O que vamos aprender nessa aula:

1. Modularização com arquivos

Refrescando um pouco a sua memória, há poucas aulas atrás te ensinamos uma forma de modularizar seu código, por meio das funções. Entretanto, com o crescimento de um projeto, modularizar apenas por meio de funções pode não ser suficiente para ter um código esteticamente agradável e, principalmente, legível.

A alternativa mais simples e popular para resolver esse problema é a separação do programa em **arquivos.** Com a separação inteligente do código em arquivos, podemos agrupar funções e classes relacionadas à resolução de problemas parecidos e, apenas referenciando um arquivo, usá-las em um programa escrito em outro arquivo. Desta forma, podemos **testar** os módulos de forma independente, ou seja, mais eficientemente. Como arquivos podem ser “importados” por outros arquivos e, consequentemente, funções e classes também são "importadas", estamos permitindo outra característica desejável em progração, a saber, o **reuso** de código.

Na teoria, parece um pouco complicado e muito abstrato, porém, na prática, veremos que é bem simples!

1.1. Separação em arquivos

Abaixo temos um programa main.py que recebe um número inteiro e exibe algumas características relacionadas ao autor deste exemplo. Perceba que já temos algumas funções modularizando o código:

In [ ]:

```
#main.py

#funções de matemática
def paridade(numero):
    if numero%2 == 0:
        return "par"
    else:
        return "ímpar"

def fatorial(numero):
    fatorial = 1

    while (numero > 1):
        fatorial = fatorial * numero
        numero = numero - 1

    return fatorial

#funções de datas
def eh_mes_do_aniversario(mes):
    if(mes == 11):
        return "É o mês do aniversário do criador do exemplo!"

    return "Não é o mês que eu queria :("

def eh_dia_do_aniversario(dia):
    if(dia == 1):
        return "É o dia do aniversário do criador do exemplo!"

    return "Não é o dia que eu queria :("

def eh_ano_de_nascimento(ano):
    if(ano == 1999):
        return "É o ano do nascimento do criador do exemplo!"

    return "Não é o ano que eu queria :("

#programa principal
numero = int(input())
print("O número é " + paridade(numero))
print("O fatorial é " + str(fatorial(numero)))
print(eh_dia_do_aniversario(numero))
print(eh_mes_do_aniversario(numero))
print(eh_ano_de_nascimento(numero))
```

Com o programa feito, podemos tentar agrupar as funções de forma inteligente. Preferimos separar as duas primeiras funções (relacionadas a matemáticas) e as últimas três que lidam com data.

Já com as categorias separadas, faremos um arquivo para cada uma delas e manteremos o programa principal com o mesmo nome.

In [ ]:

```
#minhamatematica.py

def paridade(numero):
    if numero%2 == 0:
        return "par"
    else:
        return "ímpar"

def fatorial(numero):
    fatorial = 1

    while (numero > 1):
        fatorial = fatorial * numero
        numero = numero - 1

    return fatorial
```

In [ ]:

```
#datasdocriador.py

def eh_mes_do_aniversario(mes):
    if(mes == 11):
        return "É o mês do aniversário do criador do exemplo!"

    return "Não é o mês que eu queria :("

def eh_dia_do_aniversario(dia):
    if(dia == 1):
        return "É o dia do aniversário do criador do exemplo!"

    return "Não é o dia que eu queria :("

def eh_ano_de_nascimento(ano):
    if(ano == 1999):
        return "É o ano do nascimento do criador do exemplo!"

    return "Não é o ano que eu queria :("
```

In [ ]:

```
#main.py

numero = int(input())
print("O número é " + paridade(numero))
print("O fatorial é " + str(fatorial(numero)))
print(eh_dia_do_aniversario(numero))
print(eh_mes_do_aniversario(numero))
print(eh_ano_de_nascimento(numero))
```

Agora com os arquivos separados, ainda não conseguimos utilizar as funções no programa principal. Como faremos isso?

1.2. Importando arquivos

Atenção: Os códigos abaixo só funcionarão no Jupyter pois os arquivos minhamatematica.py e datasdocriador.py estão na mesma pasta do notebook! Existem maneiras de importar arquivos em pastas diferentes, porém recomendamos que, por enquanto, salve sempre os arquivos na mesma pasta.

Figura 01: Diretório onde se encontra essa aula.

Com as demonstrações abaixo você deve perceber o quão organizado (e elegante) fica o seu programa ao criar um "arquivo principal" (exemplificado pelo nome de `main.py`) e isolar as funções em arquivos separados. Com o atual conhecimento, provavelmente criaria todas as funções utilizadas em um grande arquivo (o que faria do seu programa um pesadelo para depuração), além de reimplementá-las todas as vezes que precisasse (um grande retrabalho).

Agora que falamos um pouco sobre a importância de conceito, vamos cobrir as maneiras mais utilizadas para importar arquivos nas próximas seções.

1.2.1. Importando o arquivo completo

Para importar arquivos completos, utilizamos a palavra reservada **import**.

In [ ]:

```
#main.py

import minhamatematica
import datasdocriador

numero = int(input())
print("O número é " + minhamatematica.paridade(numero))
print("O fatorial é " + str(minhamatematica.fatorial(numero)))
print(datasdocriador.eh_dia_do_aniversario(numero))
print(datasdocriador.eh_mes_do_aniversario(numero))
print(datasdocriador.eh_ano_de_nascimento(numero))
```

Dessa maneira, basta importarmos os arquivos com seus nomes e sem a extensão ".py". Para utilizar as funções, devemos obedecer à sintaxe `nomedoarquivo.funçãoutilizada`. A importação carrega todas as funções do arquivo importado para o programa principal.

1.2.2. Importando módulos

Chamamos aquilo que importamos de outro arquivo de módulo, seja uma função, classe ou o arquivo inteiro. Não muito diferente da situação anterior, para importar funções e classes, devemos continuar utilizando a palavra reservada `import`, porém precedida de `from`.

In [ ]:

```
#main.py

from minhamatematica import paridade
from datasdocriador import eh_dia_do_aniversario, eh_mes_do_aniversario

numero = int(input())
print("O número é " + paridade(numero))
print(eh_dia_do_aniversario(numero))
print(eh_mes_do_aniversario(numero))
```

Nessa situação, retiramos algumas funcionalidades do programa principal para melhor aproveitarmos o conceito apresentado. Traduzindo `from arquivo import função` literalmente para o português, teríamos algo como `de arquivo importa função`.

Com isso, apenas carregamos para o programa principal as funções que serão utilizadas, tendo um programa mais leve e legível.

Essa sintaxe também pode ser utilizada para importar todas as funções de um arquivo, sem a necessidade de escrever todos os nomes. Para isso, basta escrever `from arquivo import *`. Apesar de funcionar, é pouco usado pela comunidade geral de Python, principalmente por perder o controle dos nomes de funções que já são utilizados.

Utiliza-se esta mesma sintaxe para importar classes de outros arquivos. Utilizando o arquivo usuario.py, que possui uma classe Usuario, observe o exemplo abaixo:

In [ ]:

```
from usuario import Usuario

usuario1 = Usuario("Chapolin", 20, "01/11")

print(usuario1.maiorDeIdade())
usuario1.imprimeAniversario()
```

O mais importante de ser absorvido deste caso é que basta importar a classe Usuario que automaticamente todos os seus métodos tornam-se utilizáveis no programa.

1.2.3. Renomeando a referência ao arquivo

No código abaixo você pode ver uma das maneiras mais utilizadas pela comunidade de Python, principalmente para importação de bibliotecas (logo mais você entenderá e conhecerá algumas!). A ideia aqui é utilizar a palavra reservada `as` para renomear o arquivo importado com o `import`:

In [ ]:

```
#main.py

import minhamatematica as matematica
import datasdocriador as datas

numero = int(input())
print("O número é " + matematica.paridade(numero))
print("O fatorial é " + str(matematica.fatorial(numero)))
print(datas.eh_dia_do_aniversario(numero))
print(datas.eh_mes_do_aniversario(numero))
print(datas.eh_ano_de_nascimento(numero))
```

Utilizando a importação dessa maneira, basicamente se replica o que foi feito na seção 1.2.1, porém abre espaço para um código mais enxuto e possivelmente mais organizado.

1.2.4. Renomeando a referência à função

A última maneira que mostraremos de importação nos permite escolher o nome para a função que será utilizada no programa (também utilizando a palavra reservada `as`:

In [ ]:

```
#main.py

from minhamatematica import paridade as parouimpar
from datasdocriador import eh_dia_do_aniversario as dia

numero = int(input())
print("O número é " + parouimpar(numero))
print(dia(numero))
```

A sintaxe exerce o mesmo papel especificado na seção 1.2.2, porém novamente proporcionando maior personalização do código.

1.3. Considerações gerais

Ao tentar dividir seu programa em arquivos no seu próprio computador, pode surgir a dúvida de como executar um programa de múltiplos arquivos. Como já fazia antes, basta executar o programa principal (que nesse caso é o main.py) que todos os outros já serão automaticamente acionados.

Recomendamos que tire um tempo para tentar executar esse programa no seu próprio computador, para que entenda realmente como fará em seus próprios programas futuramente. Basta copiar os códigos dos dois arquivos complementares e colar em dois novos arquivos, e logo depois programar um arquivo principal que os utilizará. Lembre-se de salvar todos os arquivos na **mesma pasta!**

A partir daqui, é importante que considere separar também suas classes em arquivos separados. Para importá-las, basta substituir as funções pelo nome da classe a ser utilizada. O método mais usual de utilizá-las é o descrito na seção 1.2.2. Todos os métodos da classe (as funções que são utilizadas na classe) são automaticamente importadas ao importar a classe.

Se lembra da função `__main__` , lá da aula 7 de funções? Pouco falamos dela ao longo do curso, então tudo bem caso sua resposta seja não. Como estamos deixando o código consideravelmente mais elegante não poderíamos deixar de citá-la por aqui: é uma excelente prática em Python que a utilize no arquivo principal, mas como viu anteriormente, não é obrigatória.

2. Bibliotecas

Já usamos bibliotecas antes! Vimos um pouco da biblioteca `math` no início do curso para fazer contas mais complexas, mas, afinal, o que são bibliotecas e para que servem?

Como programadores, frequentemente lidamos com problemas bastante parecidos. Para facilitar resolução destes tipos de problemas comuns, temos as bibliotecas. Uma biblioteca pode ser formada por um ou mais módulos que definem um conjunto de funções, variáveis e tipos de dados para serem utilizados em um determinado contexto. Por exemplo, com a biblioteca *math* , podemos resolver vários problemas de matemática.

Quando baixamos *Python* , também baixamos a `biblioteca padrão` . Na biblioteca padrão temos um conjunto de bibliotecas que podem ser utilizadas sem a necessidade baixá-las explicitamente. Podemos encontrar as bibliotecas, ou módulos, que fazem parte dela aqui (https://docs.python.org/3/py-modindex.html).

Alguns exemplos de bibliotecas legais e utéis são:

- os;
- sys;
- math;
- numpy;
- matplotlib.pyplot;
- turtle;
- pygame.

Vamos ver um pouco sobre algumas delas depois!

2.1. Utilizando as bibliotecas

O uso de bibliotecas é bem semelhante ao uso de arquivos. Da mesma forma que utilizamos as palavras reservadas `import` , `from` e `as` quando queremos importar um arquivo ou seus módulos, utilizamos para importar uma biblioteca. Vamos ver abaixo alguns exemplos:

In [24]:

```
import random

numero_da_sorte = random.randint(1, 100)
print(f"O seu número da sorte é {numero_da_sorte}!")
```

```
O seu número da sorte é 80!
```

In [20]:

```
from math import pi

print("Pi:",pi)
```

```
Pi: 3.141592653589793
```

In [16]:

```
import numpy as np

x = pi
y = 2
seno = np.sin(pi/y)
print(f"O seno de (pi / 2) é {seno}.")
```

```
O seno de (pi / 2) é 1.0.
```

2.2. Instalando as bibliotecas

Nós vimos que conseguimos utilizar um monte de bibliotecas, mas e se a biblioteca que queremos baixar não faz parte da `biblioteca padrão` de Python? Vamos ter que baixá-las!

Quando baixamos o `Anaconda` , baixamos também uma série de bibliotecas (além de outros pacotes) que você pode verificar aqui (https://docs.anaconda.com/anaconda/packages/old-pkg-lists/4.3.1/py35/). Se a biblioteca que você quer não se encontra nessa lista, você pode baixar pelo `Anaconda Navigator` ou `Anaconda Prompt` no Windows. Para usuários de Linux ou macOS, o terminal está disponível. Usando o Navigator, você pode procurar a biblioteca desejada. Se você for usar o Anaconda Prompt ou o terminal, basta usar o seguinte comando:

```
conda install noma_da_biblioteca
```

Você também pode baixar versões específicas de uma mesma biblioteca. Para mais informações sobre como fazer isso usando o conda, é só verificar a documentação do Anaconda aqui (https://docs.anaconda.com/anaconda/user-guide/tasks/install-packages/).

Uma outra forma de baixar é utiliando o comando `pip` ! Basta ir no terminal do Linux ou macOS e usar o seguinte comando:

```
pip install nome_da_biblioteca
```

(O *pip* não está disponível nativamente para o Windows).

E, voilà! Você agora pode utilizar a biblioteca. Como há muito a adicionar a este assunto, te deixamos com a documentação (https://docs.python.org/3/installing/index.html) do *Python* que traz bastantes informações sobre!

---

3. Manipulando matrizes com Numpy

Sabemos que Python possui suporte "nativo" a "matrizes", mas isso não funciona muito bem. O suporte, na verdade, não é nativo pois as matrizes em Python são definidas como listas de listas. Os problemas surgem quando queremos multiplicar uma matriz por um vetor, ou uma matriz por outra matriz. Nestes casos, dois problemas principais podem ocorrer: performance e praticidade. A performance é afetada pelo uso excessivo de loops; quanto à praticidade, não é nada amigável multiplicar "matrizes" (ou seja, listas de listas) em Python.

Para contornar estes problemas, utilizaremos a bibloteca `numpy` que oferece inúmeras ferramentas matemáticas, inclusive, oferece apoio ao desenvolvimento de matrizes com sintaxe bastante amigável. `numpy` ainda é otimizada para operações com matrizes e, portanto, performance não será mais um problema.

Para aqueles que não conhecem um vetor, iremos definí-lo como algo muito semelhante a uma lista. Pense num vetor $A = \begin{bmatrix} 1 & 2 & 3 & 4 \end{bmatrix}$. Muito semelhantes às listas que já conhecemos, certo? Se quisermos acessar o elemento de valor 2 devemos acessar o segundo índice do vetor, ou seja, `A[1]`, exatamente como fazemos com listas.

Uma matriz seria um vetor com vários vetores. Imagine agora o seguinte vetor:

$$M = \begin{bmatrix} M_{[0]} & M_{[1]} & M_{[2]} & M_{[3]} \end{bmatrix}$$

Se `M[0]`, `M[1]`, `M[2]` e `M[3]` forem vetores, `M` é uma matriz. Imagine `M[0] = [1, 2, 3, 4]`, `M[1] = [5, 6, 7, 8]`, `M[2] = [9, 10, 11, 12]` e `M[3] = [13, 14, 15, 16]`. Temos, nesse caso:

$$M = \begin{bmatrix} 1 & 5 & 9 & 13 \\ 2 & 6 & 10 & 14 \\ 3 & 7 & 11 & 15 \\ 4 & 8 & 12 & 16 \end{bmatrix}$$

Se estivermos interessados no elemento que vale 10 devemos utilizar a notação `M[2][1]` pois queremos o segundo elemento do terceiro vetor. De uma forma geral, temos a notação `M[i][j]` onde `i` é a linha e `j`, a coluna.

Observação: durante o ensino médio é ensinado que o primeiro índice de uma matriz é 1, aqui, entretanto, utilizaremos o primeiro índice como 0 para simplificar o aprendizado da biblioteca.

O primeiro passo para usar tal biblioteca é importá-la:

In [3]:

```
import numpy as np
```

Nosso próximo passo é criar uma matriz. Existem diversas formas de fazer isso e veremos algumas delas a seguir.

3.1. Criando matrizes

Vamos dividir essa seção em casos; cada caso representa uma possível situação em seu código.

Matriz nula e similares:

Caso queira criar uma matriz nula de tamanho $m \times n$, basta chamar a função `np.zeros(m, n)` [ref]

In [ ]:

```
# Criando matriz nula de tamanho 3x4
matriz_nula = np.zeros((3, 4))
print(matriz_nula)
```

Repare que os zeros da nossa matriz são do tipo `float`; podemos definir o tipo dos nossos dados por meio do parâmetro `dtype`. Desse modo:

In [ ]:

```
matriz_nula = np.zeros((3, 4), dtype=int)
print(matriz_nula)
```

Podemos também criar uma matriz cheia de `1`s chamando a função `np.ones` [ref] (https://docs.scipy.org/doc/numpy-1.14.1/reference/generated/numpy.ones.html)

In [ ]:

```
matriz_um = np.ones((3, 4), dtype=int)
print(matriz_um)
```

NOTA: Assim como na matemática, em Python, os nomes de variáveis que representam matrizes são normalmente uma letra maiúscula. `matriz_nula` e `matriz_um` não são bons nomes para matrizes e, portanto, a partir de agora utilizaremos nomes como `A`, `B`, `M`, `I`, `X`, `Y` e `Z`.

A partir das matrizes que já sabemos criar, podemos criar muitas outras matrizes através de operações básicas de matrizes, como **multiplicação por escalar**, **soma por escalar** e **soma de matrizes**. Os trechos de código a seguir mostram exemplos de manipulação de matrizes.

In [ ]:

```
# Obtendo uma matriz 3x3 cheia de 3:
A = np.ones((3, 3), dtype=int) * 3
print(A)
```

In [ ]:

```
# Obtendo uma matriz 3x3 cheia de 2:
B = np.zeros((3, 3), dtype=int) + 2
print(B)
```

In [ ]:

```
# Obtendo uma matriz 3x3 cheia de 5:
C = A + B
print(C)
```

Perceba como é simples manipular matrizes e como nos aproximamos muito da matemática com `numpy`.

Observação 1: na matemática, quando dizemos que somamos $B + 2$, onde B é uma matrix 2×2, estamos na realidade fazendo

$$B_{2\times 2} + \begin{bmatrix} 2 & 2 \\ 2 & 2 \end{bmatrix} = \begin{bmatrix} B_{[0][0]} + 2 & B_{[0][1]} + 2 \\ B_{[1][0]} + 2 & B_{[1][1]} + 2 \end{bmatrix}$$

e é exatamente isso que ocorre em nosso código. Pode não parecer útil no momento mas, quando trabalhamos com Estatística e *Machine Learning*, tal comportamento se torna um grande facilitador, uma vez que não precisamos recorrer a *loops*.

Observação 2: chamaremos qualquer *array* de matriz ao longo dessa aula para melhorar a didática; não necessariamente estaremos nos referindo a *arrays* bidimensionais ao falarmos de matrizes.

Matriz de valores aleatórios:

Para valores aleatórios utilizamos o módulo `np.random`, mais especificamente a função `np.random.randn` [ref] (https://docs.scipy.org/doc/numpy-1.15.1/reference/generated/numpy.random.randn.html).

In [ ]:

```
# Criando um valor aleatório:
x = np.random.randn()
print(x)
```

In [ ]:

```
# Criando uma matriz 2x2 de valores aleatórios:
X = np.random.randn(2, 2)
print(X)
```

Note que, diferentemente das funções `np.ones` e `np.zeros`, a função `np.random.randn` toma como argumentos os próprios valores do tamanho da matriz e não uma tupla.

Matriz de valores definidos:

Para criar uma matriz com valores definidos, como $A = \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix}$, basta chamar o construtor `np.array` [ref] (https://numpy.org/doc/stable/reference/generated/numpy.array.html).

In [ ]:

```
# Dessa maneira convertemos uma lista para np.array:
A = np.array([[1, 2], [3, 4]], dtype=int)
print(A)
```

Exercício 1:

Crie a matriz $M = \begin{bmatrix} 0 & 1 \\ 2 & 0 \end{bmatrix}$ e obtenha o valor presente no primeiro índice do segundo vetor.

Dica: para pegar o elemento `i` de um vetor `v` você deve chamar por `v[i]`. Para matrizes você pode pegar o elemento $V_{[i][j]}$ com `V[i][j]` ou `V[i,j]`.

In [ ]:

```
# Crie a matriz aqui: (Aproximadamente 1 linha)
M =
# FIM DA CRIAÇÃO DA MATRIZ

# Obtenha o primeiro elemento do segundo vetor de M: (Aproximadamente 1 linha)
e =
# FIM DA CHAMADA DO ELEMENTO

assert(e == 1)

print("Parabéns, você fez tudo certo!")
```

Note que qualquer matriz/vetor criado utilizando `numpy` pertence à classe `np.ndarray`:

In [ ]:

```
type(np.ones((2, 2)))
```

In [ ]:

```
type(np.zeros((2, 2)))
```

In [ ]:

```
type(np.array([[2, 2], [2, 2]]))
```

In [ ]:

```
type(np.array([1, 2, 3, 4]))
```

Essa característica nos permite utilizar métodos como `dot`, `T` e outros que veremos a seguir.

3.2. Visualizando dados com pyplot

Já que `numpy` nos proporciona tantas ferramentas matemáticas poderosas, vamos tentar *plotar* o gráfico do $sen(T)$ que calculamos anteriormente. Para isso utilizaremos outra biblioteca chamada `pyplot`. Vamos importá-la da forma como ensinamos.

In [1]:

```
import matplotlib.pyplot as plt
```

Para *plotar* o gráfico, basta chamarmos a função `plt.plot` [ref] (https://matplotlib.org/3.3.0/api/_as_gen/matplotlib.pyplot.plot.html). Essa função recebe `x`, `y` e muitos outros argumentos que podem ser encontrados na documentação (é possível combiná-los para obter inúmeros formatos de plot diferentes). `x` refere-se a uma `list`, `tuple` ou `np.ndarray` contendo os valores de x. `y` é um arranjo, assim como `x`, mas contém os valores de $f(x)$. Perceba que `len(x)` deve ser igual a `len(y)`.

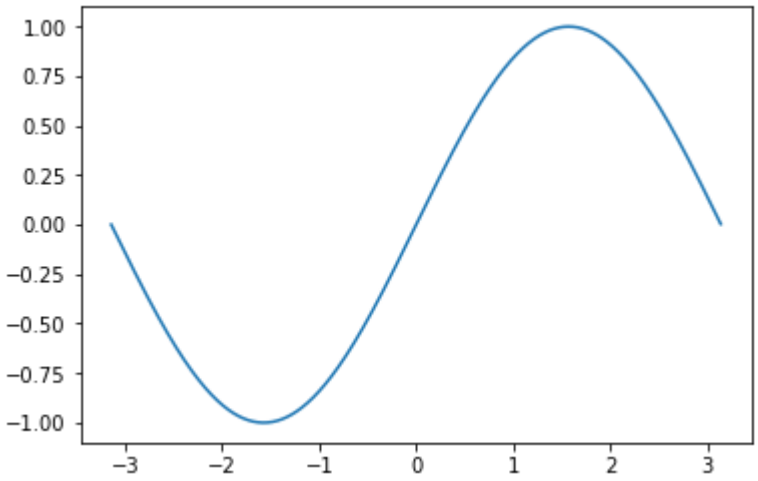
Tomando a função $f(x) = x^2$ como exemplo: `x[0] == 2`, portanto `y[0] == 4` já que `y[i] == x[i] ** 2` para todo índice `i`.

In [4]:

```
X = np.arange(-np.pi, np.pi, 0.01) # Gerando um vetor de valores ao longo do eixo X.
Y = np.sin(X) # Aplicando os valores na função e gerando uma valor com os resultados.
plt.plot(X, Y) # Plotando, conforme explicado acima.
```

Out[4]:

```
[<matplotlib.lines.Line2D at 0x1122b5588>]
```



Se quisermos, por exemplo, fazer um gráfico do valor da gasolina a cada ano também podemos fazer. Suponha que temos os dados:

Preço do litro da gasolina (em reais)	Ano
10.10	2015
12.30	2016
13.50	2017
15.20	2018
18.70	2019

Basta passarmos esses dados para variáveis, correto?

In [6]:

```
anos = [2015, 2016, 2017, 2018, 2019]
preco = [10.10, 12.30, 13.50, 15.20, 18.70]
```

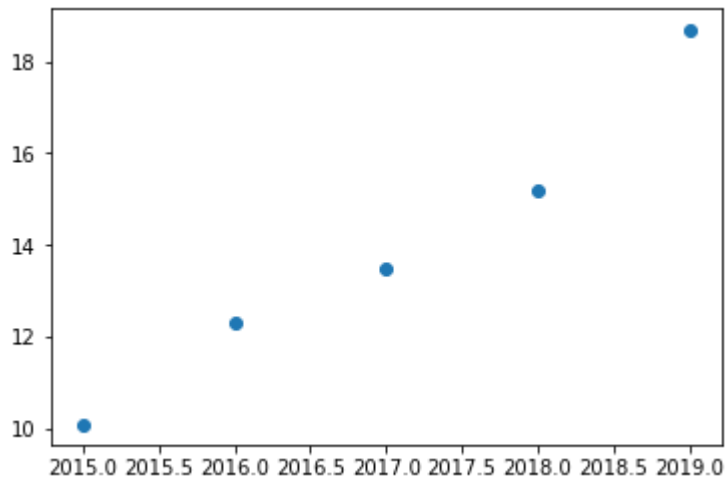
A partir daí *plotamos* os dados com `plt.scatter` . A diferença entre `plot` e `scatter` é que `scatter` exibe pontos no gráfico e `plot` exibe um gráfico contínuo.

In [7]:

```
plt.scatter(anos, preco)
```

Out[7]:

```
<matplotlib.collections.PathCollection at 0x1123bdcc0>
```



Exercício 2:

Visualize a função $f(x) = y$ onde $y = 2x + 5$ utilizando `plt` .

Dica: Para criar uma função $y = ax + b$ com `numpy` fazemos `Y = a*X + b` onde `type(a)` e `type(b)` são `float` ou `int` .

In [ ]:

```
# Definindo X:
X = np.arange(0, 5)

# Definindo Y: (Aproximadamente 1 linha)
Y =
# FIM DA DEFINIÇÃO DE Y

# Plotando Y(X): (Aproximadamente 1 linha)

# FIM DO PLOT DE Y(X)
```

Exercício 3:

Visualize a função $f(x) = y$ onde $y = x^2 + 3x + 1$ utilizando `plt` .

Dica: Você pode fazer x^a em `numpy` com `X**a` .

In [ ]:

```
# Definindo X:
X = np.arange(-10, 10)

# Definindo Y: (Aproximadamente 1 linha)
Y =
# FIM DA DEFINIÇÃO DE Y

# Plotando Y(X): (Aproximadamente 1 linha)

# FIM DO PLOT DE Y(X)
```

Exercício 4:

A função `get_dataset` retorna um dataset de número acumulado de infectados por um virus a cada dia.

Utilizando `plt.scatter` exiba os valores de um conjunto de dados carregado pela função `get_dataset` já carregada na célula.

In [ ]:

```
from dados.pandemia import get_dataset

# Carregando conjunto de dados:
X, Y = get_dataset()

# Exibindo os dados: (Aproximadamente 1 linha)

# FIM DO SCATTER DOS DADOS
```

4. Desenhando em Python

Acabamos de ver uma biblioteca muito poderosa de Python, com altíssimo nível de aplicabilidade para programas com maior profundidade teórica. Agora veremos a biblioteca Turtle que, apesar do seu teor lúdico, vai nos ajudar a entender as proporções do poder das bibliotecas em Python.

4.1. Conceitos básicos

Referenciando os conceitos da aula anterior, a biblioteca funciona na sua maior parte com os métodos de duas classes centrais, a `TurtleScreen`, que representa o display no qual as imagens serão exibidas, e a `RawTurtle`, uma seta exibida nos desenhos que desenha no display.

Apesar de toda a programação girar em torno dessas classes, os conceitos de orientação a objetos podem ser dispensáveis. A biblioteca oferece uma "interface procedural" que contém funções de todos os métodos dessas duas classes, e sempre que uma função é chamada automaticamente são criados objetos das classes utilizadas. Na teoria esse conceito pode ser complicado, mas com alguns poucos exemplos torna-se intuitivo.

Para iniciar a demonstração da biblioteca, podemos fazer o seguinte código base:

In [8]:

```
from turtle import *

forward(0)

done()
```

Iniciamos com a importação da biblioteca e logo escrevemos o primeiro comando, que cria automaticamente um objeto TurtleScreen para exibir um display e outro objeto RawTurtle que renderiza uma seta apontando para a direita. Já o segundo comando nos permite fechar o display pelo "x" (se você esquecer dele pode ficar meio perdido :P). Observe o resultado abaixo:

Figura 02: Resultado do código acima.

A seta será o referencial do desenho e funciona analogamente como uma caneta encostando em um papel. De maneira básica, podemos movimentá-la para frente e para trás com forward() e backward(), respectivamente. Para usarmos essas funções, fornecemos um valor inteiro que representa o número de pixels a ser percorrido, seja para frente ou para trás. Veja o código exemplo abaixo:

In [10]:

```
from turtle import *

forward(250)

done()
```

Executando o código, obtemos uma reta de 250 pixels com a seta apontando para a direita, como mostra a imagem abaixo:

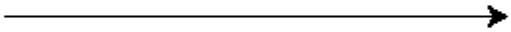


Figura 03: Traço de 250 pixels para a direita.

Para alterarmos a direção da nossa reta, podemos utilizar as funções `left()` e `right()`, que recebem um valor que representa um ângulo em graus em que a seta irá girar. Veja o código abaixo e perceba o resultado:

In [12]:

```
from turtle import *

forward(100)
left(90)
forward(100)

done()
```

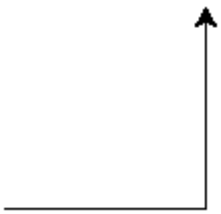


Figura 04: Traços resultantes do código acima.

Dessa vez temos mais de uma função no programa. A lógica das classes e objetos criados acaba sendo abstraída, de forma que os objetos necessários são criados apenas na primeira função e reutilizados em todas as outras seguintes.

Além das movimentações a partir dos ângulos e distâncias, a biblioteca fornece uma função muito mais simples para situações em que se deseja mover a seta para uma posição específica do display. Com `setposition()`, enxergamos o display como um plano cartesiano, onde a origem é o ponto central. A partir disso, podemos fornecer dois números que levam a seta até um ponto do display:

In [14]:

```
from turtle import *

#setposition(coordenada_x, coordenada_y)

setposition(100, 100)
setposition(-100, 100)
setposition(-100, -100)
setposition(100, -100)
setposition(100, 100)

done()
```

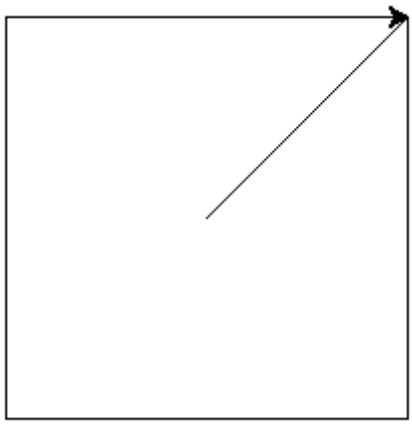


Figura 05: Figura formada utilizando setposition().

Lembra da analogia da caneta? Repare que em todas as movimentações a seta sempre traçou retas, como uma caneta presa em um papel. Para melhor controle do desenho, também temos as funções `penup()` e `pendown()`, que funcionam como "tirar a caneta do papel" e "prender a caneta no papel", respectivamente. Adaptando o exemplo anterior, conseguimos formar um quadrado perfeito centralizado:

In [5]:

```
from turtle import *

#após esse comando, as movimentações não formam mais desenhos
penup()
setposition(100, 100)
#com esse comando, as movimentações voltam a desenhar
pendown()

setposition(-100, 100)
setposition(-100, -100)
setposition(100, -100)
setposition(100, 100)

done()
```

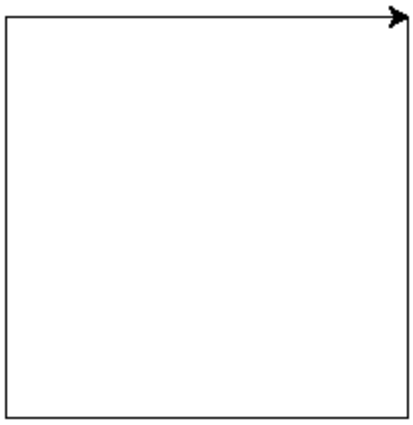


Figura 06: Quadrado com lado de 200 píxels.

Outros comandos interessantes nos permitem preencher a imagem, traçando uma reta do ponto em que o desenho se iniciou até o último ponto desenhado, preenchendo a parte interior. Para isso, basta sinalizarmos o início da parte a ser preenchida com `begin_fill()` e o fim com `end_fill()` :

In [16]:

```
from turtle import *

begin_fill()

forward(100)
left(90)
forward(100)

end_fill()

done()
```

Figura 07: Triângulo retângulo.

Como último comando básico podemos citar o `pos()` , que novamente faz uma analogia ao plano cartesiano e nos retorna uma tupla com a posição (x, y) da seta. Apesar de não ter representação gráfica nenhuma, essa função é interessante para condicionais e loops, nos permitindo fazer desenhos atrativos com poucas linhas.

In [ ]:

```
from turtle import *

forward(100)

print(pos())

left(90)
forward(100)

print(pos())

done()
```

4.2. Utilizando a biblioteca com loops

Agora que conhecemos os recursos básicos, podemos explorar um pouco mais alguns desenhos bem divertidos e com poucas linhas de código, utilizando principalmente loops.

Exemplo 1: Triângulo equilátero

Lembrando que a seta do desenho inicia-se apontando para a direita, basta escolhermos o tamanho do lado do triângulo e virar 120° para um dos lados e interromper o loop ao atingirmos novamente a posição central do display. Acompanhe o exemplo para ficar mais claro:

In [1]:

```
from turtle import *

# esse comando determina a velocidade de desenho como 10 (velocidade máxima)
speed(10)

begin_fill()

while True:
    forward(100) ## tamanho do lado do triangulo
    left(120) ## ângulo externo do triângulo

    coordenadas = pos()

    # verifica se a seta retornou ao ponto de origem
    if abs(coordenadas) < 1:
        break

end_fill()

done()
```

Figura 08: Triângulo equilátero.

O ângulo de 120° foi escolhido justamente por ser o ângulo externo do triângulo equilátero. Assim, para desenhar qualquer polígono regular basta escolher o tamanho e calcular o ângulo externo.

Exercício 5: Polígonos regulares

Faça um programa que recebe a quantidade de lados e desenhe o polígono regular correspondente de lado com tamanho de 100 pixels.

Fórmulas que podem te ajudar:

```
ângulo externo = 180° - ângulo interno
ângulo interno = [(lados - 2) * 180]/lados
```

In [ ]:

```
from turtle import *

lados = int(input("Número de lados (deve ser maior que 2): "))

#ache a formula para encontrar o angulo a partir da quantidade de lados
angulo = 0

speed(10)

begin_fill()

while True:
    forward(100)
    left(angulo)

    coordenadas = pos()

    if abs(coordenadas) < 1:
        break

end_fill()

done()
```

Exemplo 2: Cruzando linhas

Ao escolhermos um ângulo entre 180° e 270° para a esquerda (ou 90° e 180° para a direita), as linhas desenhadas se cruzarão. Dependendo do ângulo escolhido, pode-se formar imagens bem interessantes! Observe o exemplo abaixo:

In [1]:

```
from turtle import *

speed(10)

while True:
    forward(150)
    left(220)

    coordenadas = pos()

    if abs(coordenadas) < 1:
        break

done()
```

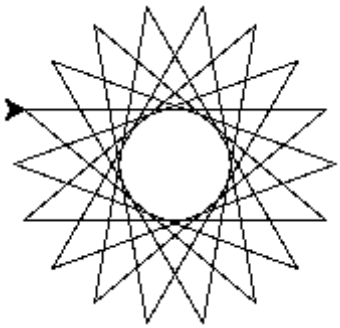


Figura 09: Resultado esperado pelo exemplo 2.

Exercício 6: Estrela

Cruze as linhas e selecione o ângulo que desenhará uma estrela de 5 pontas.

OBS: Para alterar as cores, utilize a função `color('red', 'blue')`, que desenha a linha em vermelho (red) e preenche a figura de azul (blue), e `bgcolor('yellow')` para o fundo de cor amarela.

In [7]:

```

```

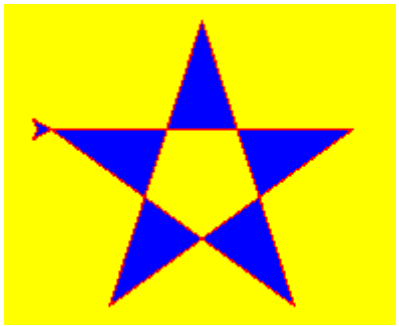


Figura 10: Estrela resultante do exercício 6.

{ mais a fundo }

Exercício 7: Espirais

Com os conhecimentos aprendidos até agora, desenvolva um programa que desenhe um triângulo espiral como o da figura 11.

In [21]:

```
```

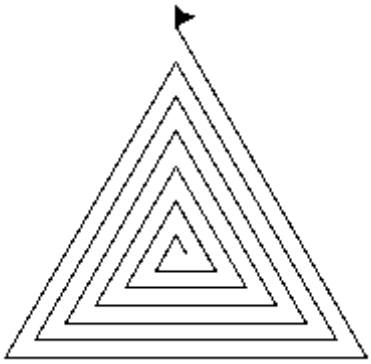


Figura 11: Espiral resultante do código acima.

Aproveite para mudar o ângulo, o número de iterações e utilizar os comandos `penup()` e `pendown()` para formar imagens diferentes (a estrela é bem bonita!).

4.3. Estude e se divirta!

É simplesmente impossível mostrar todas as funcionalidades que a biblioteca fornece em apenas uma aula, e para o escopo do Introcomp, não é interessante maior aprofundamento. Caso você tenha interesse em aprender mais sobre a `turtle` , basta acessar a documentação (https://docs.python.org/3.3/library/turtle.html). Estimule sua criatividade e faça suas próprias obras de arte :).

In [1]:

```
from turtle import *

speed(10)
pensize(10)

colors = ['blue', 'yellow', 'black', 'green', 'red']

x = -240
y = 0

for i in range(5):
    penup()
    setposition(x , y)
    pendown()

    color(colors[i])
    circle(100)

    x += 120

    if i % 2 == 0:
        y -= 120
    else:
        y += 120

done()
```

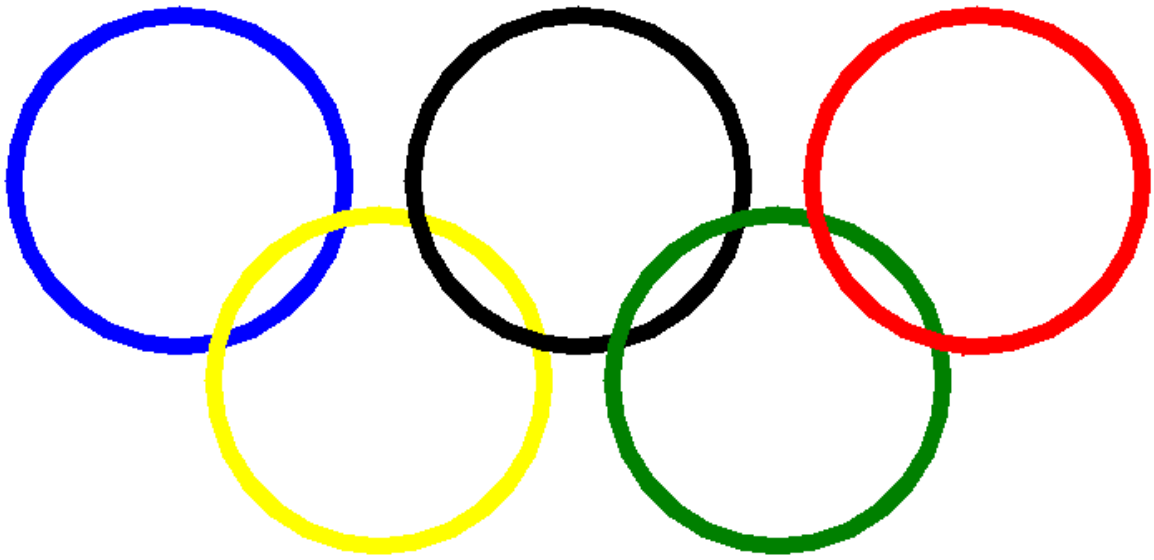


Figura 12: Símbolo das olimpíadas resultante do código acima.

5. O que pode dar errado?

5.1. Bibliotecas não instaladas

Imagine a seguinte situação: você e seu amigo estão fazendo uma tarefa em Python, tal tarefa requer o uso da biblioteca hipotética `libcomp`. Ao finalizar a parte dele da tarefa seu amigo o envia o código-fonte para que você possa fazer a sua parte, entretanto você não instalou a biblioteca `libcomp` e o interpretador exibiu uma mensagem de erro.

A situação apresentada é muito comum, principalmente quando trabalhamos com *scripts* e programas com muitas dependências. Uma maneira de resolver isso é criando um arquivo `requirementes.txt` contendo uma lista de dependências do seu programa, desse modo qualquer um que quiser usar sua aplicação só precisará executar `pip install -r requirements.txt` e todas as dependências ausentes serão automaticamente baixadas por meio do `pip`. No Windows é possível performar a mesma ação com o `conda` através do comando `conda install --file requirements.txt`.

Perceba que um arquivo `requirements.txt` de um projeto que depende das bibliotecas `numpy`, `torch` e `cv2` é:

```
numpy
torch
opencv-python
```

Existem formas mais avançadas de utilizar o `requirements.txt`, elas podem ser encontradas (em português) aqui (https://jtemporal.com/requirements-txt/).

5.2. Arquivos inexistentes

Ainda na situação hipotética da subseção anterior, considere que seu amigo criou um módulo (o que é muito comum, uma vez que programas bem modularizados/organizados possuem muitos arquivos) e esqueceu de enviá-lo. Tal situação é ainda mais complicada que a anterior pois não é possível obter tal arquivo por outras vias senão pelo autor. Você pode simplesmente pedir ao seu amigo que o envie o arquivo faltante e o problema está resolvido.

Uma forma elegante e segura de garantir que tais erros ocorrerão menos frequentemente é utilizando uma plataforma de hospedagem de código-fonte, como GitHub (https://github.com) e GitLab (https://gitlab.com). No início pode paracer algo muito sofisticado e complicado, mas com tempo e prática tais ferramentas se tornarão fortes aliadas em sua jornada como programador. Caso tenha interesse em saber mais, esse artigo (https://tableless.com.br/tudo-que-voce-queria-saber-sobre-git-e-github-mas-tinha-vergonha-de-perguntar/) (em português) pode te ajudar.

5.3. Erro ao escrever o nome do módulo

É de extrema importância que você verifique duas vezes ou mais se digitou corretamente o nome do módulo, pois importar o módulo com o nome incorreto pode levar o interpretador a produzir erros. Tal problema é de menor gravidade, mas é necessário estar atento para não atribuir outra causa ao erro. A documentação da biblioteca pode ser sua aliada nesses momentos, uma vez que costumam ensinar como importar o módulo (digite " `<nome da biblioteca> docs` " em seu buscador favorito).

Agradecemos pela atenção e esperamos que tenham aprendido bem o conteúdo, para que possamos sempre melhorar é de extrema importância que colaborem através de seu feedback, ficaremos ainda mais gratos caso respondam nosso formulário sobre a aula.