decisórias	Inferência lógica (resolução SLD) Unificação bi-direccional
Programa	Um conjunto de declarações de funções e estruturas de dados	Um conjunto de fórmulas lógicas (factos e regras)

A sua origem data da segunda metade do século XX, mas ainda hoje é amplamente usada e até está em crescimento em áreas como a Inteligência Artifical.

1.4 Programação Funcional

Possibilidade de definir funções localmente e sem nome. Por exemplo as funções lambda presentes em Lisp e Python.

1.5 Programação em Lógica

Um programa é uma teoria sobre um domínio. Por exemplo, temos **socrates é um homem**, $homem(socrates)$ e, um **homem é mortal**, $homem(X) :- mortal(X)$. Se perguntarmos se **socrates é mortal**, $mortal(socrates)$, a resposta é sim.

1.6 Atitude do programador

A programação declarativa, dada a sua elevada expressividade, é pouco compatível com aproximações empíricas (ou seja, tentativa e erro) à programação. Primeiro é preciso pensar bem na estrutura do problema antes de começar a teclar.

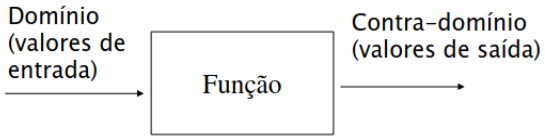
Passos a seguir: Perceber o problema → Desenhar o programa → Escrevê-lo → Rever e testar

1.7 Características da Programação Funcional

- A entidade central é a função;
- A noção de função é diretamete herdada da matemática (ao contrário das linguagens imperativas, que pode ser muito diferente);
- A estrutura de controlo fundamental é a "aplicação de funções";
- A noção de "tipo da função" captura a noção matemática de domínio (de entrada e saída);
- Os elementos do domínio de entrada e saída podem ser funções;

1.8 Função

Tem valores de entrada (domínio) e valores de saída (contradomínio).



1.9 Programação em Lógica

Um programa numa linguagem baseada em lógica representa uma teoria sobre um problems. Um programa é uma sequência de frases ou fórmulas representando **factos** (informação sobre objetos do problema/domínio de aplicação) e **regras** (leis gerais sobre esse problema/domínio). Implicitamente as frases estão reunidas nume grande conjunção, e cada frase está universalmente quantificada. Portanto, **programação declarativa**.

2 Programação ao estilo funcional em Python

2.1 Python

Criada no final dos anos 90, é uma linguagem de programação interpretada, iterativa, portável, funcional, orientada a objetos e de implementação aberta. Tem como principais objetivos: simplicidade sem prejuizo da utilidade, programação modular, legibilidade, desenvolvimento rápido, facilidade de integração, nomeadamente com outras linguagens. Apresenta-se como uma linguagem **multi-paradigma**:

- **Programação funcional:** Expressões lambda, funções de ordem superior, listas com sintaxe simples, iteradores, ...
- **Programação OO:** Classes, objetos, métodos, herança, ...
- **Programação imperativa/modular:** Atribuição, sequências, condicionais, ciclos, ...

2.1.1 Python vs Java

Comparativamente ao JAVA revela-se menos consisa porque os espaços são sintaticamente relevantes e o código, não compilado para código nativo (interpretado), demora mais tempo a executar (diferença residual), no entanto apresenta uma maior facilidade de desenvolvimento e legibilidade.

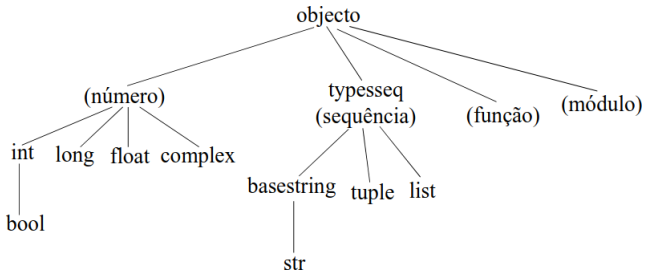
2.1.2 Áreas de aplicação

As principais áreas de aplicação são: interligação de sistemas, aplicações gráficas, aplicações para bases de dados, multimédia, internet protocol/web e robótica e inteligência artificial.

2.2 Dados, ou "objetos"

Objeto: no contexto de Pyhton, esta designação é aplicada a qualquer dado que possa ser armazenado numa variável, ou passado como parâmetro a uma função.

Cada objeto é caracterizado por uma identidade/referência (identifica a posição de memória onde está armazenado), um tipo e um valor. Alguns tipos de objetos podem ter atributos e métodos, outros (classes) podem ter sub-tipos (sub-classes).



Alguns destes são imutáveis, como as **str** e **tuple**. Para determinar o tipo de um objeto utiliza-se a função pré-definida **type()**.

2.3 Variáveis

As variáveis guardam objetos, mas não são declaradas (inicializadas apenas) nem têm tipos associados (o tipo é do objeto!). Praticamente tudo pode ser atribuido a uma variável, incluindo funções, módulos e classes. Similar as linguagens imperativas, e ao contrário das funcionais, o valor de uma variável pode ser alterado. Não se pode ler o valor de uma variável se este não tiver sido previamente inicializado.

2.4 Intrução de atribuição

Tal como é habitual na programação imperativa, e ao contrário do habitual na programação declarativa, Python possui instrução de atribuição.

Exemplo:

```
n = 10
a = b = c = 0
x = 7.25
cad = "cadeia"
t = (n, x)
l = [1, 2, 'quatro', 5.0]
```

Pode user a operação de atribuição para decompor estruturas:

```
triplo = (1, 2, 3)
(i, j, k) = triplo
```

2.5 Operadores

Operadores comuns matemáticos e lógicos. É importante saber que objetos de tipos diferentes nunca são iguais (exceto os diferentes tipos de números).

2.5.1 Operadores lógicos

Na conjunção e disjunção, o segundo elemento só é avaliado de for necessário para determinar o resultado.

```
if False and a: # a nao e avaliado
  ...

if True or a: # a nao e avaliado
  ...
```

2.6 Acesso a sequências

Os elementos das sequências são acedidos através de índices inteiros consecutivos (primeiro elemento tem índice 0).

É possível extrair "fatias" das sequências. A indexação é circular, pelo que podemos aceder a índices negativos.

```
# [inf:sup] retorna a sequencia entre os indices inf e sup-1 (uma opia da lista)

# Para fazer uma copia integral da lista usa-se [:]
```

Detalhe importante: instrução de atribuição, em vez de copiar valores, limita-se a associar um dado identificador a um dado objecto. Desta forma, a atribuição da variável x a uma variável y, apenas tem como resultado associar y ao mesmo objecto que x já estava associada. No caso dos objetos mútaveis, temos de ter cuidado:

```
a = [1, 2, 3]
b = a
b[1:2] = []
print(a) # [1, 3]
```

2.7 Definição de funções

Funções sem return, retornam None, são também conhecidas como procedimentos. As funções podem ser recursivas e são objetos do tipo **function**.

Os parâmetros são passados às funções segundo um mecânismo "passagem por valor" ("call by value"). Os parâmetros são passados por referêcia, não cópias!.

Se atribuirmos um novo objeto a uma variável passada por parâmetro, essa atribuição ocorre apenas no espaço de nomes da função (imagem à esquerda). Se modificarmos um objeto passado por parâmetro (por exemplo, apagar um elemento de uma lista), isto não altera a referência do objeto, e portanto vai permanecer após o retorno da função (imagem à direita).

```
>>> def incr(x):
...     x=x+1
...     return x
...
>>> n=10
>>> incr(n)
11
>>> n
10
```

```
>>> def acresc(l,x):
...     l[0:0]=[x]
...     return l
...
>>> lista=[5,12]
>>> acresc(lista,30)
[30,5,12]
>>> lista
[30,5,12]
```

O problema anterior resolve-se trabalhando sobre uma cópia local, no caso de uma lista l, podemos usar l[:], que dá uma cópia integral.

Podemos passar parâmetros com valores por defeito, podendo se, na chamada, omitir alguns parâmetros.

```
def abc(arg1, arg2=True, arg3=None):
  ...
# Quando so queremos passar um dos args por defeito devemos indicar o seu nome
abc(1, arg3=3)
```

2.8 Funções Recursivas

```
# devolve factorial de um numero n
def factorial(n):
  if n==0:
    return 1
  if n>0:
    return n*factorial(n-1)

# devolve o comprimento de uma lista
  def comprimento(lista):
    if lista==[]:
      return 0
    return 1+comprimento(lista[1:])
```

Esta última função, vai funcionar da seguinte forma:

```
comprimento([1, 2, 3])
=
1 + comprimento([2, 3])
=
1 + (1 + comprimento([3]))
=
1 + (1 + (1 + comprimento([])))
=
1 + (1 + (1 + 0))
=
3
```

```
# verifica se um elemento e membro de uma lista
def membro(x,lista):
  if l==[]:
    return False
  return (lista[0]==x) or membro(x,lista[1:])
```

```
# devolve uma lista com os elementos da lista de entrada por ordem inversa
def inverter(lista):
  if lista==[]:
    return []
  inv = inverter(lista[1:])
  inv[len(inv):] = [lista[0]]
  return inv
```

2.9 Expressões lambda

São **expressões cujo valor é uma função**. Funções que recebem expressões lambda como entrada e as retornam como saída chamam-se funções de ordem superior. Exemplo:

```
lambda x: x*x

# Pode ser atribuida a uma variavel
m = lambda x,y: math.sqrt(x**2 + y**2)
m(1, 2) # 2.23606797749979
```

Como qualquer objeto, uma expressão lambda pode ser passada como:

```
# Tendo uma func h que produz f(x)*X
def h(f, x): return f(x)*x

# Para usar:
h(lambda x: x+1, 7) # 56
```

Podem ser produzidas por outras funções:

```
def faz_incrementador(n):
  return lambda x: x+n

# Para usar:
f = faz_incrementador(1)
f(10) # 11
```

As expressões lambda são também conhecidas como expressões funcionais. As funções que recebem estas expressões como entrada e/ou produzem expressões lambda como saída chamam-se funções de ordem superior.

Nota importante: As expressões lambda só são úteis enquanto são simples. Uma função complexa merece ser escrita de forma clara numa definição (def) à parte.

2.10 Aplicar uma função a uma lista

Para tal podemos usar a função **map()**, predefinida em Python, retorna um iterador que pode ser convertido numa lista. Esta função é equivalente a:

```
def aplicar(funcao, lista):
  if lista==[]:
    return []
  return [funcao(lista[0])] + aplicar(funcao, lista[1:])
```

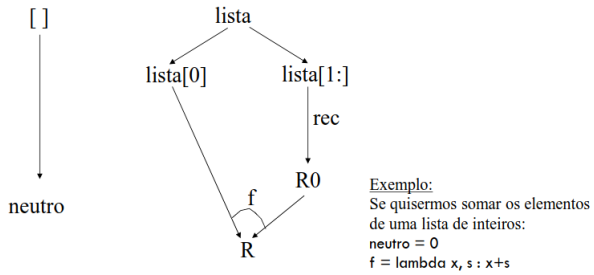
2.11 Filtrar uma lista

Para tal podemos usar a função **filter()**, predefinida em Python, retorna um iterador que pode ser convertido numa lista. Esta função é equivalente a:

```
def filtrar(funcao, lista):
  if lista==[]:
    return []
  if funcao(lista[0]):
    return [lista[0]] + filtrar(funcao, lista[1:])
  return filtrar(funcao, lista[1:])
```

2.12 Reduzir uma lista a um valor

Muitos dos procedimentos aplicados a listas consistem em combinar a cabeça da lista com o resultado da chamada recursiva sobre os restantes elementos, retornando um valor "neutro" pré-definido para a lista vazia. Em Python, podemos usar a função **reduce()** da biblioteca **functools**.




```
# Dada uma lista, uma func de combinacao e um elemento neutro
def reduzir(lista, fcomb, neutro):
  if lista==[]:
    return neutro
  return fcomb(lista[0], reduzir(lista[1:], fcomb, neutro))
```

2.13 Listas por compreensão (list comprehension)

São mecanismos compactos para processar os elementos de uma lista (alguns ou todos), podendo ser aplicadas não só a estas como também a tuplos ou cadeias de caracteres (str), o resultado é uma lista. Podem funcionar com a função pré-definida **map()** ou **filter()**.

$$< expr > \textbf{ for } < var > \textbf{ in } < seq > \textbf{ if } < condition >$$

```
[x**2 for x in [2, 3, 4]] # Output: [4, 9, 16]
map(lambda x: x**2, [2, 3, 4]) # Igual

[x for x in [2, 3, 4] if x%2==0] # Output: [2, 4]
filter(lambda x: x%2==0, [2, 3, 4]) # Igual
```

Dada uma lista de listas e uma função, podemos aplicar uma função a cada um dos elementos das listas, retornando a concatenação das listas resultantes.

```
[f(y) for x in lista for y in x]

# Exemplo:
lista = [[1, 2, 3], [4, 5, 6]]
[x+1 for x in lista for y in x] # [2, 3, 4, 5, 6, 7]
```

2.14 Classes

Tal como nas restantes linguagens Orientadas a Objetos (OO), em Python uma classe define um conjunto de objetos caracterizados por diversos atributos e métodos e organizam-se numa hierarquia que permite a herança.

```
# sintaxe
class <nome-classe>:
  <decl-1>
  ...
  <decl-n>
```

Nota: Tal como acontece com as variáveis normais, também os atributos das classes não são declarados. Os atributos são acessados através da notação "objeto.atributo". Podemos, em qualquer momento, criar um novo atributo para um objeto, bastando para isso atribuir-lhe um valor.

Existe ainda classes derivadas/herança, que herdam os métodos e atributos da classe mãe, sendo possível uma classe ter várias classes mães.

2.14.1 Métodos e atributos pré-definidos

Métodos:

- **__init__** - Construtor da classe;
- **__str__** - Conversão para cadeia de caracteres;
- **__repr__** - Representação em cadeia de caractéres que aparece na consola do interpretador;

Atributos:

- **__class__** - identifica a classe de um dado objecto

2.14.2 list de Python é uma classe

Tem os seguintes métodos:

- **append(x)** - Adiciona x ao fim da lista;
- **extend(L)** - Adiciona os elementos da lista L ao fim da lista;
- **insert(i, x)** - Adiciona x na posição i da lista;
- **remove(x)** - Remove a primeira ocorrência de x na lista;
- **index(x)** - Devolve o índice da primeira ocorrência de x na lista;
- **sort()** - Ordena a lista (modifica a lista);

3 Tópicos de Inteligência Artificial