GRADUAÇÃO PBL APRENDIZAGEM BASEADA EM PROBLEMAS

FGV EESP
NÚCLEO DE APOIO PEDAGÓGICO e PESQUISA EM EDUCAÇÃO (NAPPE)

Disciplina: Introdução à Programação

Professor-líder:
Bruno Tebaldi — Email: bruno.barbosa@fgv.br

Professores-supervisores:
Bruna Alvarez — Email: bruna.alvarez@fgv.br
Diego Pinheiro — Email: diego.pinheiro@fgv.br
Giovanni Di Pietra — Email: giovanni.pietra@fgv.br
Kelly Santos — Email: kelly.santos@fgv.br
Rhamon Peixoto — Email: rhamon.peixoto@fgv.br

1º SEMESTRE DE 2024

PROGRAMAÇÃO

Data	Encontro	Problemas Pós	Problemas Pré	Tópico (pós)	Bibliografia (pré)
DD/MM	Tutorial 01	-	1	-	Mapograf – O Guia
	Tutorial 02	1	2	Programando uma Road Trip	(PCC, Ch1)
	Tutorial 03	2	3	Organizando o planejamento	https://t.ly/hdkhn
	Tutorial 04	3	4	Ambiente de Trabalho	(PCC, Ch1), HFP (Intro)
	Tutorial 05	4	5	Sistematizando a Informação	(PCC, Ch2, 16-19)
	Tutorial 06	5	6	Informação na forma de texto	(PCC, Ch2, 19-25)
	Tutorial 07	6	7	Informação na forma de números	(PCC, Ch2, 26-29)
	Tutorial 08	7	8	Agrupando Informações	(PCC, Ch3, 34-47)
	Tutorial 09	8	9	Trabalhando com Informações Agrupadas	(PCC, Ch4, 50-65)
	Tutorial 10	9	10	Premissas Básicas	(PCC, Ch4, 65-70)
	Tutorial 11	10	11	E se...?	(PCC, Ch5, 71-89)
	Tutorial 12	11	12	Deduzindo Informações	
	Tutorial 13	12	13	Um jeito eficiente de relacionar informações	(PCC, Ch6, 91-105)
	Tutorial 14	13	14	Matrioskas	(PCC, Ch6, 105-112)
	Tutorial 15	14	15	Enquanto esperamos...	(PCC, Ch7, 114-123)
	Tutorial 16	15	16	Adquirindo e movendo informações	(PCC, Ch7, 124-127)
	Tutorial 17	16	17	Máquinas de transformar e criar informações	(PCC, Ch8, 130-140)
	Tutorial 18	17	18	Máquinas de transformar e criar informações II	(PCC, Ch8, 140-149)
	Tutorial 19	18	19	Containers de Informação	(PCC, Ch8, 149-154)
	Tutorial 20	19	20	Coisas diferentes nos detalhes, mas iguais em essência	(PCC, Ch9, 158-166)
	Tutorial 21	20	21	Transmissão genética em programação	(PCC, Ch9, 167-173)
	Tutorial 22	21	22	A mãe de todas as bibliotecas	(PCC, Ch9, 173-179)
	Tutorial 23	22	23	Arquivos	(PCC, Ch9, 179-181)
	Tutorial 24	23	24	Imprevistos acontecem...	(PCC, Ch10, 183-191)
	Tutorial 25	24	25	Armazenando informações estruturadas	(PCC, Ch10, 192-200)
	Tutorial 26	25	26	Antecipando problemas	(PCC, Ch10, 201-204)
	Tutorial 27	26	27	Ajudando a financiar a viagem	(PCC, Ch11)
	Tutorial 28	27	-		
Prova Final (PF)					

BIBLIOGRAFIA INICIAL

Primária: Python Crash Course (PCC), Eric Matthes; Third Edition, no starch press

Para apoio e extensão dos conceitos: **Head** First Python (HFP), Paul Berry; 2023 Edition, O'Reilly Media

1. Programando uma Road Trip

As aulas acabaram, você já fez a úlima prova, e está comemorando o final do ano junto com seus amigos no bar. Todos se dão conta que aquelas podem ser as últimas férias escolares longas antes que comecem os estágios, etc. Alguém joga a ideia do grupo fazer uma viagem épica para conhecer vários lugares interessantes com calma, sem filas e aeroportos.

Fonte: dreamstime.com.

A ideia é um sucesso imediato, e vocês começar a pensar na aventura de forma concreta.

Bibliografia inicial:

Ou qualquer bom guia de viagens...

PRÉ-DISCUSSÃO

Objetivos de aprendizagem:

Criar um primeiro esquema de anotações e ideias para fazer o planejamento da viagem. Esses elementos serão utilizados para aplicações dos conceitos de programação vistos ao longo do curso.

Ao final da pré:

Ter uma primeira visão do conjunto de informações que serão utilizadas no planejamento da viagem (itinerários primários e secundários, tempo, distâncias, custos, o que levar,...)

PÓS-DISCUSSÃO

Elementos esperados na pós discussão:

O objetivo é estimular iniciativa e criatividade. A natureza completamente intuitiva e atraente do assunto deve ensejar uma troca de ideias animada. Nesse ponto, não deve haver restrições quanto aos dois aspectos fundamentais envolvidos:
Diferentes roteiros e destinos de viagens considerados (pode vir naturalmente e também ser complementado por consulta a algum guia de viagens, como sugerido na bibliografia),
O mecanismo específico para registro das ideias. Nesse ponto, o mais importante é o envolvimento dos alunos com o tipo de informação ao qual ele será capacitado para armazenar e manipular através de programação ao longo do curso. Cursos tradicionais de programação contendo exemplos abstratos de listas de objetos, etc não criam esse elo entre a teoria e o interesse de aplicação dos métodos computacionais. O esforço de aprendizado a partir desse tutorial será muito estimulado pelo envolvimento dos alunos com as informações com as quais serão aos poucos ensinados a manipular por meio de uma linguagem de programação.

2. Organizando o planejamento

Fonte: dreamstime.com.

Todos estão animados com a conversa inicial sobre a viagem, mas agora está claro que vários aspectos importantes devem ser considerados com cuidado: duração, distâncias, roteiros, custos, o que levar, etc. A coisa fica ainda mais complicada considerando os objetivos e restrições de todos no grupo. Alguém dá a ideia de fazer uma planilha para as contas necessárias, porém outro membro lembra que alunos dos anos mais avançados do curso comentaram que planilhas, apesar de fáceis, são limitadas em suas funções e pouco flexíveis para lidar com problemas envolvendo modelos das várias áreas de teoria econômica. O grupo decide que, já que alguma linguagem de programação será usada nos cursos de estatística e econometria mais à frente, eles podem aproveitar a oportunidade para aprender essa linguagem. Uma busca por um bom recurso inicial para uma visão geral e comparativa entre as linguagens de programação mais populares atualmente retorna o endereço.

Bibliografia inicial:
https://graduate.northeastern.edu/resources/most-popular-programming-languages/

PRÉ-DISCUSSÃO

Objetivos de aprendizagem:

- Conhecer as principais linguagens de programação utilizadas atualmente
- Capacitação para comparar diferentes linguagens de programação em relação a
  - Disponibilidade
  - Facilidade de Aprendizado
  - Aplicações

Instruções para os tutores: existem muitos aspectos relevantes para comparar linguagens de programação, e isso pode gerar informação excessiva e desnecessária para os alunos nesse ponto. O tutor deve estimular entre esses aspectos a consideração daqueles que são, ainda de forma ainda intuitiva, mais relevantes para o tipo de aplicações que ele imagina ao longo da graduação em economia. Pelo menos de forma inicial, aspectos como facilidade de aprendizado, potencial de aplicação em diferentes problemas, velocidade de execução de códigos e disponibilidade de material didático (inclusive sem custo), devem aparecer naturalmente na discussão, e caso não apareçam, podem ser introduzidos pelo tutor.

Ao final da pré:

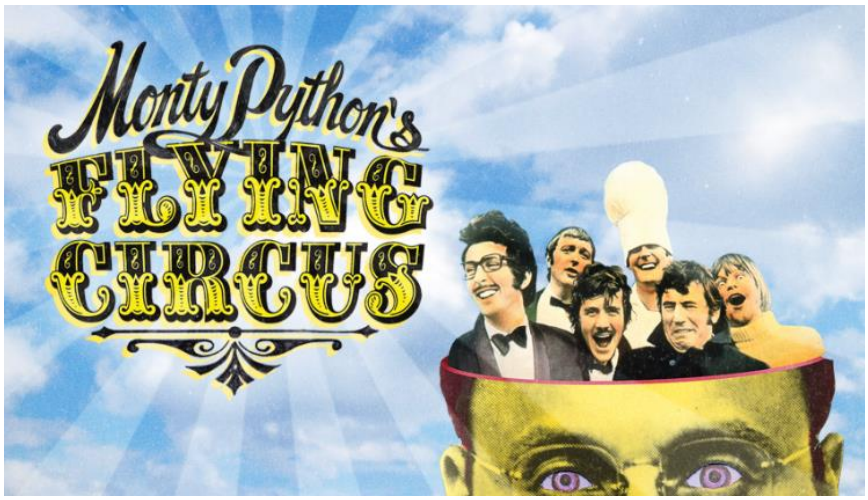
- Consultar bibliografia para encontrar uma alternativa computacional ao Excel.

PÓS-DISCUSSÃO

Elementos esperados na pós discussão:

- Quais são as principais linguagens de programação utilizadas atualmente
- Como essas linguagens se comparam em termos de facilidade de aprendizado, potencial de aplicações, disponibilidade de pacotes complementares.
- Vantagens de uma linguagem de programação em relação a planilhas eletrônicas, tais como flexibilidade para resolução de diferentes tipos de problemas, e, mesmo nos casos onde planilhas consigam produzir os resultados desejados, ganhos de velocidade de execução dos códigos, menor restrições sobre o tamanho das bases de dados que podem ser armazenadas e conveniência para compartilhamento de código e dados com terceiros.
- Argumentos para a escolha do Python:
    - Facilidade de aprendizado, inclusive através da leitura e compreensão de códigos escritos por terceiros
    - Pode ser instalado e utilizado de forma totalmente intercambiável entre as principais plataformas computacionais de hoje em dia (Windows, MacOS e Linux)
    - Velocidade de execução
    - Disponibilidade de aplicativos desenvolvidos por terceiros, disponibilizados em “pacotes de código” (que serão vistos em um tutorial específico à frente)
    - Flexibilidade de interação com informações externas, principalmente as disponíveis em redes, tais como bancos de dados dos mais diferentes formatos e conteúdo de páginas de sites na Internet.

3. Ambiente de Trabalho



Fonte: facebook.com.

As pesquisas anteriores mostraram que existem dezenas de linguagens de programação, com várias diferenças entre elas em termos de sintaxe, complexidade, velocidade de execução, etc. Alunos dos cursos de estatística, econometria e ciência de dados contaram que a enorme maioria dos cursos, livros, e exemplos na internet usam ou o R ou o Python como linguagens de aplicação e programação. Como as duas linguagens são bem parecidas em seus princípios gerais, e tanto o R é capaz de rodar códigos gerados no Python quanto vice-versa, o grupo decide começar tirando vantagem da simplicidade do Python e começar por ela.

Ao diferentes plataformas computacionais pessoais utilizadas atualmente permitem a interação com o usuário completamente por meio de interfaces gráficas utilizando ícones e menus. No entanto, todas elas (Windows, MacOS e Linux) possuem “prompts de comandos”, que ná prática são instruções e comandos para realizar as mais diferentes tarefas, exclusivamente por meio de linhas de código. O Python não faz parte das instalações padrões dessas plataformas, e portanto para ser utilizado precisa primeiramente ser instalado no computador de quem quiser programar com ele. Uma primeira leitura rápida de (PCC, Ch1) e (HFP, Intro) indica como instalar não só a linguagem de programação Python em cada plataforma, mas também aplicativos que auxiliam na tarefa de edição e execução de códigos, proporcionando ganhos significativos de produtividade. Essa mesma leitura torna claro que a entrada para a jornada de aprendizado de programação passa inevitavelmente por esse primeiro passo de instalação da linguagem escolhida, bem como de suas ferramentas de aumento de produtividade, e a execução de um primeiro exemplo bem simples de programação, com o único objetivo de testar se tudo parece estar funcionando bem antes da utilização para o aprendizado da linguagem propriamente dito.

Bibliografia inicial:
(PCC, Ch1), HFP (Intro)

PRÉ-DISCUSSÃO

Objetivos de aprendizagem:

- Criar um ambiente computacional para rodar os códigos em Pyhton
- Rodar um exemplo simples de código (“Hello World”)

Ao final da pré:

- Entender que é necessário criar um ambiente para rodar os códigos, e analisar as opções disponíveis na bibliografia fornecida.

PÓS-DISCUSSÃO

Elementos esperados na pós discussão:

- Instalação do ambiente computacional:
  - Python
  - VSCode
  - Jupyter
- Rodar um código simples (“Hello World”) nas opções:
  - Janela de Comando
  - VSCode
  - Jupyter notebook

4. Sistematizando a informação

Fonte: elaborado pelo autor com uso de imagine.art.

As anotações dos membros do grupo incluem vários elementos que devem ser considerados conjuntamente para ajudar nas escolhas que serão feitas. Essas informações podem ser de tipos diferentes, e isso deve ser levado em consideração no código que será gerado para lidar com elas. A partir daí, a construção do código deve partir da representação e armazenamento dessas informações dentro do contexto da construção de um programa.

Bibliografia inicial:
(PCC, Ch2, 16-19)

PRÉ-DISCUSSÃO

Objetivos de aprendizagem:

- Quais são os diferentes tipos de dados com os quais se pode trabalhar em códigos de Python.
- Como usar variáveis para representar dados em programas, e já especificamente como esses dados devem refletir as informações que os alunos julgaram interessantes e essenciais até esse ponto.

Ao final da pré:

- Os alunos devem entender que as anotações relevantes para o problema do planejamento da viagem, até aqui estão feitas usando papel e lápis, precisam ser transportadas para o código que permitirá a manipulação dessas informações.

PÓS-DISCUSSÃO

Elementos esperados na pós discussão:

- Variáveis são formas de representação de informações utilizadas em um código
- Diferentes tipos de informação são tratados de maneiras específicas na linguagem Python
- Os alunos já devem, dentro do editor de programação instalado, iniciar um arquivo contendo um programa. Por enquanto, o código desse arquivo inclui apenas nomes e conteúdo de variáveis que representam elementos importantes para o planejamento da viagem, e uma função para apresentar esse conteúdo na tela como resultado da execução do programa. Por exemplo:

```
nome_cidade = "Salvador"
print(nome_cidade
```

5. Informações na forma de texto

Fonte: elaborado pelo autor com uso de imagine.art.

Vários elementos nas anotações do planejamento da viagem são representados por informações não numéricas, como nomes de cidades que se pretende visitar, itens de vestimentas que devem ser levados, etc. O programa a ser gerado para auxiliar no planejamento deve ser capaz de representar e manipular essas informações.

Bibliografia inicial:
(PCC, Ch2, 19-25)

PRÉ-DISCUSSÃO

Objetivos de aprendizagem:

No resultado do tutorial anterior, o aluno, seguindo os exemplos da bibliografia, criou uma ou mais variáveis representando informação na forma de texto. Agora ele deve entender que essa é uma entre outras formas de criar variáveis em Python, e que essa forma específica é importante não só pelo tipo de informação que ela é capaz de representar, mas como o Python permite diferentes manipulações contendo textos. Nesse contexto, os objetivos desse tutorial são:

- Entender como informação na forma de texto é armazenada em linguagens de programação
- Identificar *strings* como o primeiro dos data types definidos no Python
- Aprender que cada variável contendo informação é um objeto da classe da informação correspondente

- Classes de tipos de dados contém métodos e atributos que permiter manipular a informação armazenada em cada tipo específico.

Ao final da pré:

- Temos que entender como variáveis contendo informações na forma de texto serão utilizadadas no programa de auxílio no planejamento da viagem.

PÓS-DISCUSSÃO

Elementos esperados na pós discussão:

- Informação na forma de texto é representada por variáveis tipo *string*, que armazenam sequências de caracteres
- *Strings* podem ser manipuladas por meio dos métodos e atributos da classe à qual pertence esse tipo de variável.

Exemplos de códigos que poder ser gerados a partir das informações criadas pelos alunos e sua manipulação a partir do material aprendido na leitura da bibliografia:

```
s = "Salvador"

print(s.upper())  # Output: SALVADOR
print(s.lower())  # Output: salvador
print(s.startswith("Sal"))  # Output: True
print(s.endswith("dor"))  # Output: True
print(s.replace("Sal", "Bal"))  # Output: Baldor
print(s.split("a"))  # Output: ['S', 'lv', 'dor']
print(s.count("a"))  # Output: 2
print(s.find("a"))  # Output: 1
```

6. Informação na forma de números

fonte: elaborado pelo autor com uso de imagine.art.

Vários elementos nas anotações do planejamento da viagem são representados por informações numéricas, como distâncias entre cidades que se pretende visitar, custo de combustível, etc. O programa a ser gerado para auxiliar no planejamento deve ser capaz de representar e manipular essas informações.

Bibliografia inicial:
(PCC, Ch2, 26-29)

PRÉ-DISCUSSÃO

Objetivos de aprendizagem:

Introduzir o aluno à necessidade do uso de informações numéricas em códigos de Python

- Tipos de números: inteiros e reais, e suas respectivas representações em Python: *Integers* e *Floats*.
- Formatação de números para serem apresentados na saída do programa
- Operações com números
- Precedência de operadores e parênteses em Python
- Atribuições múltiplas de valores para variáveis

Ao final da pré:

- O aluno deve se convencer do fato de informações numéricas serem diferentes das de textos vistas no tutorial anterior, em termos de como são armazenadas e manipuladas em linguagens de programação. A partir daí, deve consultar a bibliografia para explorar a sintaxe e as potencialidades desse tipo de informação na linguagem.

PÓS-DISCUSSÃO

Elementos esperados na pós discussão:

O estudo da bibliografia indicada introduz ao aluno as potencialidades de uso de informações numéricas em códigos de Python. O Python pode ser utilizado diretamente no prompt de comando como uma calculadora, com por exemplo em

```
>>> 2 + 3
```

O aluno deve entender a diferença entre números automaticamente convertidos nos tipos “int” ou “float” pela maneira como são informados. A partir daí deve entender a diferença entre os resultados das operações como, por exemplo

```
>>> 2*3
>>> 2*3.0
```

Informações numéricas também podem ser atribuídas a variáveis no Python, e o aluno deve entender os comandos como, por exemplo

```
>>> distancia = 400
>>> print(distancia)
```

7. Agrupando informações

Fonte: elaborado pelo autor com uso de imagine.art.

As anotações do planejamento da viagem possuem vários elementos complementares e alternativos para as escolhas de cada item considerado. Parece uma boa ideia ter a flexibilidade e o poder de manipulação de agruparmos essas informações conjuntos de dados relacionados a cada item.

Bibliografia inicial:
(PCC, Ch3, 34-47)

PRÉ-DISCUSSÃO

Objetivos de aprendizagem:

- Introduzir *lists* do Python como estrutura de armazenamento e manipulação de conjuntos de informações relacionadas

Ao final da pré:

- Os alunos devem identificar a necessidade de armazenamento de informações em conjuntos de valores, e não apenas valores individuais como no caso de variáveis.

- Esse tipo de armazenamento de dado deve propiciar várias formas de manipulação e extração das informações nele contidas.

Até agora, o aluno aprendeu a armazenar e manipular informações numéricas e textuais, mas apenas uma de cada vez. Do enunciado do problema, e da própria experiência dos alunos nas anotações dos planos de viagem usando papel e lápis, deve ficar evidente a necessidade e conveniência de criar estruturas de código que armazenem de uma vez um conjunto de informações relacionadas, bem como da sua manipulação para extração de informações individuais e manipulações dessa informação. A consulta à bibliografia recomendada permitirá ao aluno assimilar os conceitos fundamentais envolvidos nessa questão, bem como aprender exemplos nos quais poderá se apoiar para produzir seu próprio código.

PÓS-DISCUSSÃO

Elementos esperados na pós discussão:

- *Lists* são coleções de informação em uma ordem particular
- Como criar listas
- Como acessar elementos dentro de uma lista
- Protocolo de indexação dos elementos de uma *list* no Python
- *Lists* podem ter seus elementos modificados, adicionados, ou removidos, usando os métodos, como por exemplo em

```
cidades = ['Salvador', 'Trancoso', 'Recife', 'Fortaleza']

cidades.append('Natal')
print(cidades)    # Output: ['Salvador', 'Trancoso', 'Recife',
'Fortaleza', 'Natal']

cidades.remove('Recife')
print(cidades)    # Output: ['Salvador', 'Trancoso', 'Fortaleza',
'Natal']

cidades.insert(1, 'Rio de Janeiro')
print(cidades)    # Output: ['Salvador', 'Rio de Janeiro',
'Trancoso', 'Fortaleza', 'Natal']

index = cidades.index('Fortaleza')
print(index)  # Output: 3

cidades.sort()
print(cidades)  # Output: ['Fortaleza', 'Natal', 'Rio de Janeiro',
'Salvador', 'Trancoso']

cidades.reverse()
```

```
print(cidades)     # Output: ['Trancoso', 'Salvador', 'Rio de Janeiro', 'Natal', 'Fortaleza']
```

8. Trabalhando com informações agrupadas

Fonte: elaborado pelo autor com uso de imagine.art.

Uma vez que agrupamos as informações em listas, os passos seguintes envolvem não só acessar a informação de cada elemento da lista, mas também manipular cada um desses elementos para criar uma nova informação a partir deles. Da revisão das anotações feitas inicialmente usando papel e lápis, um conjunto de informações aparece recorrentemente ao longo dos planos: um conjunto de cidades escolhidas como potenciais destinos da viagem. Para decidir entre essas quais entrariam no roteiro final, seria desejável que essas informações pudessem ser facilmente ser armazenadas conjuntamente, de forma a permitir sua consulta rápida, assim como estabelecer relações com outros conjuntos de informações importantes (por exemplo, distâncias entre cidades, temperaturas médias, etc.).

Bibliografia inicial:
(PCC, Ch4, 50-65)

PRÉ-DISCUSSÃO

Objetivos de aprendizagem:

No tutorial anterior foi introduzido o conceito do agrupamento de informações por meio de “lists”, um conceito fundamental para a programação em Python. Essa estrutura de dados serve dois propósitos:

- Armazenar informações relacionadas em uma única variável
- Extrair e transformar as informações em outras informações, numéricas e textuais, de forma a produzir novos conjuntos de informações relevantes para processos de tomada de decisões.

Além disso, dois tópicos adicionais são introduzidos nesse ponto:

- Looping através dos elementos em uma lista
- Evitar erros da formatação de recuo no código

Ao final da pré:

- O aluno deve estar convencido da necessidade e conveniência de escrever códigos que facilitem a consulta e manipulação de grupos de informações relacionadas, mas ainda não sabe como fazer isso explicitamente usando a sintaxe de Python. Deve ficar claro aqui que a pós-discussão deve englobar os conceitos e exemplos relacionados à bibliografia indicada, com aplicações no problema específico do planejamento da viagem.

PÓS-DISCUSSÃO

Elementos esperados na pós discussão:

- Loops usando *for* :
  - O papel fundamental do uso de ":"
  - O papel fundamental da formatação do código em Python usando recuos.

  Exemplo:

```
cidades = ['Salvador', 'Trancoso', 'Recife', 'Fortaleza']

   for cidade in cidades:
    print(cidade)
```

- Fatiando listas para trabalhar só com parte da informação

Exemplo:

```
cidades = ['Salvador', 'Trancoso', 'Recife', 'Fortaleza']

   print(cidades[0])
 print(cidades[2, 3])
```

- Como fazer cópias de listas

Exemplo:

```
cidades = ['Salvador', 'Trancoso', 'Recife', 'Fortaleza']
```

```
print(cidades)
cidades_copia1 = cidades
print(cidades)
print(cidades1_copia)

cidades = ['Salvador', 'Trancoso', 'Recife', 'Fortaleza']
print(cidades)
cidades_copia2 = cidades.copy()
print(cidades_copia2)
```

9. Premissas Básicas: Informações que desejamos manter inalteradas

Fonte: memes divulgados no facebook.com e no X.

Já criamos algumas listas contendo conjuntos de informações para o problema de planejamento da viagem. Enquanto algumas delas podem ser modificadas ao longo do tempo em função de novas ideias e do próprio funcionamento do código, outras contém informações que não devem ser alteradas, como por exemplo distâncias entre cidades, etc. Já vimos algumas estruturas de armazenamento de informações que permitem a alteração de seus valores ao longo da execução do código de um programa. Em muitas ocasiões isso é essencial para que o programa produza ao final as informações necessárias. Em outras, uma condição para que os programas funcionem sem produzir resultados incorretos é que certos valores não sejam alterados nunca ao longo da execução de um código, na medida em que representam constantes, ou seja, variáveis com informações que não fazem sentido a não ser que ela esteja associada a um único valor, e que esse valor não possa ser alterado como parte do código.

Bibliografia inicial:
(PCC, Ch4, 65-70)

PRÉ-DISCUSSÃO

Objetivos de aprendizagem:

- Entender o conceito de um *tuple* como uma estrutura de armazenamento de dados que não pode ser alterada ao longo da execução de um programa em Python, e sua importância.
- Propriedade *immutable()* de objetos em Pyhton.

Ao final da pré:

- O enunciado torna explícita a necessidade de um tipo de estrutura de dado que contenha informações úteis, mas que nunca podem ser modificadas durante um processo de transformação de outras variáveis com o objetivo de auxiliar tomadas de decisões. Nesse ponto, o aluno ainda não sabe como fazer isso explicitamente no Python, mas deve estar convencido que precisa consultar a bibliografia indicada para adquirir os conceitos necessários, bem como ter acesso a exemplos que não reproduzem perfeitamente seu problema específico, mas que podem ser diretamente adaptados a ele.

PÓS-DISCUSSÃO

Elementos esperados na pós discussão:

Os elementos teóricos adquiridos pela consulta à bibliografia indicada, e que devem ser assimilados são:

- Algumas vezes desejamos criar listas de valores que não devem ser modificados ao longo da execução de um programa
- A maneira de fazer isso no Python é usando a estrutura de *tuples*
- *Tuples* são definidos usando () ao invés de [] como em listas
- *Tuples* possuem o atributo *immutable*
- Valores em *tuples* podem ser acessados por meio de seus índices, e de loops.

Exemplo:

```
cidades_lista = ['Salvador', 'Trancoso', 'Recife', 'Fortaleza']
print(cidades_lista)

cidades_lista[0] = 'Ilheus'
print(cidades_lista)

cidades_tuple = ('Salvador', 'Trancoso', 'Recife', 'Fortaleza')
print(cidades_tuple)

cidades_tuple[0] = 'Ilheus'
 print(cidades_tuple)
```

10. E se...?

Fonte: elaborado pelo autor com uso de imagine.art.

Algumas regras de planejamento devem ser respeitadas na hora de tomar as decisões sobre a viagem. Por exemplo, se o tempo total de deslocamento necessário para um dado roteiro exceder o número de dias disponíveis nas férias, essa opção de roteiro passa a ser desconsiderada. O código de auxílio de planejamento deve ter a capacidade de escolher diferentes opções condicionais nessas regras, e garantir que os resultados produzidos ao final sejam consistentes com elas. Por exemplo, se uma lista de roteiros potenciais inclui uma determinada cidade, uma sub-lista derivada a partir dela não pode incluir outra cidade da lista original.

Bibliografia inicial:
(PCC, Ch5, 71-89)

PRÉ-DISCUSSÃO

Objetivos de aprendizagem:

- Aprender como um programa pode incluir código para examinar um conjunto de condições e, com base nelas, realizar diferentes ações.

Ao final da pré:

- Os alunos devem compreender a necessidade e as utilidades em códigos que permitam aos programas tomar decisões sobre diferentes ações, com base na avaliação de um conjunto de condições. A sintaxe do Python deve permitir esse tipo de programação, e a bibliografia deve mostrar como fazer isso na prática.

PÓS-DISCUSSÃO

Elementos esperados na pós discussão:

- A linguagem Python testa condições usando o comando *if.*
- *If* deve ser programado usando ":" e recuos para separar o que está sendo testado dos códigos seguintes condicionais, e executar os códigos em *else* se o teste não é afirmativo.
- Testes condicionais para igualdade entre valores
- Comparações entre valores de texto e numéricos
- Múltiplas condições são testadas simultaneamente usando *and*
- Pertencimento de valores aos de uma lista são testados usando *in*
- Expressões booleanas

Exemplos:

```
cidades = ['Salvador', 'Trancoso', 'Recife', 'Fortaleza']

sub_cidades1 = ['Salvador', 'Trancoso', 'Recife']
sub_cidades2 = ['Salvador', 'Trancoso', 'Fortaleza']
sub_cidades3 = ['Salvador', 'Recife', 'Fortaleza']
sub_cidades4 = ['Trancoso', 'Recife', 'Fortaleza']

sublists = [sub_cidades1, sub_cidades2, sub_cidades3, sub_cidades4]

sublists = [sublist for sublist in sublists if ('Salvador' in sublist)]

print(sublists)


cidades = ['Salvador', 'Trancoso', 'Recife', 'Fortaleza']

sub_cidades1 = ['Salvador', 'Trancoso', 'Recife']
sub_cidades2 = ['Salvador', 'Trancoso', 'Fortaleza']
sub_cidades3 = ['Salvador', 'Recife', 'Fortaleza']
sub_cidades4 = ['Trancoso', 'Recife', 'Fortaleza']
```

```
   sublists = [sub_cidades1, sub_cidades2, sub_cidades3, sub_cidades4]

   sublists = [sublist for sublist in sublists if ('Salvador' in sublist
and 'Recife' in sublist)]

   print(sublists)



   cidades = ['Salvador', 'Trancoso', 'Recife', 'Fortaleza']

   sub_cidades1 = ['Salvador', 'Trancoso', 'Recife']
   sub_cidades2 = ['Salvador', 'Trancoso', 'Fortaleza']
   sub_cidades3 = ['Salvador', 'Recife', 'Fortaleza']
   sub_cidades4 = ['Trancoso', 'Recife', 'Fortaleza']

   sublists = [sub_cidades1, sub_cidades2, sub_cidades3, sub_cidades4]

   sublists = [sublist for sublist in sublists if not('Salvador' in sublist
and 'Recife' in sublist)]

   print(sublists)



   cidades = ['Salvador', 'Trancoso', 'Recife', 'Fortaleza']

   sub_cidades1 = ['Salvador', 'Trancoso', 'Recife']
   sub_cidades2 = ['Salvador', 'Trancoso', 'Fortaleza']
   sub_cidades3 = ['Salvador', 'Recife', 'Fortaleza']
   sub_cidades4 = ['Trancoso', 'Recife', 'Fortaleza']

   sublists = [sub_cidades1, sub_cidades2, sub_cidades3, sub_cidades4]

   sublists = [sublist for sublist in sublists if ((not 'Salvador' in
sublist)  and 'Recife' in sublist)]

   print(sublists)
```

11. Deduzindo informações

Fonte: elaborado pelo autor com uso de imagine.art.

No problema anterior, vimos que um programa pode executar códigos diferentes, dependendo da avaliação de um conjunto de condições, que podem envolver comparações entre valores numéricos, tamanho de informações em formato de texto, etc. No caso do planejamento da nossa viagem, um exemplo seria a escolha de um determinado roteiro, com base nas preferências dos participantes com relação a diferentes aspectos envolvidos nesse roteiro, como distância máxima entre cidades, se ele passa por praias ou não, se possui comércio de artesanatos no caminho, etc. A escolha final deve considerar as preferências e expectativas de todos os envolvidos, de forma que o código para auxiliar nessa decisão deve levar em conta essas combinações de preferências como restrições às escolhas possíveis. Temos duas dimensões complementares envolvidas aqui: o raciocínio lógico que deduz resultados a partir de combinações de condições, e a implementação dessas restrições em códigos de Python capazes de condicionar diferentes segmentos de instruções ao atendimento de diferentes conjuntos de restrições.

Em termos de raciocínio lógico, “tabelas verdade” são utilizadas para avaliar e representar resultados consistentes com proposições adotadas, enquanto em programação a estrutura de dados associado à ideia matemática de conjuntos (“sets” no Python) permitem obter esse tipo de consistência por meio de operações envolvendo conjuntos.

Bibliografia inicial:

https://www.techtarget.com/whatis/definition/truth-table

HFP(Ch1 , “The Big 4: list, tuple, dictionary and set”)

PRÉ-DISCUSSÃO

Objetivos de aprendizagem:

- Compreensão de Tabelas-Verdade utilizadas em raciocínio lógico
- Entender o uso de informações armazenadas no Python como *boolean values*.
- Relações entre conjuntos e tabelas-verdade
- Como representar e manipular conjuntos por meio da estrutura de dados *set* do Python

Ao final da pré:

- A noção intuitiva de um “conjunto” está disponível para qualquer aluno, assim como a de raciocínios lógicos essenciais, mesmo que nesse ponto o aluno não conheça o formalismo matemático exato das ideias, nem as maneiras de lidar com isso em linguagens de programação.
- O enunciado já propõe um exemplo, com preferências da viagem em relação a distância máxima entre cidades, se ele passa por praias ou não, se possui comércio de artesanatos no caminho. Os alunos devem ser estimulados a pensar em alternativas e complementos a essas preferências, inclusive refletindo suas próprias preferências.
- A ideia da conexão entre a satisfação de restrições de preferências e a ideia de operações envolvendo conjuntos deve agora ser formalizada na pós-discussão após a leitura da bibliografia recomendada.

PÓS-DISCUSSÃO

Elementos esperados na pós discussão:

- Podemos considerar três viajantes com relação a preferências de um roteiro:
  - Custo máximo de combustível
  - Ter praias
  - Ter comércio de artesanato
  - Ter museus

Exemplo:

```
andre = {'150', 'praia', 'artesanato', 'museu'}
ana = {'250', 'praia'}
mario = {'150', 'praia', 'museu'}
```

- Roteiros que agradam a todos os participantes podem ser obtidos a partir de operações entre esses conjuntos:
  - *union()*
  - *difference()*
  - *intersect()*

Exemplo:

```
print(andre.intersection(ana))
print(ana.union(mario))
print(andre.difference(mario))
```

- Os equivalentes de operações lógicas a partir dessas informações consideram variáveis *booleanas* assumindo valor *true* quando um elemente pertence a um conjunto, e *false* quando ele não pertence.

Exemplo:

```
andre = {'150', 'praia', 'artesanato', 'museu'}
ana = {'250', 'praia'}
mario = {'150', 'praia', 'museu'}

prefs_comuns = ana.intersection(mario).intersection(andre)

print(prefs_comuns)

elementos = ana.union(mario).union(andre)

for elemento in andre:
    if elemento in prefs_comuns:
        print(f"{elemento} faz parte das preferencias comuns.")
    else:
       print(f"{elemento} nao faz parte das preferencias comuns.")
```

12. Um jeito eficiente de relacionar informações

Fonte: elaborado pelo autor com uso de imagine.art.

Informações sobre variáveis e parâmetros do problema de planejamento da viagem podem ser estruturadas em listas no código em Python, mas algumas coisas parecem ser feitas de maneira mais fácil e eficiente quando conseguimos conectar partes de informações relacionadas entre si. Já vimos que listas agrupam informações de variáveis distintas, mas seria interessante ter algum tipo de rótulo para identificar e referenciar cada um dos valores dentro de uma lista. No tutorial anterior, por exemplo, podemos pensar nas informações contidas nos conjuntos como associações entre uma lista de viajantes, associada com as respectivas preferências de cada um com relação a aspectos da viagem.

Bibliografia inicial:
(PCC, Ch6, 91-105)

PRÉ-DISCUSSÃO

Objetivos de aprendizagem:

- Incorporar o conceito de estrutura de dados *dictionary* no Python:
  - Como são criados
  - Como valores são:
    - Inseridos

- Transformados
- Acessados

Ao final da pré:

- O conceito de uma estrutura de armazenamento de informações relacionando conjuntos de diferentes variáveis, como pessoas e suas respectivas preferências deve aparecer de forma clara, ainda que intuitiva, durante a discussão. Os alunos ainda não sabem como escrever códigos para representar e manipular essas estruturas, conhecidas no Python como “dictionaries”. A bibliografia indicada deverá ser consultada para habilitar o aluno a entender os conceitos formais e fornecer exemplos de código.

PÓS-DISCUSSÃO

Elementos esperados na pós discussão:

- *Dictionaries* armazenam informações estruturas por pares do tipo “chave-valor”.
- A sintaxe de construção de um *dictionary* envolve o uso de “{}” e “:”.
- Informações específicas dentro de *dictionaries* podem ser obtidas usando o método *.get()*.
- Informações individuais dentro de *dictionaries* podem ser obtidas por meio da combinação de loops e do método *.items()*.

Exemplo

```
andre = {'150', 'praia', 'artesanato', 'museu'}
ana = {'250', 'praia'}
mario = {'150', 'praia', 'museu'}

pessoas_preferencias = {
    'andre': andre,
    'ana': ana,
    'mario': mario
}

print(pessoas_preferencias)
print(type(pessoas_preferencias))


andre_prefs = pessoas_preferencias['andre']
print(andre_prefs)


andre_prefs = pessoas_preferencias.get('andre')
print(andre_prefs)
```

```
for pessoa, prefs in pessoas_preferencias.items():
    print(f"{pessoa} prefere {prefs}")
```

13. Matrioskas

Fonte: elaborado pelo autor com uso de imagine.art.

Algumas das informações para o planejamento da viagem já foram armazenadas usando as propriedades e métodos eficientes da estrutura de *dictionaries*. Olhando para alguns dos itens representando essas informações, notamos que eles podem ser melhor organizados por meio de uma estrutura hierárquica na forma de sub-items. Por exemplo, um *dictionary* contendo informações sobre diferentes cidades pode, para cada uma dessas cidades, conter informações sobre custo médio de hotéis, principais atrações, melhores épocas do ano para se visitar, etc.

Bibliografia inicial:
(PCC, Ch6, 105-112)

PRÉ-DISCUSSÃO

Objetivos de aprendizagem:

- Aprender a estruturação de informações de forma hierárquica dentro de *lists* e *dictionaries* no Python:
  - Listas de *dictionaries*
  - Listas em *dictionaries*
  - *Dictionaries* dentro de *dictionaries*

Ao final da pré:

- O aluno deve propor ainda de forma conceitual uma estrutura hierárquica de organização de dados referenciados.
- A implementação dessa ideia no Python será possível após estudar a bibliografia apontada.

PÓS-DISCUSSÃO

Elementos esperados na pós discussão:

- *Dictionaries* e listas podem ser definidos de forma hierárquica, em qualquer combinação que se imagine eficiente para um problema específico em questão:
  - Lista de *dictionaries*. Ex*:*
    - Dict1 = {‘nome: ‘A’, ‘praia’: false, ‘atracoes’: ‘parques’}
    - Dict2 = {‘nome: ‘B’, ‘praia’: true, ‘atracoes’: ‘restaurantes’}
    - Cidades = [Dict1, Dict2]
  - Lista em um *dictionary*. Ex:
    - Dict1 = {‘nome: ‘A’, ‘praia’: false, ‘atracoes’: [‘parques’, ‘trilhas’]}
  - *Dictionaries* em *dictionaries*. Ex:
    - *Cidades* = {
      ‘cidade1’: {
      ‘nome: ‘A’, ‘praia’: false, ‘atracoes’: ‘parques’
      },
      ‘cidade2’: {
      ‘nome: ‘B’, ‘praia’: true, ‘atracoes’: ‘restaurantes’
      },
      }

Exemplo:

```
cidades_info = {
    'Salvador': {
        'custo_medio_hoteis': 200,
        'atracoes': ['praia', 'carnaval', 'historia'],
        'melhores_epocas': ['verao', 'carnaval']
    },
    'Trancoso': {
        'custo_medio_hoteis': 300,
        'atracoes': ['praia', 'natureza', 'tranquilidade'],
        'melhores_epocas': ['verao', 'primavera']
    },
    'Recife': {
        'custo_medio_hoteis': 150,
```

```
            'atracoes': ['praia', 'carnaval', 'historia'],
            'melhores_epocas': ['verao', 'carnaval']
        },
        'Fortaleza': {
            'custo_medio_hoteis': 180,
            'atracoes': ['praia', 'sol', 'comida regional'],
            'melhores_epocas': ['verao', 'outono']
        }
    }

print(cidades_info)
```

14. Enquanto esperamos...

Fonte: elaborado pelo autor com uso de imagine.art.

O programa para o planejamento da viagem está ganhando corpo. Várias informações importantes para as escolhas a serem feitas já estão sendo armazenadas em diferentes estruturas, cada uma da forma mais conveniente e eficiente para podermos trabalhar com elas. Outras informações dependem de respostas dos interessados para questões que ainda não foram consideradas, e seria desejável que o programa fosse capaz de pedir essas informações para os viajantes, para então armazená-las da forma mais adequada junto com as demais variáveis.

Bibliografia inicial:
(PCC, Ch7, 114-123)

PRÉ-DISCUSSÃO

Objetivos de aprendizagem:

- Aprender como um programa pode solicitar e aceitar informações de seus usuários
- Como as informações recebidas são armazenadas e manipuladas programaticamente
- Para de esperar informações do usuário e continuar a execução do programa após alguma condição ser satisfeita

Ao final da pré:

- Os alunos devem identificar a necessidade do programa solicitar informações adicionais dos usuários, como incorporar as informações recebidas, e quando para de solicitar informações para dar prosseguimento às demais tarefas consideradas pelo código.

PÓS-DISCUSSÃO

Elementos esperados na pós discussão:

- Informações de usuários devem ser solicitadas pela função *input()*
- Informações a partir de *input()* serão sempre consideradas como *strings*. Se uma informação particular é do tipo numérico, os valores devem ser convertidos pelas funções *int()* ou *float()*, conforme o caso.
- Loops do tipo *while* podem ser usados para testar uma condição e parar de solicitar inputs de usuários quando essa condição for atendida.

Exemplo:

```
pessoas_preferencias = {}

while True:
    pessoa = input("Entre o nome da pessoa (ou 'stop' para terminar): ")
    if pessoa.lower() == 'stop':
        break

    prefs = input("Entre as preferências da pessoa, separadas por vírgulas:
")
    prefs = set(prefs.split(', '))

    pessoas_preferencias[pessoa] = prefs

for pessoa, prefs in pessoas_preferencias.items():
    print(f"{pessoa} prefere {prefs}")
```

15. Adquirindo e movendo informações

Fonte: elaborado pelo autor com uso de imagine.art.

As informações para o planejamento da viagem já estão armazenadas em diferentes estruturas dentro do programa. Isso não impede que informações adicionais possam ser requisitadas, à medida em que novas ideias, restrições e oportunidades vão surgindo durante o processo de planejamento. Um loop do tipo *while* permite que várias informações sejam solicitadas ao mesmo tempo para um usuário, e que sejam armazenadas nas estruturas correspondentes. Esse processo vai modificando listas e *dictionaries* ao longo da execução do programa, não só adicionando, mas também removendo e transferindo essas informações entre essas listas e *dictionaries*. Por exemplo, a partir de uma lista inicial de cidades consideradas para a viagem, duas listas podem ser criadas e modificadas com base em informações dos usuários. Se uma determinada cidade for descartada por um dos participantes da viagem, ela pode passar de uma lista de cidades consideradas para outra de cidades vetadas.

Bibliografia inicial:
(PCC, Ch7, 124-127)

PRÉ-DISCUSSÃO

Objetivos de aprendizagem:

- Possibilitar a um programa absorver informações fornecidas por usuários
- Armazenar essas informações em estruturas de dados adequadas
- Modificar conjuntos de valores de acordo com a interação entre informações dos usuários e decisões de ação tomadas pelo código em um programa

Ao final da pré:

- O aluno deve entender a necessidade da flexibilidade na coleta de informações por parte de um usuário do programa que está sendo desenvolvido, e antecipar que métodos eficientes estão presentes para essa tarefa, a serem aprendidos a partir da bibliografia.

PÓS-DISCUSSÃO

Elementos esperados na pós discussão:

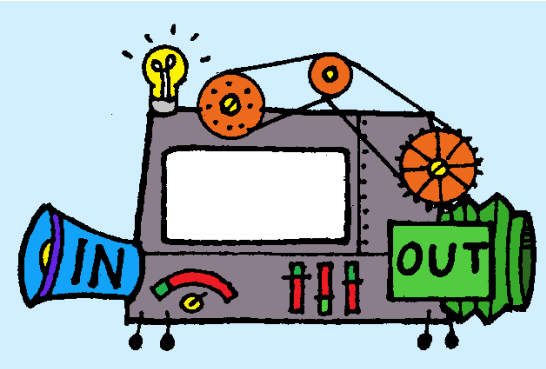
O código deve incorporar a coleta de informações a respeito de múltiplas variáveis por meio do método *input()* combinado com *loops* do tipo *while*. Com base nos valores fornecidos pelo usuário, o programa deve conseguir mover informações novas e já existentes entre diferentes estruturas de dados armazenados. O código específico para isso está na bibliografia indicada.

Exemplo

```
if 'andre' in pessoas_preferencias:
    pessoas_preferencias['andre'].discard('carnaval')

cidades_com_prefs = []
cidades_sem_prefs = []

for cidade, info in cidades_info.items():
    for pessoa, prefs in pessoas_preferencias.items():
        if prefs.issubset(info['atracoes']):
            cidades_com_prefs.append(cidade)
        else:
            cidades_sem_prefs.append(cidade)

print("cidades com preferencias de todas as pessoas:", cidades_com_prefs)
  print("cidades com ao menos uma preferencia ausente:", cidades_sem_prefs)


while True:
    command = input("Entre 'add' para adicionar uma preferência, 'mudar' para alterar uma preferência, ou 'stop' para terminar: ")
    if command.lower() == 'stop':
        break
    elif command.lower() == 'add':
        pessoa = input("Entre o nome da pessoa: ")
        pref = input("Entre a preferência a adicionar: ")
        if pessoa in pessoas_preferencias:
            pessoas_preferencias[pessoa].add(pref)
        else:
```

```
            pessoas_preferencias[pessoa] = {pref}
    elif command.lower() == 'mudar':
       pessoa = input("Entre o nome da pessoa: ")
       pref_remove = input("Entre a preferência a ser removida: ")
       pref_nova = input("Entre a preferência a ser adicionada: ")
        if pessoa in pessoas_preferencias:
            pessoas_preferencias[pessoa].discard(pref_remove)
            pessoas_preferencias[pessoa].add(pref_nova)

for pessoa, prefs in pessoas_preferencias.items():
 print(f"{pessoa} prefere {prefs}")
```

16. Máquinas de transformar e criar informações



Fonte: elaborado pelo autor com uso de imagine.art.

À medida em que o programa de planejamento da viagem vai crescendo e incorporando novas informações, você começa a notar que certas tarefas são realizadas várias vezes, gerando muita repetição do código. Por exemplo, um dicionário de cidades consideradas para o trajeto da viagem contém informações sobre a diária mínima de um quarto de hotel, e o número de dias recomendado para ficar naquela cidade. Com base nessas informações, você se encontra usando o mesmo código para calcular o custo de estadia total para cada cidade. Um recurso de praticamente qualquer linguagem de programação evita essa repetição (assim como erros potenciais de cópia de código!) é o análogo ao conceito matemático de “função”.

Bibliografia inicial:
(PCC, Ch8, 130-140)

PRÉ-DISCUSSÃO

Objetivos de aprendizagem:

- Aprender o conceito de função em programação, como blocos de código criados para realizar uma tarefa específica
- Como na definição matemática, funções em linguagens de programação mapeiam argumentos em resultados
- Funções tornam o código de um programa mais fácil de escrever, ler, testar e eventualmente corrigir

Ao final da pré:

- O aluno deve deduzir que cálculos repetidos, onde apenas os argumentos são alterados, devem ser implementados em um conjunto de instruções único.

- O conceito de função, no sentido matemático, deve surgir na discussão como forma de resolver problemas repetidos desse tipo, e a mesma ideia aplicada à programação deve ser apresentada na bibliografia indicada.

PÓS-DISCUSSÃO

Elementos esperados na pós discussão:

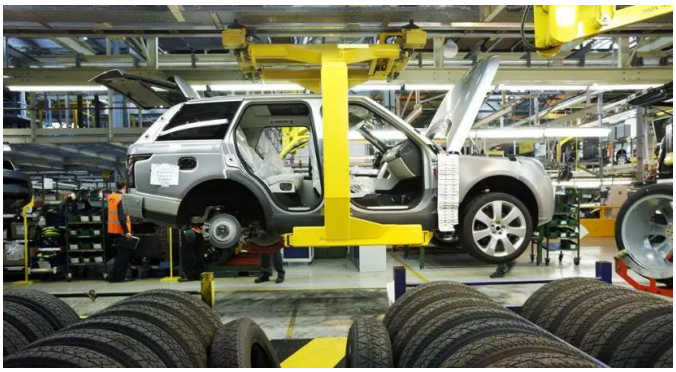
Em programação, funções são:

- Uma forma eficiente de mapear argumentos em resultados, contendo o código para realização dos cálculos necessários
- Uma forma de evitar duplicidade de código ao longo de um programa, evitando ao mesmo tempo possíveis erros, ou no mínimo tornar mais fácil identificar e corrigir eventuais erros
- Definidas no Python usando a sintaxe *def nome_funcao(argumentos): código*
- Podem aceitar argumentos na forma de variáveis, listas ou *dictionaries*
- Podem retornar valores na forma de variáveis, listas ou *dictionaries*
- Podem ser definidas para aceitar argumentos múltiplos, opcionais e com valores pré-definidos

Exemplo:

```
def calculate_costs(cidades_info, dias_estadia):
    custo_cidade_hotel = {}
    for cidade, info in cidades_info.items():
        if cidade in dias_estadia:
            custo_cidade_hotel[cidade] = info['custo_medio_hoteis'] *
dias_estadia[cidade]
    return custo_cidade_hotel

custo_medio_hoteis= {
    'Salvador': 150,
    'Trancoso': 100,
    'Recife': 130,
    'Fortaleza': 250
}


dias_estadia = {
    'Salvador': 5,
    'Trancoso': 7,
    'Recife': 4,
    'Fortaleza': 6
}

custo_cidade_hotel = calculate_costs(cidades_info, dias_estadia)
print(custo_cidade_hotel)
```

17. Máquinas de transformar e criar informações II



Fonte: elaborado pelo autor com uso de imagine.art.

No problema anterior, imaginamos um *dictionary* de cidades consideradas para o trajeto da viagem contendo informações sobre a diária média de um quarto de hotel, e o número de dias recomendado para ficar naquela cidade. Essa estrutura de dados pode ir crescendo à medida em que o planejamento progride, passando a incluir informações adicionais, como por exemplo listas de atrações de cada cidade, temperatura média ao longo dos meses, avaliações em guias de viagem, etc. Quando programamos uma função para transformar essas informações, devemos considerar as possibilidades de algumas delas não serem usadas em nenhum momento, de algumas serem consideradas opcionais, e de outras estarem eventualmente ausentes e poderem ser substituídas por valores pré-definidos. A nova função pode considerar o custo total de estadia em cada cidade analisada incorporando descontos para as diárias quando tivermos atrações na cidade, e diárias que aumentam junto com a temperatura média da cidade.

Bibliografia inicial:
(PCC, Ch8, 140-149)

PRÉ-DISCUSSÃO

Objetivos de aprendizagem:

- Entender como funções no Python podem ser programadas de forma flexível para aceitar argumentos em diferentes configurações

Ao final da pré:

- O aluno deve concluir que programação de funções deve levar em consideração situações em que não existe certeza de antemão sobre o número e disponibilidade de informações de seus argumentos. Portanto, no momento em que são criados os códigos

para implementar funções, precisamos de flexibilidade em relação à definição e uso dos argumentos.

PÓS-DISCUSSÃO

Elementos esperados na pós discussão:

- Funções definidas em Python podem:
  - Receber argumentos múltiplos na forma de listas e *dictionaries*
  - Conter um número arbitrário de valores para um argumento (usando a sintaxe **arg*)
  - Conter um número arbitrário de argumentos (usando a sintaxe ***arg*)

Exemplo:

```
def calcula_costs(cidades_info, dias_estadia, **args):
    atracoes = args.get('atracoes', None)
    temperatura_media = args.get('temperatura_media', 25)

    custo_cidade_hotel = {}
    for cidade, info in cidades_info.items():
        if cidade in dias_estadia:
            custo = info['custo_medio_hoteis'] * dias_estadia[cidade]
            if atracoes and atracoes in info['atracoes']:
                custo *= 0.9
            if info.get('temperatura_media', temperatura_media) > 30:
                custo *= 1.1
            custo_cidade_hotel[cidade] = custo
    return custo_cidade_hotel

dias_estadia = {
    'Salvador': 5,
    'Trancoso': 7,
    'Recife': 4,
    'Fortaleza': 6
}

custo_cidade_hotel = calcula_costs(cidades_info, dias_estadia,
atracoes='praia')
print(custo_cidade_hotel)
```

18. Containers de Informação

Fonte: elaborado pelo autor com uso de imagine.art.

Além de evitar duplicação de códigos em programas, funções também possuem a vantagem de organizar e tornar mais fácil a leitura. É sempre uma boa ideia formatar o código de forma a facilitar sua compreensão por parte de outras pessoas, e de você mesmo quando voltar a ele algum tempo depois. Um passo adicional para ajudar na organização e reuso de códigos envolve armazenar funções em arquivos separados de um programa específico que utiliza essas funções.

Bibliografia inicial:
(PCC, Ch8, 149-154)

PRÉ-DISCUSSÃO

Objetivos de aprendizagem:

- Introduzir o conceito de formatação e estilo em códigos definindo funções
- Aprender a armazenar conjuntos de funções que desempenham tarefas relacionadas em módulos no Python

Ao final da pré:

- O aluno deve ter consciência da utilidade de gerar código não só sintaticamente e conceitualmente correto, mas também de organizá-lo e formatá-lo de formas eficientes, e buscar na bibliografia indicada as maneiras de fazê-lo.

PÓS-DISCUSSÃO

Elementos esperados na pós discussão:

- Funções devem ser codificadas usando estratégias de estilo para melhorar sua compreensão:
  - Comentários explicando o que a função faz, quais valores ela espera como argumentos, e quais resultados são produzidos
  - Nomes devem ser escolhidos de forma a identificar facilmente uma função, e distingui-la de outras estruturas de dados presentes no programa, como variáveis, listas e *dictionaries*
- Módulos podem ser criados para separar o código que define funções do código que as utiliza para problemas específicos

Módulos contendo funções são incorporados a um programa por meio do comando *import*, sempre no começo do programa.

Exemplo:

Salvar o código abaixo em um arquivo “calculador_viagem.py” em um diretório com nome escolhido por você (por exemplo “meus_modulos”):

```
def calcula_custos(cidades_info, dias_estadia, **args):
    atracoes = args.get('atracoes', None)
    temperatura_media = args.get('temperatura_media', 25)

    custo_cidade_hotel = {}
    for cidade, info in cidades_info.items():
        if cidade in dias_estadia:
            custo = info['custo_medio_hoteis'] * dias_estadia[cidade]
            if atracoes and atracoes in info['atracoes']:
                custo *= 0.9
            if info.get('temperatura_media', temperatura_media) > 30:
                custo *= 1.1
            custo_cidade_hotel[cidade] = custo
    return custo_cidade_hotel
```

No arquivo contendo outros códigos que utlizarão a função “calcula_custos”, incluir o seguinte código:

```
from meus_modulos.calculador_viagem import calcula_custos
```

A partir daí, as outras linhas de código desse arquivo têm acesso à função “calcula_custos” importada do pacote.

19. Coisas diferentes em detalhes, mas iguais na essência

Fonte: elaborado pelo autor com uso de imagine.art.

As variáveis utilizadas pelo nosso programa de planejamento de viagem vêm sendo definidas em função do tipo de informação que contém (texto, número, etc). Vimos que essas informações podem ser agrupadas em listas, *dictionaries*, conjuntos e *tuples*, de acordo com a conveniência de cada uma dessas estruturas. Por trás disso tudo, o Python opera segundo um princípio básico: cada tipo de estrutura de dados possui atributos e funcionalidades que são utilizadas programaticamente. Por exemplo, uma variável do tipo *string* possui um atributo *len* informando o número de caracteres nessa palavra, e a função *.upper()* que transforma todos os caracteres da palavra em maiúsculos. Qualquer *string* terá essas (e muitas outras!) funcionalidades prontas uso. No código do seu programa de planejamento da viagem, alguns elementos podem se beneficiar de estruturas que incorporem ao mesmo tempo a semelhança e as especificidades entre eles. Por exemplo, cada pessoa do grupo que vai viajar pode possuir um carro. Na escolha de qual carro usar, informações específicas para cada modelo podem ser combinadas com atributos e funcionalidades comuns a todos eles (número de passageiros, consumo médio de combustível...). Como outro exemplo, as cidades consideradas potencialmente no roteiro possuem características básicas (temperatura, atrações, custo de hotéis, localização, etc) que as diferenciam assumindo valores particulares.

Bibliografia inicial:
(PCC, Ch9, 158-166)

PRÉ-DISCUSSÃO

Objetivos de aprendizagem:

- Introduzir o conceito fundamental de linguagens de programação orientadas a objetos (como o Python): classes
- Sintaxe para definição de classes
- Programação de métodos e atributos em classes

Ao final da pré:

- O aluno deve deduzir a existência de uma forma programática para definir estruturas de dados com as características do problema: características gerais comuns dos objetos, junto com atributos específicos que diferenciam casos particulares.

PÓS-DISCUSSÃO

Elementos esperados na pós discussão:

- Classes são definidas no Python por meio da sintaxe *class nome_classe: código*
- Objetos específicos representam instâncias de uma classe geral, possuindo todas as funcionalidades e atributos dessa classe, mas podendo ser modificados para representar informações específicas
- Métodos de classes são definidos usando a sintaxe de funções: *def nome_metodo():*
- Atributos de classes são acessados e modificados usando a sintaxe *objeto_classe.atributo*

Exemplo:

O código abaixo usa funções do modulo “math” (referindo ao tutorial passado sobre módulos), e cria uma classe “Cidade” com atributos “temperatura_media”, “custo_medio_hoteis”, “atracoes”, “lon” (longitude) e “lat” (latitude). Essas duas últimas informações são facilmente obtiveis, por exemplo pelo Google Maps, e podem ser utilizadas para calcular aproximadamente a distância entre cidades.

```
   import math

   class Cidade:
       def __init__(self, temperatura_media, custo_medio_hoteis, atracoes,
praia, lon, lat):
           self.temperatura_media = temperatura_media
           self.custo_medio_hoteis = custo_medio_hoteis
           self.atracoes = atracoes
           self.praia = praia
           self.lon = lon
           self.lat = lat

       def calculate_distance(self, other_city):
           return math.sqrt((self.x - other_city.x)**2 + (self.y -
other_city.y)**2)

   # Exemplos de utilização:
   cidade1 = Cidade(25, 100, ['museu', 'parque'], True, 1, 1)
   cidade2 = Cidade(30, 150, ['praia', 'parque'], False, 4, 5)

   print(cidade1.calculate_distance(cidade2))
```

20. Transmissão genética em programação

Fonte: elaborado pelo autor com uso de imagine.art.

Classes são uma forma eficiente de estruturar e usar informações em um programa. Em algumas situações, classes estendem os atributos e funcionalidades de outras classes já existentes, e assim o código que as definem não precisa começar do zero. Por exemplo, no caso do nosso programa de planejamento da viagem, alguns dos elementos do grupo que vai viajar podem possuir carros elétricos ou híbridos, enquanto outros possuem carros movidos a combustão.

Bibliografia inicial:
(PCC, Ch9, 167-173)

PRÉ-DISCUSSÃO

Objetivos de aprendizagem:

- Introduzir o conceito de *inheritance*, fundamental no paradigma de programação orientada a objetos, como no caso do Python.

Ao final da pré:

- Pelo enunciado do problema, deve ficar claro para o aluno que para não começar a codificar a criação de uma classe do zero, deve existir alguma forma programática de absorver atributos e métodos de alguma classe já existente, para depois estender o comportamento definindo uma nova classe. O título do problema já sugere alguma ideia de “herança”, que o aluno deve esperar seja formalizada durante a leitura da bibliografia sugerida para a pós-discussão.

PÓS-DISCUSSÃO

Elementos esperados na pós discussão:

- Quando uma classe extende a funcionalidade de outras já existentes, podemos usar o método de *inheritance* no Python
- A classe original é chamada *parent class*, enquanto a nova classe é chamada *child class*, tornando evidente a analogia com o processo de herança genética em seres vivos
- Métodos e Atributos da *child class* podem ser codificados da mesma forma que na definição de classes, podendo inclusive substituir os valores herdados da *parent class*, se for o caso.

Exemplo:

O código abaixo define uma classe “Carro”, com atributos “marca”, “modelo”, “ano” e “cor”, e métodos “acelerar”, “brecar” e “buzinar”.

```
class Carro:
    def __init__(self, marca, modelo, ano, cor):
        self. marca = marca
        self.modelo = modelo
        self.ano = ano
        self.cor = cor
        self.velocidade = 0

    def acelerar(self, montante):
        self.velocidade += montante
        return self.velocidade

    def brecar(self, montante):
        self.speed -= montante
        if self.velocidade < 0:
            self.velocidade = 0
        return self.velocidade

    def buzinar(self):
        return f"{self.marca} {self.modelo} faz fon-fon!”}
```

Exemplo de uso:

```
meu_carro = Car('Toyota', 'Corolla', 2021, 'Azul')
print(meu_carro.acelerar(20))
print(meu_carro.brecar(10))
print(meu_carro.buzinar())
```

Baseada na classe “Carro”, o código abaixo define uma classe derivada (child class) “CarroEletrico”, herdando os atributos “marca”, “modelo”, “ano”, “cor”, e introduzindo os atributos “capacidade_bateria” e “nivel_carga”, e os métodos “carregar” e “dirigir”.

```
class CarroEletrico (Carro):
    def __init__(self, marca, modelo, ano, cor, capacidade_bateria):
        super().__init__(marca, modelo, ano, cor)
        self. capacidade_bateria = capacidade_bateria
        self.nivel_carga = 100

    def carregar(self):
        self. nivel_carga = 100
        return self.nivel_carga

    def dirigir(self, distancia):
        if distancia > self.nivel_carga:
            print("Not enough charge to drive that far.")
        else:
            self.nivel_carga -= distancia
            return self.nivel_carga
```

Exemplo de uso:

```
meu_carro_eletrico = CarroEletrico('Tesla', 'Model 3', 2020, ‘Vermelho’,
75)
print(meu_carro_eletrico.dirigir(50))
print(meu_carro_eletrico.carregar())
```

21. Recebendo entrega de informações

Fonte: elaborado pelo autor com uso de imagine.art.

Anteriormente, vimos que o Python permite armazenar funções em arquivos separados de um programa principal. Qualquer programa pode então ter acesso às funções armazenadas em um arquivo por meio do comando *import*. O mesmo vale para estruturas de dados cujas propriedades e funcionamento são implementados por meio da definição de classes.

Bibliografia inicial:
(PCC, Ch9, 173-179)

PRÉ-DISCUSSÃO

Objetivos de aprendizagem:

- Introduzir os conceitos fundamentais do comando *import* para acessar classes contidas em módulos:
  - Importar uma única classe entre todas contidas em um módulo
  - Importar várias classes contidas em um módulo
  - Importar um módulo inteiramente
  - Uso de *aliases*

Ao final da pré:

- O aluno deve identificar a existência da possibilidade do uso de classes armazenadas em módulos, e consultar a bibliografia para conhecer os detalhes envolvidos.

PÓS-DISCUSSÃO

Elementos esperados na pós discussão:

- Classes definidas por códigos podem ser armazenadas em módulos
- Qualquer classe armazenada em um módulo, seja pelo programador ou por outros programadores, pode ser acessada em um programa por meio do comando *import*
- *import* tem diferentes opções, para lidar com a importação eficiente de classes contidas em módulos contendo potencialmente um grande número de outras classes

Exemplo:

As duas classes definidas no tutorial anterior devem ser salvas em um mesmo arquivo "modulo_carros.py" no diretório "meus_modulos":

```
class Carro:
    def __init__(self, marca, modelo, ano, cor):
        self. marca = marca
        self.modelo = modelo
        self.ano = ano
        self.cor = cor
        self.velocidade = 0

    def acelerar(self, montante):
        self.velocidade += montante
        return self.velocidade

    def brecar(self, montante):
        self.speed -= montante
        if self.velocidade < 0:
            self.velocidade = 0
        return self.velocidade

    def buzinar(self):
        return f"{self.marca} {self.modelo} faz fon-fon!"}



class CarroEletrico (Carro):
    def __init__(self, marca, modelo, ano, cor, capacidade_bateria):
        super().__init__(marca, modelo, ano, cor)
        self. capacidade_bateria = capacidade_bateria
        self.nivel_carga = 100

    def carregar(self):
        self. nivel_carga = 100
        return self.nivel_carga

    def dirigir(self, distancia):
```

```
        if distancia > self.nivel_carga:
            print("Not enough charge to drive that far.")
        else:
            self.nivel_carga -= distancia
            return self.nivel_carga
```

Os códigos abaixo são exemplos de como tornar essas classes disponíveis dentro de um outro programa, usando opções do comando “import” para carregá-las conjuntamente ou separadamente, e usando “aliases”:

```
from modulos_carro import Carro, Carro_Eletrico
from modulos_carro import Carro
from modulos_carro import Carro_Eletrico
from modulos_carro import Carro as car
from modulos_carro import CarroEletrico as carelet
```

22. A mãe de todas as bibliotecas

Fonte: elaborado pelo autor com uso de imagine.art.

A essa altura já vimos que qualquer informação armazenada em algum programa no Python é uma instância particular de uma classe representando uma estrutura de dados (por exemplo, uma *string*, ou um *dictionary* de *strings*, etc.). Enquanto algumas estruturas de dados são implementadas em módulos que não fazem parte da instalação padrão do Python, outras são de uso tão comum e fundamental que fazem com que sejam incluídas na instalação Padrão. Algumas das decisões do grupo que vai fazer a viagem estão ficando difíceis, e alguém dá a sugestão de usar um critério aleatório para tomada de decisões em alguns desses casos. Por exemplo, o roteiro escolhido até agora passa por Fortaleza e Recife, e o grupo decide parar em alguma outra cidade no meio do caminho. Olhando o mapa, algumas alternativas ao longo da mesma rodovia são:

1. **Itaitinga, CE:** Município próximo a Fortaleza.
2. **Horizonte, CE:** Outro município situado ao longo do caminho.
3. **Pacajus, CE:** Cidade localizada na região metropolitana de Fortaleza.
4. **Chorozinho, CE:** Mais uma cidade no percurso entre Fortaleza e Recife.
5. **Russas, CE:** Município importante no interior do Ceará.
6. **Limoeiro do Norte, CE:** Outra cidade situada ao longo da BR-116.
7. **Quixeré, CE:** Município no trajeto entre Fortaleza e Recife.
8. **Icó, CE:** Cidade histórica situada na região central do Ceará.
9. **Salgueiro, PE:** Primeira cidade em Pernambuco, já na BR-232.
10. **Serra Talhada, PE:** Importante cidade no sertão pernambucano.
11. **Arcoverde, PE:** Outro município na BR-232, a caminho de Recife.
12. **Caruaru, PE:** Conhecida cidade pernambucana, famosa pelo São João.
13. **Gravatá, PE:** Município próximo a Recife, já na região metropolitana.

A princípio todos são indiferentes em relação à escala em qualquer uma delas. Uma alternativa seria estudar com maior profundidade as atrações listadas acima, mas o grupo decide ser espontâneo nessa parte e simplesmente escolher alguma entre elas aleatoriamente.

Bibliografia inicial:
(PCC, Ch9, 179-181)

PRÉ-DISCUSSÃO

Objetivos de aprendizagem:

- Conhecer a PSL (Python Standard Library), sua organização em módulos e as várias funcionalidades que ela oferece.

Ao final da pré:

- Os alunos devem identificar a necessidade de um mecanismo para escolha aleatória entre uma lista de alternativas que dificulta soluções simples, como jogar uma moeda. Geradores de números aleatórios são muito importantes no estudo de estatística e econometria, e aqui temos uma oportunidade de introduzir esses algoritmos como parte das funcionalidades contidas na instalação default do Python. Nesse ponto, os alunos devem estar convencidos da possibilidade de fazer esse tipo de escolha aleatória usando códigos de Python, e deverão buscar na bibliografia indicada a maneira formal de fazê-lo.

PÓS-DISCUSSÃO

Elementos esperados na pós discussão:

- A *Python Standard Library* inclui um conjunto de módulos que fazem parte da instalação padrão do Python
- As funcionalidades e estruturas de dados contidas nesses módulos podem ser acessadas pelo comando *import*
- Por exemplo, a função que gera números aleatórios inteiros que podem ser utilizados em um sorteio é acessada pelo comando *from random import randint*

Exemplo 1:

```
import random

cidades = ['Itaitinga', 'Horizonte', 'Pacajus', 'Chorozinho', 'Russas', 'Limoeiro do Norte', 'Quixeré', 'Icó', 'Salgueiro', 'Serra Talhada', 'Arcoverde', 'Caruaru', 'Gravatá']

indice_aleatorio = random.randint(0, len(cidades) - 1)
cidade_escolhida = cidades[indice_aleatorio]

print("Cidade escolhida:", cidade_escolhida)
```

Exemplo 2:

```
import random

cidades = ['Itaitinga', 'Horizonte', 'Pacajus', 'Chorozinho', 'Russas', 'Limoeiro do Norte', 'Quixeré', 'Icó', 'Salgueiro', 'Serra Talhada', 'Arcoverde', 'Caruaru', 'Gravatá']

cidade_escolhida = random.choice(cidades)

print("Cidade escolhida:", cidade_escolhida)
```

23. Arquivos

Fonte: elaborado pelo autor com uso de imagine.art.

Durante suas pesquisas para o planejamento da viagem, você descobre um site contendo arquivos com informações sobre atrações de diferentes cidades. Por exemplo, o arquivo “atracoes_cidadeA.txt”, disponibilizado no Teams, contém informações sobre horários de funcionamento e localização de algumas atrações em uma cidade. Adicionalmente, você percebe que poderia também ser eficiente armazenar parte das informações que estão no código do programa sendo gerado em para arquivos externos.

Bibliografia inicial:
(PCC, Ch10, 183-191)

PRÉ-DISCUSSÃO

Objetivos de aprendizagem:

- Aprender a ler arquivos de dados dentro de programas em Python, utilizando funções da *Python Standard Library*
- Aprender a escrever arquivos contendo dados gerados e armazenados dentro do código de um programa, utilizando funções da *Python Standard Library*

Ao final da pré:

- O aluno inspecionou o arquivo citado no enunciado do problema, e identificou informações relevantes nele contidas. O próximo passo é estudar a bibliografia para entender como importar essas informações para dentro do programa.
- O aluno deve também entender a conveniência de armazenar informações em arquivos externos ao do programa, e aprender a fazer isso consultado a bibliografia indicada

PÓS-DISCUSSÃO

Elementos esperados na pós discussão:

- As informações do arquivo fornecido devem ser lidas utilizando as funções *Path()* e *path.read_text()* do módulo *pathlib*
- Os valores importados podem ser manipulados por funções como *splitlines()* para que sejam manipulados de forma mais eficiente pelo programa
- As informações contidas e geradas pelo programa podem ser armazenadas em arquivos externos utilizando as funções *Path()* e *path.write_text()* do módulo *pathlib*

Exemplo 1:

```
from pathlib import Path

path = Path('atracoes_cidadeA.txt')
atracoes_cidadeA = path.read_text().splitlines()

print(atracoes_cidadeA)
```

Exemplo 2:

```
with open('atracoes_cidadeA.txt', 'r') as file:
    atracoes_cidadeA = file.read().splitlines()

print(atracoes_cidadeA)
```

Exemplo3:

```
import random
from pathlib import Path

cidades = ['Itaitinga', 'Horizonte', 'Pacajus', 'Chorozinho', 'Russas',
'Limoeiro do Norte', 'Quixeré', 'Icó', 'Salgueiro', 'Serra Talhada',
'Arcoverde', 'Caruaru', 'Gravatá']

cidades_escolhidas = random.sample(cidades, 3)
```

```
cidades_escolhidas _text = '\n'.join(cidades_escolhidas)

path = Path('cidades_escolhidas.txt')
path.write_text(cidades_escolhidas _text)
```

24. Imprevistos acontecem

Fonte: elaborado pelo autor com uso de imagine.art.

Ao planejarmos uma viagem, inicialmente imaginamos que tudo vai dar certo e as coisas vão acontecer exatamente de alguma maneira. Pensando com mais calma, nos damos conta que na prática pode não ser bem assim, na medida em que eventos que nem imaginávamos podem mudar o rumo para longe do planejamento inicial. Por exemplo, uma chuva muito forte pode causar um deslizamento de terra e bloquear uma estrada pela qual deveríamos passar. Precisamos pensar então em alguns tipos de "plano B" para situações como essa, de forma a não comprometer o restante da viagem no caso de algum imprevisto acontecer. Com programação, uma situação muito semelhante acontece às vezes: o código encontra um problema, normalmente gerando uma mensagem de erro, e simplesmente para de executar se não fizermos algo para, além de antecipar essa possibilidade, fornecer um "plano B" para o código. Como exemplo, em nosso programa de planejamento da viagem, algum cálculo pode acabar usando uma variável que assume o valor zero no denominador de uma expressão contida em uma função.

Bibliografia inicial:
(PCC, Ch10, 192-200)

PRÉ-DISCUSSÃO

Objetivos de aprendizagem:

- Aprender a ler mensagens de erro geradas pelo Python
- Utilizar os objetos criados por essas mensagens de erro (*exceptions*) para criar códigos que contornem os erros, evitando que a execução do programa termine.

Ao final da pré:

- O aluno deve entender a necessidade de antecipar problemas que causem erros na execução do código, e consultar a bibliografia para aprender como implementar soluções para isso no Python.

PÓS-DISCUSSÃO

Elementos esperados na pós discussão:

- Mensagens de erro contém informações fundamentais para entender o que eventualmente deu errado em alguma parte do código
- O comando *traceback()* aplicado a uma função ajuda a identificar a natureza de erros de execução
- Erros criam objetos da classe *exception*, que são considerados em blocos de código definidos nos comandos *try:*, *except:* e *else:*

Exemplo:

```
def calcula_tempo_estimado(distancia_total, velocidade_media):
    try:
        tempo_estimado = distancia_total / velocidade_media
    except ZeroDivisionError:
        return "Velocidade media não pode ser zero!"
    except TypeError:
        return "distancia_total e velocidade_media precisam ser
números!"
    else:
        return tempo_estimado
```

25. Armazenando informações estruturadas

Fonte: elaborado pelo autor com uso de imagine.art.

Na medida em que o programa do planejamento da viagem vai se desenvolvendo, mais e mais informações são utilizadas para guiar as escolhas que serão feitas. Já vimos que informações relativamente simples, armazenadas em variáveis no código, podem ser armazenadas em arquivos externos e depois recuperadas. No caso de estruturas de dados mais complexas, como *dictionaries*, seria desejável uma forma equivalente de armazenamento externo. Se obedecido algum padrão pré-estabelecido para armazenar essas informações em arquivos externos, outros programas, inclusive em linguagens diferentes do Python, podem acessar esses valores.

Bibliografia inicial:
(PCC, Ch10, 201-204)

PRÉ-DISCUSSÃO

Objetivos de aprendizagem:

- Entender o formato JSON de armazenamento de informações contidas em estruturas de dados utilizadas em programas do Python.

Ao final da pré:

- O aluno deve entender a necessidade e conveniência do armazenamento externo de informações, em formato mais flexível que o dos arquivos de texto simples vistos

anteriormente, e buscar na bibliografia indicado os códigos de funções necessários para a tarefa.

PÓS-DISCUSSÃO

Elementos esperados na pós discussão:

- Códigos para o armazenamento de um conjunto de informações usando os comandos *Path()* e *json.dumps()*
- Códigos para a leitura de arquivos contendo informações na forma de estruturas de dados usando os comandos *Path()* e *json.loads()*
- Discussão sobre como o padrão JSON gera arquivos que podem ser lidos por outras linguagens de programação.

Exemplo:

Imagine que o programa organizou informações sobre um conjunto de cidades na forma de um dicionário “cidades_para_escolher”:

```
cidades_para_escolher = {'Itaitinga': {'custo_estimado': 624, 'tempo_viagem': 10, 'atracoes': ['Parque', 'Praia', 'Comida regional']}, 'Horizonte': {'custo_estimado': 261, 'tempo_viagem': 7, 'atracoes': ['Parque']}, 'Pacajus': {'custo_estimado': 340, 'tempo_viagem': 8, 'atracoes': ['Museu', 'Comida regional', 'Artesanato', 'Parque']}, 'Chorozinho': {'custo_estimado': 414, 'tempo_viagem': 7, 'atracoes': ['Comida regional', 'Museu', 'Praia', 'Parque', 'Artesanato']}, 'Russas': {'custo_estimado': 115, 'tempo_viagem': 1, 'atracoes': ['Comida regional', 'Praia', 'Parque']}, 'Limoeiro do Norte': {'custo_estimado': 879, 'tempo_viagem': 6, 'atracoes': ['Comida regional', 'Museu']}, 'Quixeré': {'custo_estimado': 467, 'tempo_viagem': 6, 'atracoes': ['Parque']}, 'Icó': {'custo_estimado': 629, 'tempo_viagem': 7, 'atracoes': ['Parque', 'Praia', 'Museu']}, 'Salgueiro': {'custo_estimado': 318, 'tempo_viagem': 5, 'atracoes': ['Parque', 'Praia', 'Artesanato', 'Comida regional', 'Museu']}, 'Serra Talhada': {'custo_estimado': 461, 'tempo_viagem': 7, 'atracoes': ['Artesanato', 'Parque', 'Praia', 'Comida regional']}, 'Arcoverde': {'custo_estimado': 623, 'tempo_viagem': 4, 'atracoes': ['Comida regional', 'Museu']}, 'Caruaru': {'custo_estimado': 315, 'tempo_viagem': 6, 'atracoes': ['Comida regional', 'Parque']}, 'Gravatá': {'custo_estimado': 450, 'tempo_viagem': 7, 'atracoes': ['Museu', 'Artesanato']}}
```

Para armazenar as informações contidas nesse dicionário em um arquivo Json:

```
import json

with open('cidades_info.json', 'w') as json_file:
    json.dump(cidades_info, json_file)
```

Para recuperar as informações desse arquivo dentro de um programa:

```
import json

with open('cidades_info.json', 'r') as json_file:
    cidades_info = json.load(json_file)

print(cidades_info)
```

26. Antecipando problemas

Fonte: elaborado pelo autor com uso de imagine.art.

Como vimos anteriormente, podemos usar *exceptions* para lidar com situações imprevistas durante a execução de programas. Antes disso, podemos tentar antecipar problemas escrevendo códigos de teste para funções e classes definidas por nós. Por exemplo, nosso programa de planejamento de viagem pode em dado momento solicitar informações dos usuários tais como o montante máximo que cada membro do grupo estaria disposto a gastar na viagem. O programa espera por uma informação numérica, mas um usuário pode entrar com uma informação do tipo "R$ 1500", que automaticamente seria considerada um *string*, e geraria uma mensagem de erro se o programa tentasse converte-la diretamente para uma variável numérica.

Bibliografia inicial:
(PCC, Ch11)

PRÉ-DISCUSSÃO

Objetivos de aprendizagem:

- Aprender a usar o pacote *pytest* para testar códigos de funções e classes
- Uma vez que *pytest* não faz parta da instalação padrão do Python, aprender a instalar pacotes usando o aplicativo *pip*

Ao final da pré:

- O aluno deve entender a necessidade de testar funções e classes definidas em seus códigos, e antecipar que existem ferramentas dedicadas para isso disponíveis para o Python, às quais ele será introduzido na leitura da bibliografia sugerida.

PÓS-DISCUSSÃO

Elementos esperados na pós discussão:

- Como instalar e atualizar o pacote *pip*
- Como usar o *pip* para instalar o módulo *pytest*
- Acessar as funções do *pytest* para testar:
  - Funções
  - Classes
- Como proceder em casos onde os testes revelam problemas no código

Exemplo:

Supondo que algumas funções de cálculos necessários para o planejamento da viagem (como "calcula_tempo_estimado" vista anteriormente) foram agrupadas no "modulo_calculos":

```
import pytest

from modulo_calculos import calcula_tempo_estimado

def test_calculate_estimated_time():
    assert calculate_estimated_time(100, 10) == 10
    assert calculate_estimated_time(0, 10) == 0
    assert calculate_estimated_time(100, 0) == "Velocidade média não
pode ser zero!"
    assert calculate_estimated_time(100, 'ten') == "Todos inputs dessa
função precisam ser numéricos!"
```

Agora supondo que a classe "CarroEletrico" vista anteriormente foi armazenada em um modulo "modulo_carros":

```
import pytest

from modulo_carros import CarroEletrico

def test_ CarroEletrico ():
    meu_carro = CarroEletrico ()

    meu_carro.carregar()
    assert meu_carro.battery_level == 100

    meu_carro.dirigir(50)
    assert meu_carro.battery_level == 50
```

27. Ajudando a financiar a viagem

Fonte: elaborado pelo autor com uso de imagine.art.

Seu programa para planejamento da viagem foi incorporando várias funções e informações, e parece estar funcionando bem. Você recebe a sugestão de disponibilizar seu programa para outros grupos de usuários, dispostos inclusive a pagar algum valor para sua utilização.

Bibliografia inicial:
(PCC, Ch18)

PRÉ-DISCUSSÃO

Objetivos de aprendizagem:

- Introduzir ao aluno ferramentas como o Django para disponibilizar programas na forma de aplicativos rodando em sites da Internet.

Ao final da pré:

- O aluno deve imaginar diferentes maneiras de tornar seu programa de planejamento de viagem disponível para outros usuários. Uma alternativa, de criar um Web App, é apresentada na bibliografia recomendada.

PÓS-DISCUSSÃO

Elementos esperados na pós discussão:

- Códigos escritos em Python podem ser acessados em sites da Internet, colhendo informação de usuários e retornando resultados que os auxiliem em decisões.
- Páginas dinâmicas em sites na Internet são uma combinação de códigos apresentando informações na tela com códigos processando informações fornecidas pelos usuários juntamente com informações já existentes. O pacote *Django* auxilia implementar essas ideias programaticamente no Python.

Exemplo:

1) Criar um projeto de Django:

```
django-admin startproject meu_projeto
```

2) Navegar para o diretório escolhido e criar um novo app de Django:

```
cd meu_projeto
python manage.py startapp myapp
```

3) No arquivo "views.py" do seu app, criar um "view" que utilize a função "calcula_tempo_estimado":

```
from django.http import JsonResponse
from .utils import calcula_tempo_estimado

def tempo_estimado_view(request):
    distancia_total = request.GET.get('distancia_total', 0)
    velocidade_media = request.GET.get('velocidade_media', 1)
    tempo_estimado = calculate_estimated_time(float(distancia_total),
float(velocidade_media))
    return JsonResponse({'tempo_estimado': tempo_estimado})
```

4) No arquivo "urls.py" do seu app, criar uma rota de URL mapeando o "view":

```
from django.urls import path
from .views import tempo_estimado_view

urlpatterns = [
    path('tempo-estimado/', tempo_estimado_view),
]
```

5) Finalmente, incluir o URL do app no arquivo de configuração de URL do projeto ("meu_projeto/urls.py"):

```
from django.urls import include, path

urlpatterns = [
    path('myapp/', include('myapp.urls')),
]
```

Agora, o servidor de desenvolvimento para o Django pode ser iniciado pelo comando (no prompt do sistema):

```
Python manage.py runserver
```

e a função calcula tempo estimado pode ser acessada em qualquer navegador de web por meio do endereço:

http://localhost:8000/myapp/tempo-estimado/?distancia_total=100&velocidade_media=80

mudando os valores dos parâmetros “distancia_total” e “velocidade_media” dentro do próprio endereço temos estimativas para diferentes valores.