Universidade de Brasília
Departamento de Ciência da Computação
Linguagens de Programação
Prof.: Marcelo Ladeira

Python
Programação Funcional

Linguagens de programação auxiliam a resolução de problemas de diversas maneiras:

•A maioria das linguagens são procedurais: programas são listas de instruções que dizem ao computador o que fazer com as entradas do programa. C, Pascal e todos os shell Unix são linguagens procedurais.

•Em linguagens declarativas escreve uma especificação que descreve o problema a ser resolvido e a implementação da linguagem como realizar o cálculo de forma eficiente. SQL é a linguagem declarativa que mais provavelmente estamos familiarizados. Uma consulta SQL descreve o conjunto de dados que desejamos recuperar, e o motor SQL decide se a varre tabelas ou usa índices, que subseções devem ser executadas primeiro, etc.

•Programas orientados a objetos manipulam coleções de objetos. Objetos têm estados internos e suportam métodos que recuperam ou modificam esses esteados internos de alguma forma. Smalltalk e Java são linguagens orientadas a objetos. C++ e Python são linguagens que suportam programação orientada a objetos, mas não forçam o uso dos recursos de orientação a objetos.

•Programação funcional decompõe o problema em um conjunto de funções. Idealmente funções apenas recebem entradas e produzem saídas e não existe qualquer estado interno que afeta a saída produzida por um dado de entrada. Linguagens funcionais incluem a Standard ML, OCaml, Haskell, dentre outras.

Os projetistas de algumas linguagens de computador escolhem enfatizar uma abordagem específica para a programação. Isso muitas vezes faz com que seja difícil escrever programas que usam

Alunos: **Apolônio Santiago**
Danilo Alves
Élida Borges
Thiago Xavier

uma abordagem diferente. Outras linguagens são multi-paradigmas que suportam várias abordagens diferentes. Lisp, C++ e Python são multi-paradigma; você pode escrever programas ou bibliotecas que são em grande parte procedural, orientada a objeto, ou funcional em todas essas linguagens. Em um grande programa, seções diferentes podem ser escritas usando diferentes abordagens; o GUI pode ser orientada a objetos, enquanto a lógica de processamento é de natureza processual ou funcional, por exemplo.

Em um programa funcional, a entrada flui através de um conjunto de funções. Cada função opera em sua entrada e produz alguma saída. O estilo funcional desencoraja o uso de funções que modificam o estado interno ou fazem outras alterações que não são visíveis no valor de retorno da função. Funções que não causam efeitos colaterais são chamadas puramente funcionais. Evitar efeitos colaterais significa não usar estruturas de dados que atualizam como um programa é executado; os dados de saída de cada função dependem apenas dos dados de entrada.

Algumas linguagens são muito estritas quanto a pureza e não possuem sequer instruções de atribuição, como a = 3 ou c = a + b, mas são difíceis de se evitar todos os efeitos colaterais. Imprimir na tela ou gravar um arquivo em disco são efeitos colaterais, por exemplo. Por exemplo, em Python uma declaração de impressão ou uma time.sleep(1) não retornam um valor útil; eles são chamados apenas por seus efeitos que são enviar algum texto para a tela ou pausar a execução por um segundo.

Programas Python escritos em estilo funcional geralmente não vão ao extremo ao evitar toda E/S ou todas as atribuições; em vez disso, eles fornecem uma interface de aparência funcional, mas internamente usam recursos não-funcionais. Por exemplo, a implementação de uma função ainda usará atribuições para variáveis locais, mas não modificam variáveis globais ou tem outros comportamentos.

Alunos: **Apolônio Santiago**
Danilo Alves
Élida Borges
Thiago Xavier

Programação funcional pode ser considerada o oposto da programação orientada a objetos. Objetos são pequenas cápsulas contendo algum estado interno, juntamente com um conjunto de chamadas de métodos que permitem modificar este estado, e os programas consistem em fazer o conjunto correto de alterações de estado. A programação funcional evita mudanças de estado tanto quanto possível e trabalha com dados que fluem entre as funções. Em Python você pode combinar as duas abordagens escrevendo funções que recebem e retornar instâncias que representam objetos em sua aplicação (mensagens de correio eletrônico, transações, etc).

Design funcional pode parecer estranho de se trabalhar. Por que devemos evitar objetos e efeitos colaterais? Há vantagens teóricas e práticas para o estilo funcional:

- Provabilidade formal (Formal provability).
- Modularidade (Modularity).
- Componibilidade (Composability).
- Facilidade para depuração e teste.

Provabilidade formal

A vantagem teórica é que é mais fácil provar matematicamente que um programa funcional está correto.

Durante muito tempo, os pesquisadores têm se interessado em encontrar maneiras de provar matematicamente programas corretos. Isso é diferente de testar um programa em numerosas entradas e concluindo que sua saída é geralmente correta, ou lendo o código-fonte de um programa e concluir que o código é certo; o objetivo é provar rigorosa que um programa produz o resultado correto para todas as entradas possíveis.

A técnica usada para provar que programas estão correto é escrever invariantes, as propriedades dos dados de entrada e de variáveis do

Alunos: **Apolônio Santiago**
Danilo Alves
Élida Borges
Thiago Xavier

programa que sempre são verdadeiras. Para cada linha de código, você então mostra que se as invariantes X e Y são verdadeiras antes da linha ser executada, as outras invariantes X' e Y' são verdadeiras após a linha ser executada. Isso continua até chegar ao final do programa, em que ponto as invariantes devem coincidir com as condições desejadas na saída do programa.

Evitar atibuições na programação funcional surgiu porque as atribuições são difíceis de lidar com esta técnica; atribuições podem quebrar invariantes que eram verdadeiras antes da atribuição sem gerar nenhuma nova invariante que possa ser propagadas a diante.

Infelizmente, provar programas que programas estão corretos é, em grande parte, impraticável e não é relevante para o software Python. Mesmo programas triviais exigem provas que são várias páginas; a prova de correção para um programa moderadamente complicado seria enorme, e poucos ou nenhum dos programas que você usa diariamente (o interpretador Python, o analisador XML, seu navegador) podem ser provados corretamente. Mesmo que você tenha escrito ou gerado uma prova, não seria, então, a questão de verificar a prova; talvez haja um erro na mesma, e você acreditou que tenha provado o programa correto.

Modularidade

O maior benefício da linguagem funcional é que ela força você a quebrar o problema em pedaços menores. O resultado são programas mais modulares. É mais fácil especificar e escrever uma pequena função que faz uma coisa que uma grande função que executa uma transformação complexa. Pequenas funções também são mais fáceis de ler e verificar se há erros.

Facilidade para depuração e teste

Testar e depurar um programa de estilo funcional é mais fácil.

Alunos: **Apolônio Santiago**
Danilo Alves
Élida Borges
Thiago Xavier

A depuração é simplificada porque as funções são geralmente pequenas e claramente especificadas. Quando um programa não funciona, cada função é um ponto de interface, onde você pode verificar se os dados estão corretos. Você pode olhar para as entradas e saídas intermediárias para isolar rapidamente a função que é responsável por um bug.

O teste é mais fácil, porque cada função é um assunto potencial para um teste de unidade. Funções não dependem de estados do sistema que precisam ser replicados antes de executar um teste; em vez disso você só tem que sintetizar a entrada correta e, em seguida, verificar se a saída corresponde às expectativas.

Componibilidade

Como você trabalha em um programa em estilo funcional, você vai escrever uma série de funções com diferentes entradas e saídas. Algumas dessas funções serão inevitavelmente especializadas para uma aplicação em particular, mas outros serão úteis em uma ampla variedade de programas. Por exemplo, uma função que recebe um caminho de diretório e retorna todos os arquivos XML no diretório, ou uma função que recebe um nome de arquivo e retorna o seu conteúdo, pode ser aplicado a muitas situações diferentes.

Com o tempo você vai formar uma biblioteca pessoal de utilidades. Muitas vezes você vai montar novos programas organizando as funções existentes em uma nova configuração e escrever algumas funções especializadas para a tarefa atual.

Iteradores

Vamos começar por olhar para um recurso de linguagem Python que é uma base importante para a escrita de programas de estilo funcional: iteradores.

Uma iteração é um objecto que representa um fluxo de dados; este

Alunos: **Apolônio Santiago**
Danilo Alves
Élida Borges
Thiago Xavier

objeto retorna os dados de um elemento de cada vez. Um iterador Python deve suportar um método chamado next() que não recebe argumentos e sempre retorna o próximo elemento do fluxo. Se não há mais elementos no fluxo, next() deve levantar a exceção StopIteration. Iteradores não tem que ser finito, embora; é perfeitamente razoável para escrever um iterador que produz um fluxo infinito de dados.

A função built-in iter () recebe um objeto arbitrário e tenta retornar um iterador que retornará conteúdos ou elementos do objeto, elevando TypeError se o objeto não suporta iteração. Vários dos built-in tipos de dados suportar iteração do Python, sendo a mais comum listas e dicionários. Um objeto é chamado de um objeto iterable se você pode obter um iterador para ele.

Você pode experimentar a interface de iteração manualmente:

```
>>> L = [1,2,3]
>>> it = iter(L)
>>> print it
<...iterator object at ...>
>>> it.next()
1
>>> it.next()
2
>>> it.next()
3
>>> it.next()
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
StopIteration
>>>
```

Python espera objetos iteráveis em vários contextos diferentes, sendo a mais importante a de declaração. Na declaração de X em Y, Y deve ser um iterador ou algum objeto para o qual iter () pode criar uma iteração. Estas duas afirmações são equivalentes:

```
for i in iter(obj):
    print i
```

Alunos: **Apolônio Santiago**
Danilo Alves
Élida Borges
Thiago Xavier

```
for i in obj:
    print i
```

Iteradores podem ser materializados como listas ou tuplas usando as funções construtoras list() ou tupla ():

```
>>> L = [1,2,3]
>>> iterator = iter(L)
>>> t = tuple(iterator)
>>> t
(1, 2, 3)
```

Sequência de desempacotamento também suportam iterators: se você sabe os N elementos retornados pelo iterador, basta descompactá-los em uma tupla de N elementos:

```
>>> L = [1,2,3]
>>> iterator = iter(L)
>>> a,b,c = iterator
>>> a,b,c
(1, 2, 3)
```

Funções internas, como max() e min() podem ter um único argumento iterator e irá retornar o maior e menor elemento respectivamente. Os operadores "in" e "not in" também suportam iterators: X em iterator é verdadeiro se X é encontrado no fluxo retornado pelo iterador. Você irá percorre por problemas óbvios se o iterador é infinito; max(), min() nunca retornarão, e se o elemento X nunca aparece no fluxo, os operadores "in" e "not in" operadores também não retornam.

Note que você só pode ir para a frente em uma iteração; não há nenhuma maneira de obter o elemento anterior, redefinir o iterator, ou fazer uma cópia do mesmo. Objetos Iterator pode opcionalmente fornecer esses recursos adicionais, mas o protocolo iterador só especifica o método next (). Funções podem, portanto, consumir toda a produção do iterator, e se você precisa fazer algo diferente com o mesmo fluxo, você vai ter que criar uma nova iteração.

Alunos: **Apolônio Santiago**
Danilo Alves
Élida Borges
Thiago Xavier

Tipos de Dados que suportam Iteradores

Nós já vimos como as listas e tuplas suportam iterators. Na verdade, qualquer tipo de seqüência Python, tais como strings, suportam automaticamente a criação de um iterador.

```
>>> m = {'Jan': 1, 'Feb': 2, 'Mar': 3, 'Apr': 4, 'May': 5, 'Jun': 6,
...      'Jul': 7, 'Aug': 8, 'Sep': 9, 'Oct': 10, 'Nov': 11, 'Dec': 12}
>>> for key in m:
...     print key, m[key]
Mar 3
Feb 2
Aug 8
Sep 9
Apr 4
Jun 6
Jul 7
Jan 1
May 5
Nov 11
Dec 12
Oct 10
```

Note que a ordem é essencialmente aleatória, porque é baseado na ordenação de hash dos objetos no dicionário.

Aplicando iter() a um dicionário sempre laços sobre as teclas, mas dicionários têm métodos que retornam outros iteradores. Se você quiser interagir sobre chaves, valores, ou pares de chave / valor, você pode chamar explicitamente os iterkeys(), itervalues () ou iteritems () métodos para obter um iterador apropriado.

O construtor dict () pode aceitar um iterador que retorna um fluxo finito de (chave, valor) tuplas:

```
>>> L = [('Italy', 'Rome'), ('France', 'Paris'), ('US', 'Washington DC')]
>>> dict(iter(L))
{'Italy': 'Rome', 'US': 'Washington DC', 'France': 'Paris'}
```

Arquivos também suportam iteração pela chamada ao método readline() até que não haja mais linhas no arquivo. Isso significa que

Alunos: **Apolônio Santiago**
Danilo Alves
Élida Borges
Thiago Xavier

você pode ler cada linha de um arquivo como este:

```
for line in file:
    # do something for each line
    ...
```

Conjuntos podem pegar seus conteúdos a partir de um iterator podendo interagir sobre seus elementos:

```
S = set((2, 3, 5, 7, 11, 13))
for i in S:
    print i
```

Gerador de expressões e definição lista

Duas operações comuns sobre a saída de uma iteração são: 1) realizando alguma operação para cada elemento, 2) a seleção de um subconjunto de elementos que atendam algumas condições. Por exemplo, dada uma lista de strings, você pode querer retirar espaços em branco de cada linha ou extrair todas as cadeias que contém uma determinada substring.

Definições de lista e gerador de expressões (forma curta: "listcomps" e "genexps") são uma notação concisa para tais operações, emprestado da linguagem de programação funcional Haskell (http://www.haskell.org/). Você pode tirar todos os espaços de um fluxo de strings com o seguinte código:

```
line_list = ['  line 1\n', 'line 2  \n', ...]

# Generator expression -- returns iterator
stripped_iter = (line.strip() for line in line_list)

# List comprehension -- returns list
stripped_list = [line.strip() for line in line_list]
```

Você pode selecionar apenas alguns elementos, adicionando uma condição "se":

```
stripped_list = [line.strip() for line in line_list
```

Alunos: **Apolônio Santiago**
Danilo Alves
Élida Borges
Thiago Xavier

```
                if line != ""]
```

Com uma compreensão de lista, você recebe de volta uma lista Python; stripped_list é uma lista contendo as linhas resultantes, não um iterador. Gerador de expressões retornar um iterador que calcula os valores conforme necessário, não precisando materializar todos os valores de uma só vez. Isso significa que compreensões lista não são úteis se você estiver trabalhando com iteradores que retornam um fluxo infinito ou uma quantidade muito grande de dados. Gerador de expressões são preferíveis nestas situações.

Gerador de expressões são cercados por parênteses ("()") e definições de listas são cercados por colchetes ("[]"). Gerador de expressões têm a forma:

```
( expression for expr in sequence1
             if condition1
             for expr2 in sequence2
             if condition2
             for expr3 in sequence3 ...
             if condition3
             for exprN in sequenceN
             if conditionN )
```

Novamente, para definição de lista apenas os colchetes externos são diferentes (entre colchetes em vez de parênteses).

Os elementos de saída gerados serão os valores sucessivos da expressão. A cláusula if é opcional; se presente, a expressão é apenas avaliada e adicionada ao resultado quando a condição é verdadeira.

Gerador de expressões sempre tem que ser escrito entre parênteses, mas os parênteses também definem uma chamada de função. Se você deseja criar um iterador que será imediatamente transferido para uma função, você pode escrever:

```
obj_total = sum(obj.count for obj in list_all_objects())
```

Alunos: **Apolônio Santiago**
Danilo Alves
Élida Borges
Thiago Xavier

As cláusulas for...in contêm as sequências a serem iteradas. As sequências não têm que ser do mesmo comprimento, pois elas são iteradas da esquerda para a direita, não em paralelo. Para cada elemento na Sequencia1, Sequence2 é repetido ao longo desde o início. Sequencia3 é então posto em loop para cada par resultante de elementos de Sequencia1 e Sequencia2.

Dito de outra forma, uma compreensão de lista ou gerador de expressão é equivalente ao seguinte código Python:

```
for expr1 in sequence1:
    if not (condition1):
        continue   # Skip this element
    for expr2 in sequence2:
        if not (condition2):
            continue    # Skip this element
        ...
        for exprN in sequenceN:
             if not (conditionN):
                 continue   # Skip this element

             # Output the value of
             # the expression.
```

Isto significa que, quando há múltiplas cláusulas for...in, mas não cláusulas if, o comprimento da saída resultante será igual ao produto dos comprimentos de todas as sequências. Se você tem duas listas de comprimento 3, a lista de saída é contém 9 elementos:

```
>>> seq1 = 'abc'
>>> seq2 = (1,2,3)
>>> [(x,y) for x in seq1 for y in seq2]
[('a', 1), ('a', 2), ('a', 3),
 ('b', 1), ('b', 2), ('b', 3),
 ('c', 1), ('c', 2), ('c', 3)]
```

Para evitar a introdução de uma ambiguidade na gramática do Python, se a expressão é a criação de uma tupla, ela deve ser

Alunos: **Apolônio Santiago**
Danilo Alves
Élida Borges
Thiago Xavier

cercada com parênteses. A definição da primeira lista abaixo é um erro de sintaxe, enquanto o segunda está correto:

```
# Syntax error
[ x,y for x in seq1 for y in seq2]
# Correct
[ (x,y) for x in seq1 for y in seq2]
```

Geradores

Geradores são uma classe especial de funções que simplificam a tarefa de escrever iteradores. Funções normais calculam um valor e o retornam, mas geradores retorna um iterador que retorna um fluxo de valores.

Você sem dúvida está familiarizado com a forma como as chamadas de função regulares trabalham em Python ou C. Quando você chama uma função, ela recebe um namespace privado, onde suas variáveis locais são criadas. Quando a função atinge uma instrução de retorno, as variáveis locais são destruídas e o valor é retornado para o chamador. A chamada posterior para a mesma função cria um novo namespace privado e um novo conjunto de variáveis locais. Mas, e se as variáveis locais não foram jogados fora à saída de uma função? E se você pudesse depois retomar a função de onde parou? Isto é o que os geradores fornecem; eles podem ser pensados como funções retomáveis.

Aqui está o exemplo mais simples de uma função geradora:

```
def generate_ints(N):
    for i in range(N):
        yield i
```

Qualquer função que contém uma palavra-chave yield é uma função geradora; este é detectado pelo compilador bytecode do Python que compila a função especialmente como resultado.

Quando você chamar uma função geradora, ela não retorna um

único valor; em vez disso, retorna um objeto gerador que suporta o protocolo iterador. Na execução da expressão yield, o gerador gera o valor de i, semelhante a uma instrução de retorno. A grande diferença entre yield e uma instrução de retorno é que ao atingir um yield o estado de execução do gerador está suspenso e variáveis locais são preservadas. Na próxima chamada para o método .next() do gerador, a função vai retomar a execução.

Aqui está uma amostra de uso do gerador generate_ints():

```
>>> gen = generate_ints(3)
>>> gen
<generator object generate_ints at ...>
>>> gen.next()
0
>>> gen.next()
1
>>> gen.next()
2
>>> gen.next()
Traceback (most recent call last):
  File "stdin", line 1, in ?
  File "stdin", line 2, in generate_ints
StopIteration
```

Pode-se igualmente escrever for i in generate_ints(5), ou a, b, c = generate_ints (3).

Dentro de uma função geradora, a instrução de retorno só pode ser usado sem um valor, e sinaliza o fim da processão de valores; depois de executar um retorno o gerador não pode retornar quaisquer valores. retornar com um valor, como o return 5, é um erro de sintaxe dentro de uma função geradora. O final dos resultados do gerador pode também ser indicado manualmente pelo lançamento de uma StopIteration, ou simplesmente deixar o fluxo de execução sair da função.

Você pode conseguir o efeito de geradores manualmente, escrevendo sua própria classe e armazenar todas as variáveis locais do gerador como variáveis de instância. Por exemplo, o retorno de

uma lista de números inteiros pode ser feito através da fixação de self.count como 0, e tendo o método next() que incrementa o self.count e o retorna. No entanto, para um gerador moderadamente complicado, escrever uma classe correspondente pode ser muito mais confuso.

O conjunto de testes incluídos na biblioteca Python, test_generators.py, contém uma série de exemplos mais interessantes. Aqui está um gerador que implementa uma travessia no fim de uma árvore usando geradores de forma recursiva.

```
# A recursive generator that generates Tree leaves in in-order.
def inorder(t):
    if t:
        for x in inorder(t.left):
            yield x

        yield t.label

        for x in inorder(t.right):
            yield x
```

Dois outros exemplos produzem soluções test_generators.py para o problema N-Queens (colocar N rainhas em um tabuleiro de xadrez NxN de forma que nenhuma rainha ameace a outra) e Knight's Tour (encontrar uma rota que leva um cavalo a cada quadrado de um tabuleiro de xadrez NxN sem visitar qualquer quadrado duas vezes).

Passando valores para o gerador

Em Python 2.4 e anteriores, os geradores só produzem uma saída. Uma vez que o código de um gerador foi chamado para criar um iterador, não havia nenhuma maneira de passar qualquer informação nova para a função quando a sua execução é retomada. Você poderia contornar isso tornando o gerador em uma variável global ou passando em algum objeto mutável que os chamadores então modificam, mas essas abordagens são desorganizados.

Em Python 2.5, há uma maneira simples de passar valores em um

gerador. yield tornou-se uma expressão, retornando um valor que pode ser atribuído a uma variável ou não operado em:

```
val = (yield i)
```

Eu recomendo que você sempre coloque a expressão yield entre parênteses quando você está fazendo algo com o valor retornado, como no exemplo acima. Os parênteses nem sempre são necessários, mas é mais fácil sempre adicioná-los em vez de ter que lembrar de quando eles são necessários.

(PEP 342 explica as regras exatas, que a expressão yield deve sempre estar entre parênteses, exceto quando ocorre a expressão de nível superior no lado direito de uma atribuição. Isto significa que você pode escrever val = yield i, mas têm usar parênteses quando há uma operação, como em val = (i yield) + 12)

Os valores são enviados em um gerador chamando seu método send(valor). Este método retoma código do gerador e a expressão yield devolve o valor especificado. Se o método normal next() é chamado, o rendimento retorna None.

Aqui está um simples contador que incrementa por 1 e permite alterar o valor do contador interno.

```
def counter (maximum):
    i = 0
    while i < maximum:
        val = (yield i)
        # If value provided, change counter
        if val is not None:
            i = val
        else:
            i += 1
```

E aqui está um exemplo que altera o contador

```
>>> it = counter(10)
```

Alunos: **Apolônio Santiago**
Danilo Alves
Élida Borges
Thiago Xavier

```
>>> print it.next()
0
>>> print it.next()
1
>>> print it.send(8)
8
>>> print it.next()
9
>>> print it.next()
Traceback (most recent call last):
  File "t.py", line 15, in ?
    print it.next()
```

Alunos: **Apolônio Santiago**
Danilo Alves
Élida Borges
Thiago Xavier