Desenhando com com Turtle Graphics

APRENDA E ENSINE **MATEMÁTICA** E **PROGRAMAÇÃO** COM A LINGUAGEM **Python**

UFPR

Helio H. Monte Alto

DESENHANDO COM TURTLE GRAPHICS

Aprenda e ensine Matemática e Programação com a linguagem Python

Helio H. Monte-Alto

2024

Elaboração e Organização:

Prof. Dr. Helio Henrique Lopes Costa Monte-Alto

Realização:

UFPR
UNIVERSIDADE FEDERAL DO PARANÁ

UFPR
JANDAIA DO SUL

Sumário

Apresentação **1**

O que é Turtle Graphics? **3**

Qual a relação entre Turtle Graphics, Computação e Matemática? **4**

Por que utilizar o Python Turtle? **5**

Preparação do ambiente/máquina **7**

Instalação local **7**

Editores on-line **8**

Introdução ao ambiente **8**

Inserindo os primeiros comandos da tartaruga **13**

Desenhando um quadrado **17**

Opcional: Discussão sobre códigos de programação **22**

Linhas tracejadas **23**

Polígonos regulares e ângulos **31**

Um gancho para introduzir funções matemáticas **38**

De polígonos para círculos **40**

Plotagem de gráficos de funções no plano cartesiano **50**

Indo além: estrelas, espirais e fractais **55**

Estrelas **55**

Espirais **67**

Fractais **77**

Apêndice I **99**

Dicas diversas de uso do Python Turtle **99**

Comandos "Desfazer" e "Refazer" 99

Cores RGB 99

Apresentação

Este livro propõe uma abordagem de ensino cujo objetivo principal é auxiliar professores que queiram utilizar linguagens de programação do tipo Logo (também conhecidas como *Turtle Graphics*, ou Gráficos de Tartaruga) em sala de aula, de modo a introduzir conceitos matemáticos importantes de **geometria** e **álgebra**, e, ao mesmo tempo, estimular e desenvolver o **pensamento computacional** de estudantes da Educação Básica. No entanto, o estudante individual de nível superior pode também tirar proveito do material apresentado, especialmente o estudante de Ciência da Computação que deseja conhecer novas perspectivas de **aprendizagem de algoritmos e programação**.

A fim de proporcionar ao leitor a possibilidade de explorar a abordagem utilizando tecnologias que sejam mais adequadas a diferentes faixas etárias e níveis de conhecimento e habilidades com computadores, o material apresentado é dividido em múltiplos volumes distintos, com conteúdo teórico similar, porém utilizando diferentes tecnologias. Dois volumes estão sendo lançados inicialmente: o primeiro utiliza uma ferramenta inovadora e lúdica, baseada em uma linguagem de blocos no estilo do Scratch, o **Blockly Turtle**. O segundo utiliza uma das linguagens de programação mais populares nos dias atuais: o **Python**. Na introdução de cada volume, uma breve introdução da tecnologia utilizada é apresentada a fim de direcionar a escolha mais adequada para cada leitor. Todos os volumes estarão disponíveis no Acervo Digital - Biblioteca Temática: REA/PEA UFPR[1], bastando pesquisar pelo nome do livro ("Desenhando com Turtle Graphics") e/ou pelo nome do autor.

O conteúdo do livro inclui, além do tutorial básico mais focado no ensino de geometria e álgebra, alguns exemplos mais avançados, como os que envolvem o desenho de estrelas, espirais e fractais. Esse conteúdo introduz conceitos um pouco mais avançados de Algoritmos e Estruturas de Dados, como recursão, *strings*, listas e pilhas. Dessa forma, o estudante pode se aprofundar para além do conteúdo mais focado na transferência de aprendizado entre Matemática e Computação, que é a proposta principal e basilar deste livro.

[1] https://acervodigital.ufpr.br/

Agradecimentos

Parte do conteúdo deste livro teve como ponto de partida uma apostila de programação para professores de Matemática desenvolvida para o Curso de Capacitação de Professores de Matemática para o Ensino com o Apoio de Programação de Computadores, realizado entre setembro e dezembro de 2016 na Universidade Federal do Paraná - Setor Palotina, de autoria do presente autor com a colaboração dos seguintes coautores: Rafael Garcia Cerci, Cassiele Thais dos Santos, Júlio Cezar da Silva Ferreira, e dos professores Marcos Antonio Schreiner e Eliana Santana Lisbôa. Aquela apostila apresentava um tutorial de forma similar ao apresentado neste livro, mas de forma mais simplificada e utilizando tecnologias diferentes. Fica, portanto, meu agradecimento aos demais autores daquela apostila, que foi o ponto de partida para a elaboração deste livro de forma mais completa, revisada e expandida.

O que é *Turtle Graphics*?

Turtle Graphics (Gráficos de Tartaruga) é um método de programação de desenhos vetoriais que utiliza um cursor relativo (a "tartaruga") sobre um plano cartesiano. Esse método é a base da linguagem de programação Logo, originalmente concebida por Seymour Papert[2] como uma linguagem que controla um "robô tartaruga" com o objetivo educacional de introduzir o pensamento computacional para crianças, sendo o precursor do que atualmente é conhecido como Robótica Educacional. Papert descreve a tartaruga como um "objeto-para-se-pensar-com" (do inglês, *object-to-think-with*), no sentido de ser um objeto que pode ser programado de forma tão intuitiva que a pessoa consegue se colocar no lugar dele, como se ela mesma estivesse desenhando com uma grande caneta sobre uma superfície. Assim, o "robô tartaruga" projetado por Papert utiliza uma pequena caneta retrátil, acoplada ao robô, que permite desenhar sobre uma superfície enquanto o robô se move. A linguagem Logo, por sua vez, contém comandos específicos para movimentar o robô, que são usados para construir algoritmos (sequências de instruções) capazes de desenhar diversas formas geométricas e desenhos. Essas instruções são então transmitidas para o robô, que as executa, realizando os movimentos e, consequentemente, desenhando com a caneta sobre a superfície — o próprio chão ou uma mesa forrada com algum tipo de papel, dependendo do tamanho do robô.

Após os trabalhos originais de Papert, a ideia se disseminou, e diversos softwares, que emulam o comportamento da tartaruga na tela do computador de forma totalmente virtual, foram produzidos. Tais ferramentas de software não se utilizam de um dispositivo robótico para executar as instruções de desenho em uma superfície física, porém, permitem, de forma prática, realizar o mesmo tipo de pensamento computacional e criativo. A grande vantagem é a possibilidade de se utilizar o *Turtle Graphics* sem a necessidade de qualquer dispositivo adicional, bastando ter à disposição um computador com acesso à Internet ou com o software específico instalado. Exemplos notáveis de tais ferramentas são: o KTurtle, lançado sob a Licença Pública Geral GNU e parte da KDE Software Compilation 4 (um conjunto de aplicações para o sistema operacional GNU/Linux); o módulo *turtle* da linguagem de programação Python; o Blockly Turtle, uma linguagem de blocos específica para *Turtle Graphics*; e o Scratch, também uma linguagem de blocos, que, embora não seja específica para atividades baseadas em *Turtle Graphics*, é fortemente inspirada nele e permite a realização de desenhos usando esse método.

[2] PAPERT, Seymour. Mindstorms: Children, Computers, and Powerful Ideas (2nd ed.). New York: Basic Books. ISBN 0-465-04674-6. OCLC 794964988. 1993

Qual a relação entre *Turtle Graphics*, Computação e Matemática?

O *Turtle Graphics*, além de estimular o pensamento computacional e criativo, pode ser utilizado, seguindo uma sequência didática adequada, para o ensino de conceitos de Matemática, especialmente aqueles relacionados à geometria e álgebra.

A **geometria** acaba sendo utilizada naturalmente durante o desenvolvimento de algoritmos com o *Turtle Graphics* pela própria natureza deste, que consiste na realização de desenhos vetoriais. No entanto, é possível introduzir conceitos como polígonos, ângulos e plano cartesiano de forma dirigida e fácil de ser aplicada em sala de aula. A primeira parte do conteúdo deste livro é baseada principalmente nessa premissa.

O ensino de **álgebra** provém de outros recursos que estão mais relacionados a linguagens de programação em geral, mas que acabam combinando perfeitamente com o uso da geometria permitido pelo *Turtle Graphics*. A ideia é principalmente explorar a transferência de aprendizagem sobre **funções** em linguagens de programação para **funções matemáticas**. Para exemplificar brevemente como isso funciona: o(a) estudante aprende primeiro como construir uma função/procedimento que desenha um triângulo equilátero na tela baseado no tamanho do lado do triângulo desejado; e então, mostramos que a mesma lógica é utilizada na definição de funções matemáticas cujo domínio e imagem se restringem somente a números — como, por exemplo, $f(x) = 2x$. Essa abordagem já foi estudada por diversos pesquisadores com excelentes resultados, como é o caso da equipe do *Bootstrap*[3], um currículo de ensino de matemática e computação desenvolvido na Universidade de Brown, nos Estados Unidos.

Além disso, o ensino de álgebra é também aumentado por meio da explicação sobre o conceito de **variáveis**, assim como a necessidade de se construir **fórmulas**, como é o caso do cálculo da circunferência de um círculo tendo como parâmetro o raio do círculo que se deseja desenhar. Pode-se explorar também a plotagem de **gráficos** de funções, o que permite um aprofundamento ainda maior no conceito de funções matemáticas e **plano cartesiano**.

Portanto, o principal objetivo deste livro é apresentar um currículo, ou sequência didática, que permita ao menos dar um norte ao educador que pretenda utilizar *Turtle Graphics* para ensinar Computação ao mesmo tempo em que ensina Matemática (ou vice-versa), contribuindo assim para o desenvolvimento do pensamento computacional e um maior engajamento, interesse e compreensão do conteúdo matemático tangido pela abordagem.

[3] SCHANZER, Emmanuel et al. Transferring skills at solving word problems from computing to algebra through Bootstrap. In: Proceedings of the 46th ACM Technical symposium on computer science education. 2015. p. 616-621.

Por que utilizar o Python Turtle?

A ferramenta utilizada neste livro é o módulo *turtle* disponível na linguagem Python, o qual chamaremos simplesmente de "Python Turtle" no decorrer do material. O Python é uma linguagem de programação de alto nível, de propósito geral, interpretada e multiparadigma, lançada inicialmente em 1991 por Guido van Rossum. No decorrer dos anos, o Python tem ganhado cada vez mais adeptos, e atualmente é uma das linguagens de programação mais populares, estando, em 2022, em 2º lugar no ranking das linguagens mais utilizadas em projetos no GitHub (o maior repositório de código-fonte da web), atrás apenas do Javascript. A linguagem é de fácil aprendizagem e muito versátil, sendo utilizada por muitos desenvolvedores nos mais diversos tipos de projetos de software. A linguagem também tem se popularizado nos últimos anos na área de Ciência de Dados e Aprendizagem de Máquina (*Machine Learning*), graças à boa variedade de bibliotecas de qualidade para esse fim.

A instalação padrão do Python possui, entre seus módulos nativos (*built-in modules*), o *turtle*, que implementa a ideia da linguagem de programação Logo, permitindo o uso do *Turtle Graphics*. Portanto, muitos currículos de Introdução à Ciência da Computação e Algoritmos e Estruturas de Dados utilizam a linguagem juntamente com exemplos em Python Turtle com o intuito de facilitar o aprendizado de conceitos de Algoritmos. Um exemplo notável é o livro interativo "Como Pensar Como um Cientista da Computação" (do original em inglês, *How to Think Like a Computer Scientist: Interactive Version*), disponível no site Panda do Instituto de Matemática e Estatística da Universidade de São Paulo (IME-USP)[4], que utiliza o Python Turtle em alguns exemplos no decorrer do livro.

Portanto, este volume tem o intuito de apresentar um tutorial, que pode ser aplicado em sala de aula ou usado para estudo individual, que apresenta conceitos de Matemática e Computação utilizando o Python Turtle.

Algumas das vantagens de se utilizar o Python Turtle são as seguintes:

1. A linguagem Python é de fácil aprendizagem em comparação com outras linguagens de programação. Também é de fácil instalação e inclui um ambiente de desenvolvimento (editor) de fácil utilização.
2. A linguagem Python é uma linguagem de propósito geral, isto é, pode ser utilizada no desenvolvimento dos mais diversos tipos de software, e não apenas como uma forma de aprender com o *Turtle Graphics*. Essa é uma vantagem em relação a outros ambientes de *Turtle Graphics*, que utilizam uma linguagem específica para esse fim e que não pode ser utilizada em outros projetos, como é o caso do Blockly Turtle, KTurtle e Turtle Academy.
3. Pelo fato de ser uma linguagem de propósito geral, também é apropriada para o ensino de programação para adolescentes e jovens de idade mais avançada, ou mesmo para crianças que já tiveram contato com a programação por meio de linguagens mais simples, como o Scratch ou o Blockly Turtle. Assim, eles terão a oportunidade não só

[4] https://panda.ime.usp.br/panda/static/pensepy/index.html

de fortalecer os conhecimentos matemáticos tratados neste material, como também aprenderão a usar uma linguagem que pode ser usada em projetos diversos e que possui demanda no mercado de trabalho.

Quando não utilizar o Python Turtle: Em turmas de alunos com menos idade e/ou que nunca tiveram nenhum contato com programação. Pelo fato de ser uma linguagem de programação textual, é comum que os(as) alunos(as) cometam erros de digitação e outros erros na escrita do código que podem prejudicar o andamento das aulas. Portanto, recomenda-se o uso para um público que tenha um pouco mais de experiência no uso do computador. Para os menos experientes, sugere-se o uso principalmente do Blockly Turtle, disponível em outro volume deste material, também disponibilizado no Acervo Digital - Biblioteca Temática: REA/PEA UFPR[5].

[5] https://acervodigital.ufpr.br/

Preparação do ambiente/máquina

Há duas formas de se utilizar o Python: instalando-o no computador, ou por meio de editores on-line.

A primeira forma (*instalação local*) é mais indicada para quando não se tem uma conexão estável com a Internet, quando pretende-se aprender/ensinar a preparar o próprio ambiente de desenvolvimento, e/ou quando deseja se evitar problemas relacionados à necessidade de fazer cadastro e *login* ou de copiar o código para um lugar seguro a cada mudança. Portanto, esta é a forma mais indicada em geral, especialmente se você tem acesso prévio às máquinas do laboratório no qual as aulas serão ministradas, de modo que você mesmo(a) possa prepará-las de antemão.

A segunda forma (*editores on-line*) é indicada para quando se tem conexão estável com a Internet e deseja-se evitar problemas relacionados à instalação de programas no computador. No entanto, é necessário, em alguns casos, fazer cadastro e *login* no editor on-line escolhido, ou instruir os alunos a salvaram os projetos da forma correta, geralmente baixando ou copiando o código e então fazendo o upload por e-mail ou algum serviço de armazenamento de dados em nuvem.

Ambas as formas são válidas para o estudante individual, e serão apresentadas a seguir.

Instalação local

No Windows:

1. Acesse o site https://www.python.org/
2. Procure e acesse a opção de fazer download. Um arquivo de instalação será baixado para seu computador, geralmente para a pasta *Downloads*.
3. Execute o arquivo de instalação clicando-se duas vezes sobre ele. Siga as instruções para instalação.
4. Se nenhum erro ocorreu, vá ao menu Iniciar (símbolo do Windows) e procure por IDLE. Se abrir uma janela em branco com algumas coisas escritas no topo, significa que a instalação foi feita corretamente.

No Linux Ubuntu/Mint/Debian:

- Abra o terminal (Ctrl+Alt+T) e digite: *sudo apt install python3 -y*
  - A senha do usuário será solicitada; insira-a e tecle *Enter*.
- Para instalar o IDLE (editor), há duas opções:
  - Digite no terminal *sudo apt install idle-python3.x*, substituindo o *x* pela versão do Python que foi instalada, ou
  - Vá ao Ubuntu Software ou Gerenciador de Aplicativos e pesquise por “Python IDLE”, e então peça para instalar.

Os passos acima servem para instalar o interpretador do Python e seu editor padrão, o IDLE. Se desejar, após os passos acima, é possível (e recomendado) também instalar um editor mais

completo, como o Visual Studio Code. Pesquise na Internet como fazer sua instalação, que também é muito simples.

Editores on-line

Há vários editores on-line para Python disponíveis na web. Se você quiser um editor mais simples, que não requer cadastro e *login* e no qual funcione o Python Turtle, acesse um dos seguintes sites:

- https://trinket.io/turtle
  - Esta é a **opção mais recomendada**, pois permite fazer o download do código, mas também permite enviar o código para o e-mail e criar/compartilhar (menu *Share*) um link que pode ser aberto mais tarde, bastando copiá-lo em algum lugar seguro. Todas essas opções são encontradas clicando-se no ícone  , no canto superior esquerdo da área de código.
- https://www.pythonsandbox.com/turtle
  - Esta é também uma boa opção, mas permite apenas fazer download do código.

Caso você queira um ambiente mais completo, que permita salvar seu código diretamente na nuvem, como se fosse uma pasta virtual, sugere-se o https://replit.com/. Acesse o site, faça cadastro (*Sign Up*) ou *login*, caso já possua conta. Após logado(a), faça o seguinte:

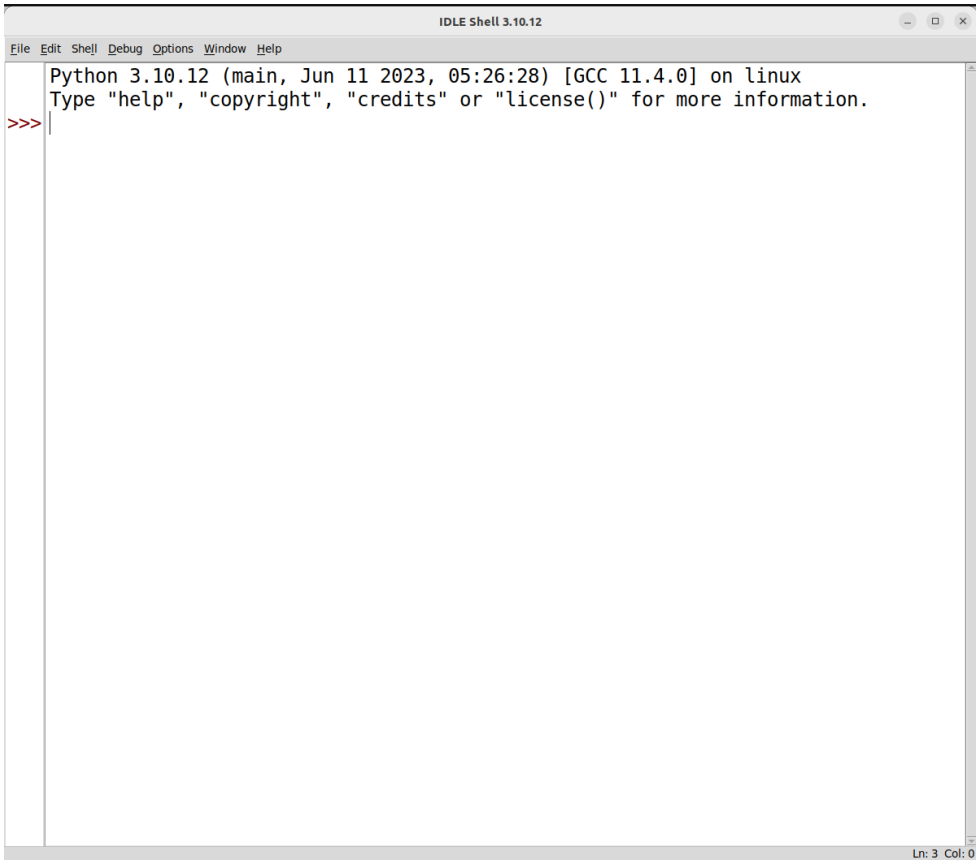
1. Clique no botão *Create Repl*
2. Na caixa de opções *Templates*, procure por "*Python (with Turtle)*"
3. Em *Title*, insira um nome para o projeto
4. Clique em *Create Repl*

Uma vez feito isso, um ambiente de desenvolvimento com um editor será aberto. Do lado esquerdo você encontrará uma área de navegação de arquivos na qual você pode criar e selecionar diferentes arquivos de código-fonte. No centro, você verá o editor, onde escrevemos nosso código, e do lado direito você verá o *shell*, a área em que o resultado da execução é exibido.

Introdução ao ambiente

No decorrer do material, será utilizado o IDLE, que é o editor padrão da linguagem Python. No entanto, tudo o que faremos nele pode ser feito — muitas vezes de forma até mais facilitada — nas demais opções apresentadas na seção anterior. Não se preocupe, pois, no decorrer do texto, serão dados indicativos de como fazer a mesma coisa que fizermos no IDLE em outros editores.

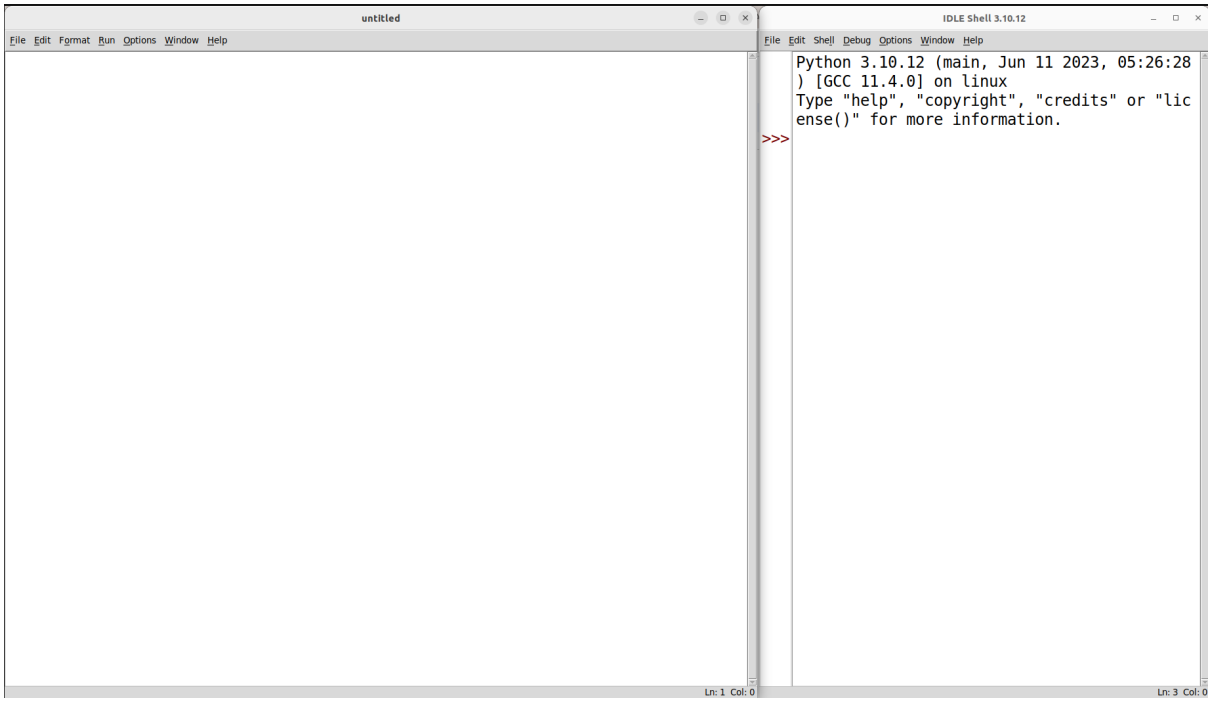
Primeiro, abra o IDLE (por meio do menu Iniciar, no caso do Windows, e de modo similar em outros sistemas operacionais). Você verá uma tela similar à seguinte:



Em seguida, no menu superior, vá em *File → New File*. Uma nova janela em branco se abrirá.

Essa é a **área de edição** de código. É aqui que vamos escrever nosso código para controlar nossa tartaruga.

A outra janela, que foi aberta primeiro, permanecerá aberta. Essa é a que chamamos de ***shell***, mas pense nela como a área em que serão exibidos os resultados da execução do nosso código (embora a animação da tartaruga seja aberta ainda em outra janela). Recomenda-se deixar as duas janelas abertas uma ao lado da outra durante a execução. Basta arrastá-las para ficarem lado a lado, como mostrado abaixo.

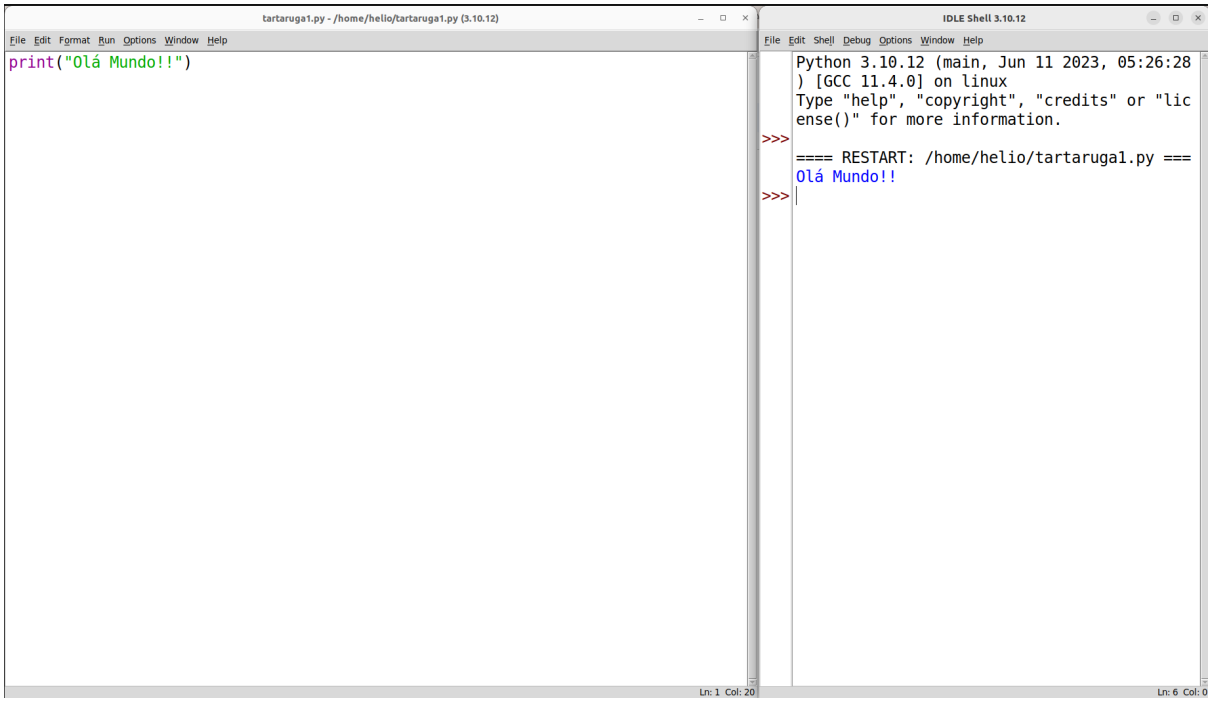


> **Nota:** se você estiver utilizando algum dos outros ambientes sugeridos, você irá notar que também possui os mesmos elementos na tela: uma área de edição, geralmente mais à esquerda, e uma área de resultados, geralmente à direita ou embaixo.

Antes de começarmos a escrever nossos primeiros programas, é necessário salvar o nosso código em um arquivo. Portanto, na janela de edição (da esquerda), vá em *File* → *Save* (ou tecle Ctrl+S). Uma janelinha se abrirá para que você escolha a pasta em que deseja salvar seu código e o nome do arquivo. Escreva um nome terminado com *.py*, por exemplo: *tartaruga1.py*. Todos os arquivos de código-fonte em Python devem ser terminados com a extensão *.py*. Clique em *Save* para confirmar.

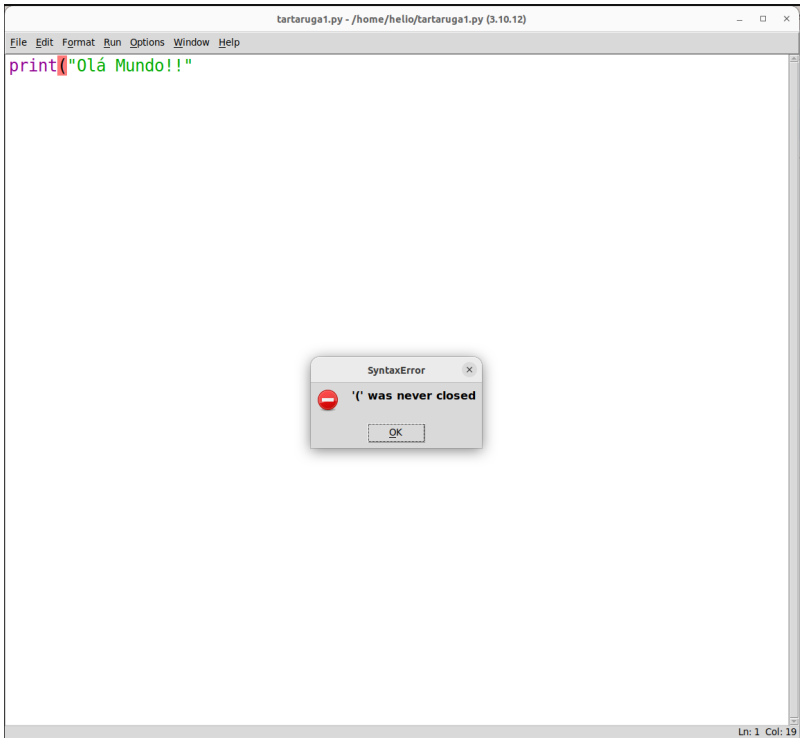
Agora já podemos começar a escrever nosso primeiro programa em Python. Antes de brincarmos com a tartaruga, vamos testar alguns comandos da linguagem. Vamos fazer um programa que escreva na tela a frase: "Olá Mundo!!". Para isso, na área de edição (da esquerda), escreva, na primeira linha: `print("Olá Mundo!!")`. O comando `print` serve para escrever/imprimir um texto na tela.

Não se esqueça de abrir e fechar as aspas em torno da frase, e de fechar também os parênteses corretamente. Em seguida, vá no menu *Run → Run Module*. Aparecerá uma janelinha perguntando se você deseja salvar (*Save)* o arquivo. Simplesmente aperte em *Ok* ou tecle *Enter*. Pronto! Escrevemos e executamos nosso primeiro programa em Python! Note que, na outra janela (da direita), o programa escreverá a frase "Olá Mundo!!". Veja abaixo:



Caso você não esteja vendo o resultado, verifique se a janela do *shell* (a janela dos resultados) está visível na sua tela.

Se, em vez de imprimir a frase, algum erro tiver ocorrido, verifique se você escreveu corretamente o comando. Por exemplo, abaixo eu esqueci propositalmente de fechar o parêntese no final, e veja que o programa reportou um erro de sintaxe, indicando que eu esqueci de fechar o parêntese.



É possível também executar nosso código simplesmente pressionando a tecla F5. Tente fazer isso!

> **Nota:** se você estiver usando algum dos outros ambientes sugeridos, procure por um botão em formato de *"play"*, isto é, um triângulo deitado. Esse é o botão no qual você deve clicar para executar.

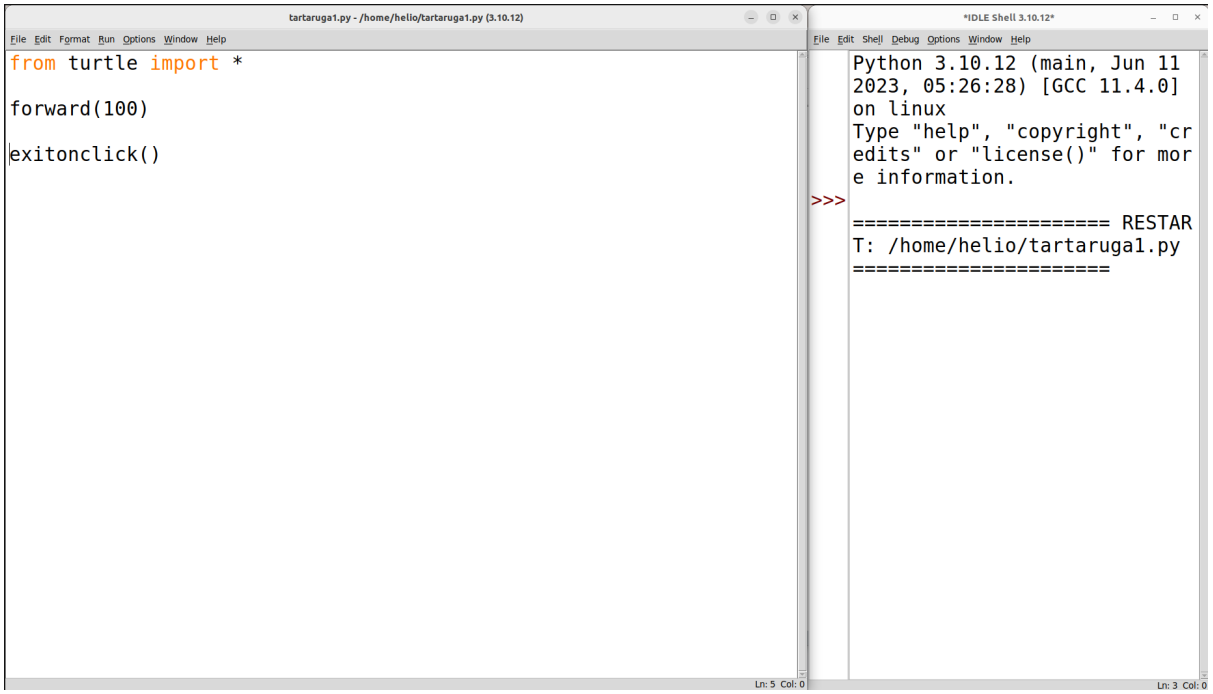
É interessante notar que o Python, por ser uma linguagem de programação textual, requer que você escreva corretamente o código, conforme as suas regras próprias de sintaxe. Por isso, ele indicará erros e não executará corretamente o programa se algo estiver fora dessas regras, como é o caso de esquecer de fechar aspas ou parênteses. Mas fique tranquilo(a), pois no decorrer deste livro serão apresentadas, aos poucos, outras dessas regras de sintaxe.

> **Dica:** Pode ser útil também alterar o tamanho da fonte do editor, deixando-a maior caso deseje. Para isso, vá ao menu *Options → Configure IDLE*, e, na aba *Font*, procure por uma caixa de seleção chamada *Size*. Altere o tamanho da fonte para um tamanho maior utilizando essa caixa, e então clique em *Ok*. Teste vários tamanhos de fonte e veja qual fica melhor para você.

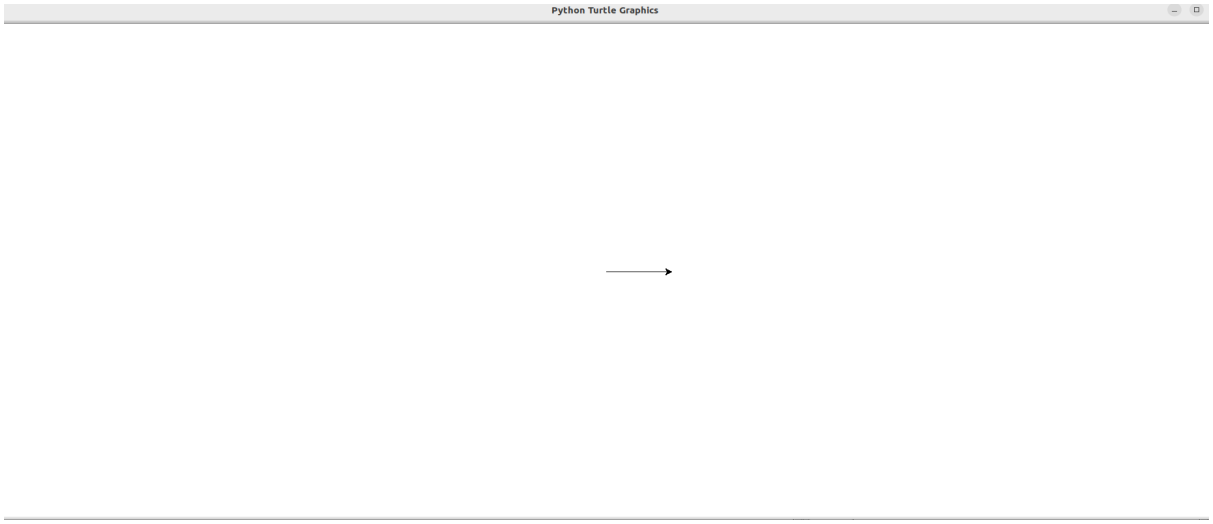
Inserindo os primeiros comandos da tartaruga

Apague a linha com o `print` que escrevemos anteriormente e agora escreva os seguintes comandos:

```
from turtle import *

forward(100)

exitonclick()
```



Agora, aperte F5 ou *Run → Run Module* para executar o código. Uma nova janela se abrirá mostrando o resultado da execução da tartaruga (verifique se não abriu em outra tela/monitor, caso esteja utilizando várias telas):



Clique em qualquer lugar dentro dessa janela para fechá-la.

O primeiro comando, `from turtle import *`, é um comando que sempre precisaremos colocar no início do nosso código, pois serve para importar o módulo *turtle* para dentro do nosso código.

O segundo comando, `forward(100)`, já é um exemplo de instrução que podemos dar à tartaruga. No caso, estamos pedindo para a tartaruga se mover para frente por uma distância de

100 pixels[6]. A palavra em inglês *forward* significa “para frente”. Portanto, explique para os alunos que estamos pedindo que a tartaruga, representada pelo ponteiro (➤), ande para a frente. Note que a tartaruga começa apontando para a direita.

O último comando, `exitonclick()`, será um comando que sempre colocaremos na **última** linha de código, após todas as instruções. Ele serve para facilitarmos o fechamento da janela de resultado (onde aparece a tartaruga e seu desenho).

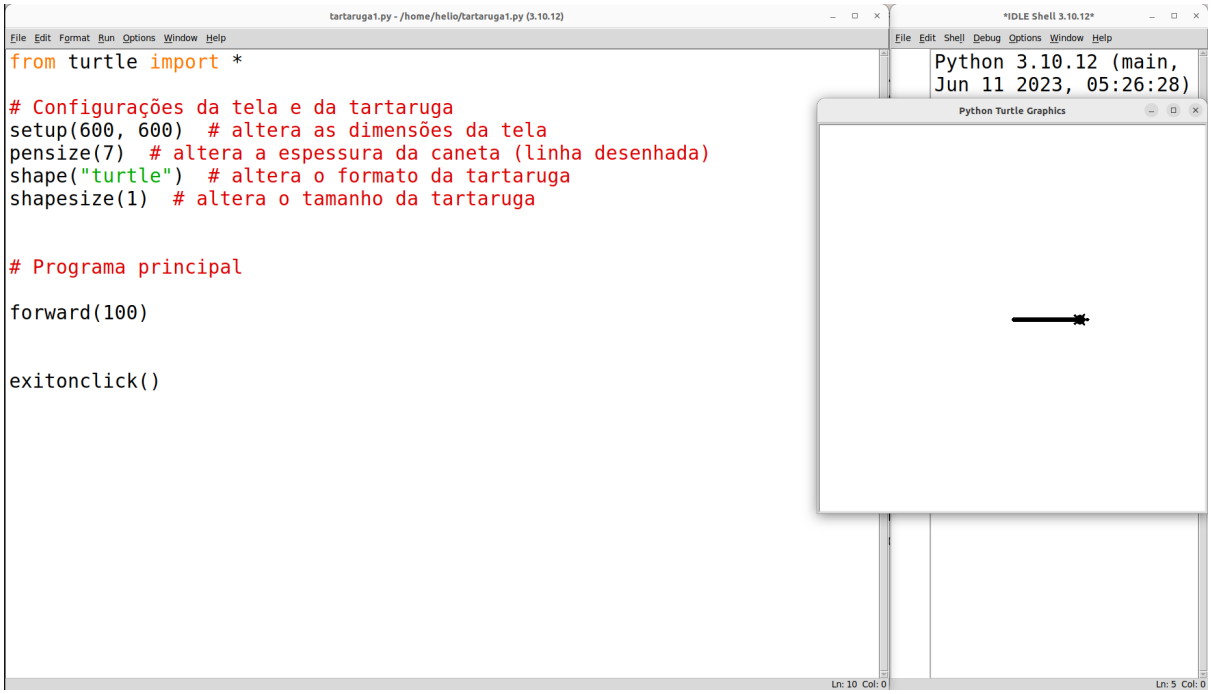
Antes de vermos mais comandos, é possível fazer algumas alterações na visualização da tartaruga, principalmente no tamanho da tela, no formato do ponteiro da tartaruga e na espessura da linha desenhada. Para isso, logo após o comando de `import` e antes do comando de `forward`, insira as seguintes linhas de código:

```
# Configurações da tela e da tartaruga
setup(600, 600)  # altera as dimensões da tela
pensize(7)  # altera a espessura da caneta (linha desenhada)
shape("turtle")  # altera o formato da tartaruga
shapesize(1)  # altera o tamanho da tartaruga
```

Note que colocamos todas essas linhas juntas, pois servem apenas para fazer algumas configurações. As partes do código iniciadas por `#` (*hashtag*, ou jogo da velha) são chamadas de **comentários**, e não são executadas pelo Python como código de programação. Servem justamente para acrescentar comentários que explicam ou delimitam diferentes partes do nosso código.

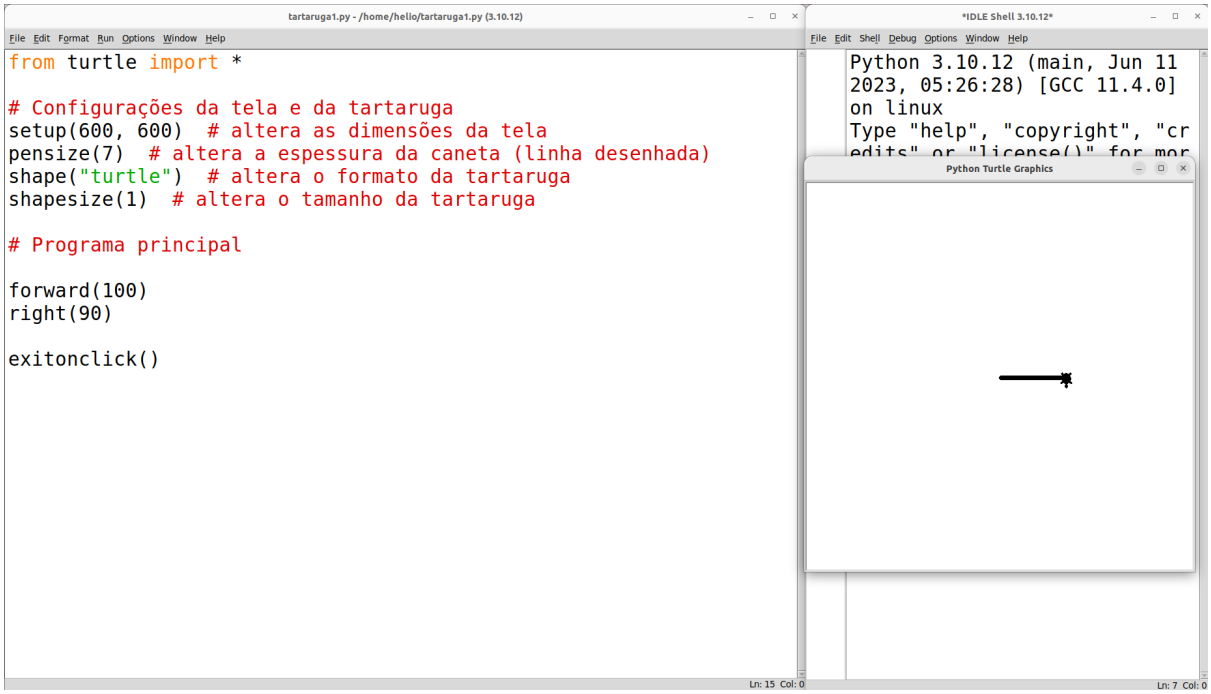
Execute (F5) e você verá o seguinte resultado:

[6] Uma tela de computador é formada por diversos pontinhos de luz chamados de pixels. Portanto, ao tratarmos das distâncias percorridas pela tartaruga, dizemos que ela se moveu na tela por uma certa quantidade de pixels. Para entender mais sobre pixels, visite https://pt.wikipedia.org/wiki/Pixel.



Peça para os alunos alterarem o número dentro do comando `forward` para que ande por uma quantidade diferente de espaço. Por exemplo, troque para 200 e peça que executem novamente o programa. Não se esqueça de abrir e fechar parênteses em torno do 200, ficando assim: `forward(200)`. Não se esqueça de executar novamente o programa.

Mostre também que é possível fazer a tartaruga girar. Peça para os alunos incluírem, logo após o `forward`, o comando: `right(90)`. Execute (tecla F5).



Pergunte: *"O que aconteceu?"*. Perceba que a tartaruga virou à direita por 90º, e explique que *right*, em inglês, significa "direita". Logo, esse comando permite fazer a tartaruga girar, em torno de seu próprio eixo, pela quantidade de ângulos colocada entre parênteses.

Peça para os alunos alterarem o ângulo para vários ângulos diferentes, tais como: 180, 270, 360 e 45. Lembre-se de executar novamente após cada alteração para ver o que acontece.

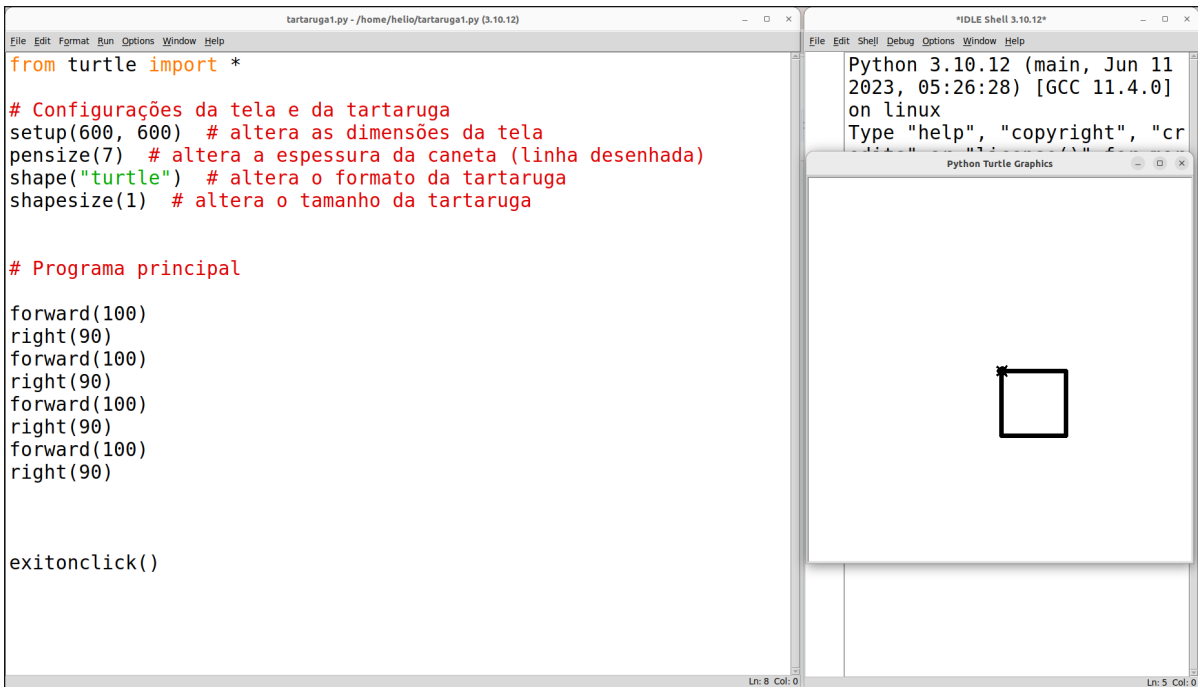
Agora pergunte a eles: *"E se eu quisesse que a tartaruga girasse para a esquerda?"*. Peça que, no lugar de `right(90)`, coloquem `left(90)`, e explique que *left*, em inglês, significa "esquerda". Teste novamente para vários ângulos distintos.

> **Opcional:** pode ser interessante notar aqui que, mesmo que só tivéssemos o comando `right`, conseguiríamos fazer a tartaruga girar em qualquer ângulo. Por exemplo, girar 90º à esquerda tem o mesmo resultado de girar 270º à direita, pois *360-90 = 270*. Do mesmo modo, tanto faz girar 180º à direita ou 180º à esquerda.

Desenhando um quadrado

Agora proponha um desafio aos alunos. Peça que eles tentem criar um quadrado usando os dois comandos aprendidos (`forward` e `right`). Mostre na lousa como se faz um quadrado, fazendo uma analogia entre seu giz/caneta e a tartaruga. Dê alguns minutos para eles tentarem desenhar um quadrado.

Elogie e, talvez, recompense os alunos que conseguiram desenhar o quadrado. Parabenize também os que tentaram mas não conseguiram, tentando estimulá-los a não desanimar e continuar se esforçando. Mostre uma possível solução/algoritmo (sem utilizar *loop*/laço de repetição):



Agora, instigue os alunos a analisarem a solução quanto ao fato de ter que ficar repetindo os mesmos comandos várias vezes para desenhar o quadrado. *"Vocês perceberam que tivemos que repetir várias vezes o 'forward' e o 'right'? Quantas vezes repetimos esses comandos?"*. Junto com os alunos, conte quantas vezes foram repetidas as sequências de comandos `forward` e `right`. Você deverá chegar, junto a eles, à conclusão de que cada sequência foi repetida 4 vezes.

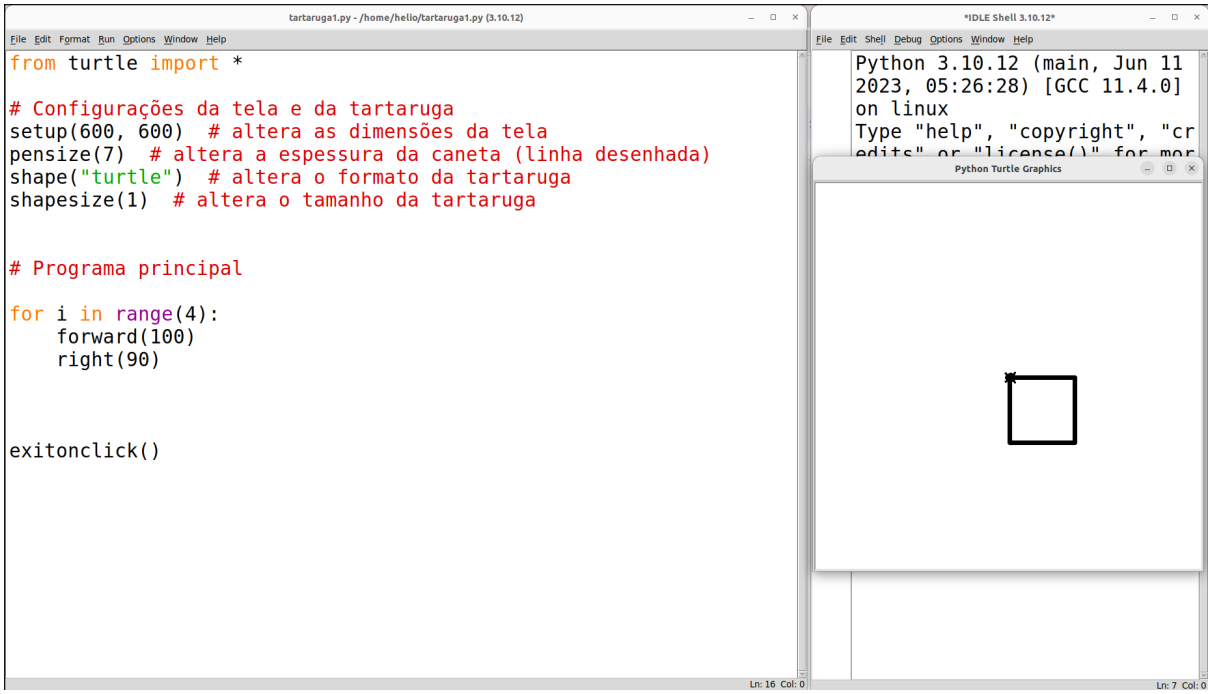
Hora de introduzir nosso primeiro comando de laço (*loop*) de repetição: o **`for`.** Peça que os alunos apaguem as 8 linhas com os comandos repetidos, e escrevam no lugar:

```
for i in range(4):
      forward(100)
      right(90)
```

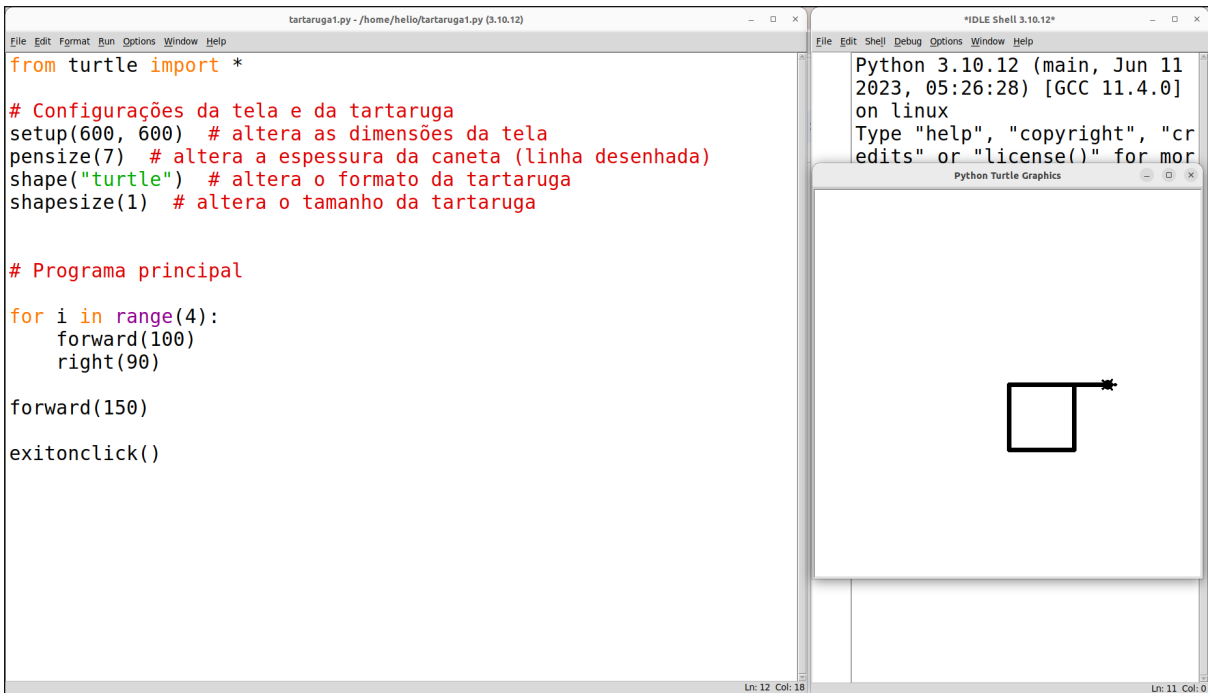
O **`for`** repete uma quantidade de vezes um conjunto de instruções. Note que é necessário dar um espaçamento no início das linhas que vêm logo depois do comando **`for`**. Isso serve para indicar que essas linhas de instrução estão **dentro** do **`for`**. Imagine que o **`for`** é uma caixa, ou bloco, e que dentro dela colocamos comandos que devem ser repetidos por uma determinada quantidade de vezes. Assim, todos os comandos que devem ser repetidos devem estar com esse espaçamento (ou **indentação**). Essa indentação pode ser feita usando-se a tecla *tab* do teclado, ou inserindo-se 4 espaços em branco. Tome cuidado para que o mesmo tamanho de espaço seja usado em cada linha. Caso contrário, o Python retornará um erro de sintaxe.

Note que colocamos o valor 4 na frente do `range`, entre parênteses, e por isso os dois comandos dentro do **`for`** (o `forward` e o `right`) são repetidos 4 vezes.

Enfim, execute o programa, e teremos o quadrado desenhado, como antes, mas com um código muito mais enxuto e inteligente.



Para entender melhor a diferença de comandos que estão dentro e fora do `for`, peça para os alunos escreverem, logo após o último comando do `for`, porém sem indentação (espaçamento), um comando `forward(150)`. Peça para executarem. Veja abaixo:

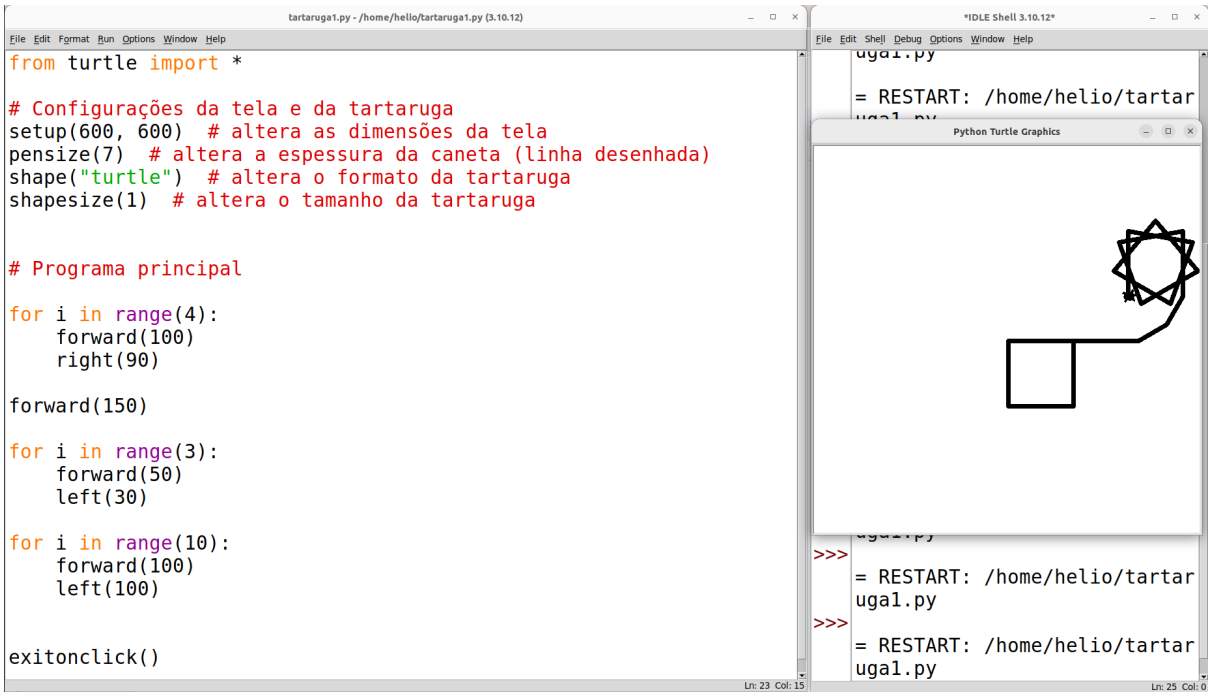


Note que esse último comando só foi executado uma única vez, e não quatro. Isso se deve ao fato de que ele está **fora** do `for`, sendo executado logo após o término de suas 4 repetições. Veja abaixo uma forma de visualizar o que está dentro e fora do `for`. Novamente, pense e reforce aos alunos o fato de o `for` ser como um novo bloco, ou caixa, e que os itens que ele repete estão dentro de seu escopo, isto é, dentro da caixa/bloco, e por isso estão indentados.

```
for i in range(4):
    forward(100)
    right(90)

forward(150)

exitonclick()
```

Pode ser conveniente também escrever o código no quadro e desenhar um retângulo em torno do `for` e dos comandos que estão dentro dele, de modo a deixar ainda mais clara a ideia, como na figura acima.

Peça em seguida para eles fazerem alguns novos blocos de repetição após o desenho do quadrado. Abaixo são dados alguns exemplos:



Aqui pode ser interessante introduzir também os comandos **levantar e abaixar a caneta**. Explique que a tartaruga tem uma caneta na mão, e por isso ela desenha quando anda. Normalmente, ela anda com a caneta encostada no "chão" (aqui simulado pela tela), o que faz ela desenhar. No entanto, ela também pode levantar a caneta para que possa andar sem desenhar nada no "chão". Peça para eles inserirem as seguintes linhas depois dos `for:`

```
penup()
right(180)
forward(100)
```

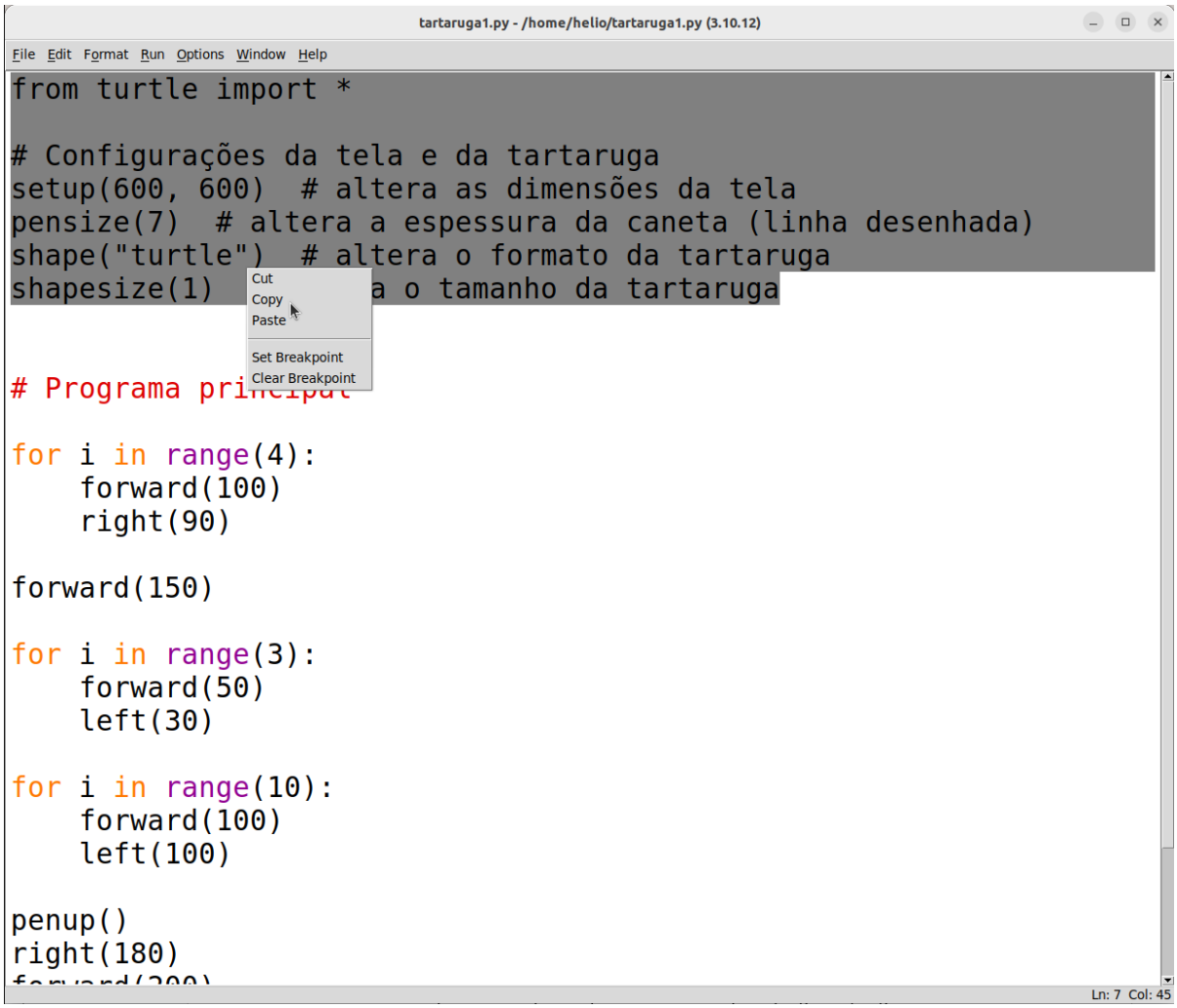
Pergunte o que aconteceu. Note que a tartaruga se moveu sem desenhar. Agora peça para eles colocarem, logo em seguida:

```
pendown()
forward(100)
```

Pergunte o que aconteceu. Deverá ficar evidente para eles que o comando `penup()` faz a tartaruga levantar a caneta (deixando de desenhar), e `pendown()` faz ela abaixar a caneta (voltando a desenhar). Explique que *pen* em inglês significa "caneta", *down* significa "abaixar", e *up* significa "levantar". Logo, *pen up* seria "levantar caneta" e *pen down* seria "abaixar caneta".

Peça para os alunos salvarem seus programas, pois em seguida iremos fazer um novo programa do zero. Basta ir em *File → Save* ou teclar Ctrl+S.

Em seguida, peça para eles copiarem somente até a linha que tem o comando `shapesize`. Como já explicado, essas são linhas de configuração, portanto teremos que incluí-las em cada novo arquivo de código. Para copiar, basta selecionar (clicando e arrastando o mouse) o código a ser copiado, e então teclar Ctrl+C no teclado, ou clicar com o botão direito do mouse sobre o código selecionado e escolher a opção *Copy*.



Agora, vá ao menu *File → New File*. Isso fará abrir um novo editor em branco. Salve-o, como já fizemos antes (menu *File → Save*), com o nome *linhatracejada.py*. Por fim, cole o pedaço de código que copiamos do outro arquivo. Para isso, basta teclar Ctrl+V no teclado ou clicar com o botão direito do mouse na tela em branco e escolher a opção *Paste*. Se desejar, pode-se fechar o arquivo anterior. Podemos abri-lo novamente mais tarde se desejarmos.

Opcional: Discussão sobre códigos de programação

Esta é uma boa oportunidade para explicar aos alunos que a programação é feita de códigos como os que acabamos de escrever. Explique que todos os aplicativos de celular, programas de computador e jogos são programas, e foram escritos utilizando códigos de programação. Uma ideia para exemplificar isso é mostrar o código-fonte de uma página qualquer da web. Abra, por exemplo, a página inicial do site da sua instituição de ensino em uma nova aba do navegador, clique com o botão direito do mouse em qualquer parte em branco da página, e procure a opção "Mostrar código-fonte" (a forma como isso está escrito varia de navegador para navegador). Será aberta uma janela/aba contendo o código em HTML[7] da página atual. Se quiser mostrar um pouquinho mais, feche a janela/aba do código-fonte, volte à página, clique com o botão direito e escolha a opção "Inspecionar". Uma parte diferente será aberta dentro do navegador. Dentro dessa parte, provavelmente haverá algumas abas (*"Console"*, *"Elements"*, *"Source"*, etc.). Clique na aba *"Source"* (ou "Fonte"). Ali você verá uma estrutura de diretórios (pastas) contendo os diversos arquivos de código-fonte que estão sendo executados para a página funcionar como ela funciona. Mostrando isso, você explica que cada página da Internet, assim como cada aplicativo de celular e os jogos que os alunos jogam no computador ou em *consoles* de videogame, são todos feitos de códigos de programação que outras pessoas escreveram.

Explique que, por mais que esses códigos pareçam "assustadores" à primeira vista, é importante entender que um programa complexo é geralmente desenvolvido por uma equipe de várias pessoas e/ou com um longo tempo de desenvolvimento, que vão desde meses até anos. Exemplifique isso com alguns dados sobre o tempo de desenvolvimento de alguns jogos famosos. Abaixo são apresentados alguns exemplos.

Jogo	**Tempo de desenvolvimento**	**Quantidade de pessoas na equipe**	**Orçamento**
Call of Duty: Modern Warfare 2	2 anos	500+	Mais de 50 milhões de dólares

[7] HTML (*Hypertext Markup Language*) é a linguagem de marcação usada para *design* e construção das páginas web (popularmente chamadas, no Brasil, de "páginas da Internet", ou *"sites"*).

The Legend of Zelda: Breath of the Wild	5 anos	300	Aprox. 120 milhões de dólares
Grand Theft Auto V	5 anos	1000	Aprox. 265 milhões de dólares

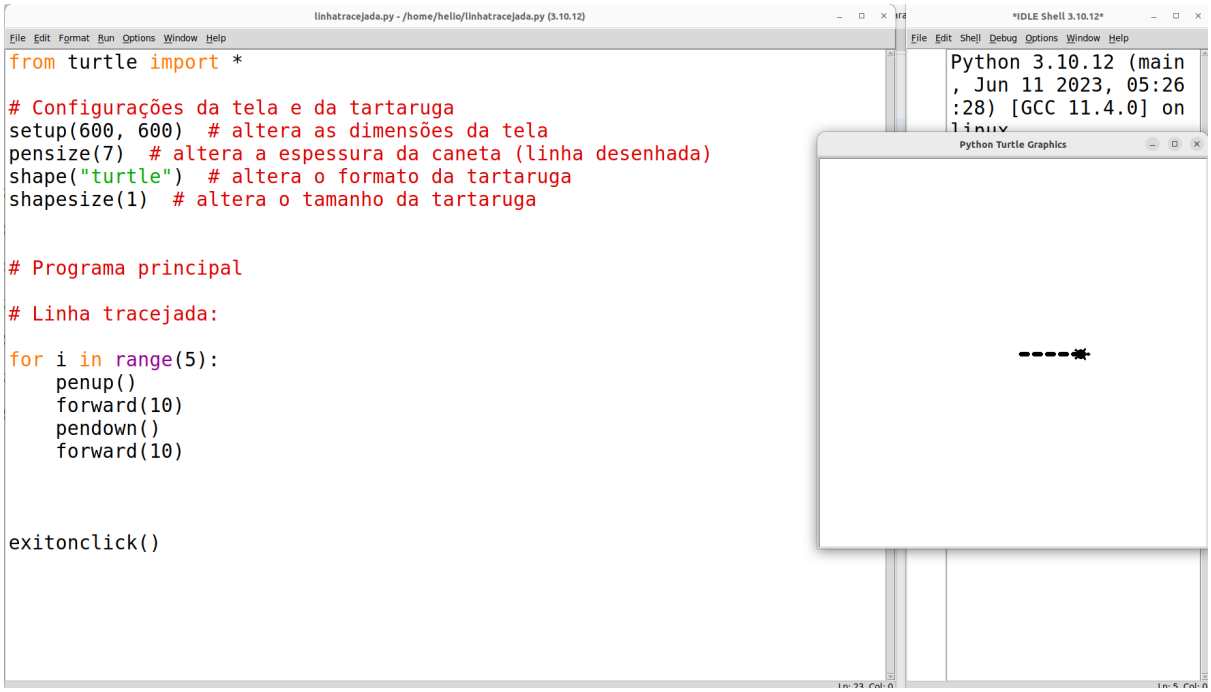
Linhas tracejadas

Agora, proponha um desafio: *"E se eu quiser que a tartaruga faça uma linha tracejada? Quero que ela desenhe uma linha tracejada com cinco traços"*. Mostre na lousa o que você quer fazer. Dê alguns minutos para os alunos tentarem por si próprios. Dê a dica de que eles podem usar o `for` para ficar mais fácil.

Elogie os que conseguiram e os que se esforçaram, e mostre a solução:

```
for i in range(5):
    penup()
    forward(10)
    pendown()
    forward(10)
```

Veja abaixo a figura de como fica:

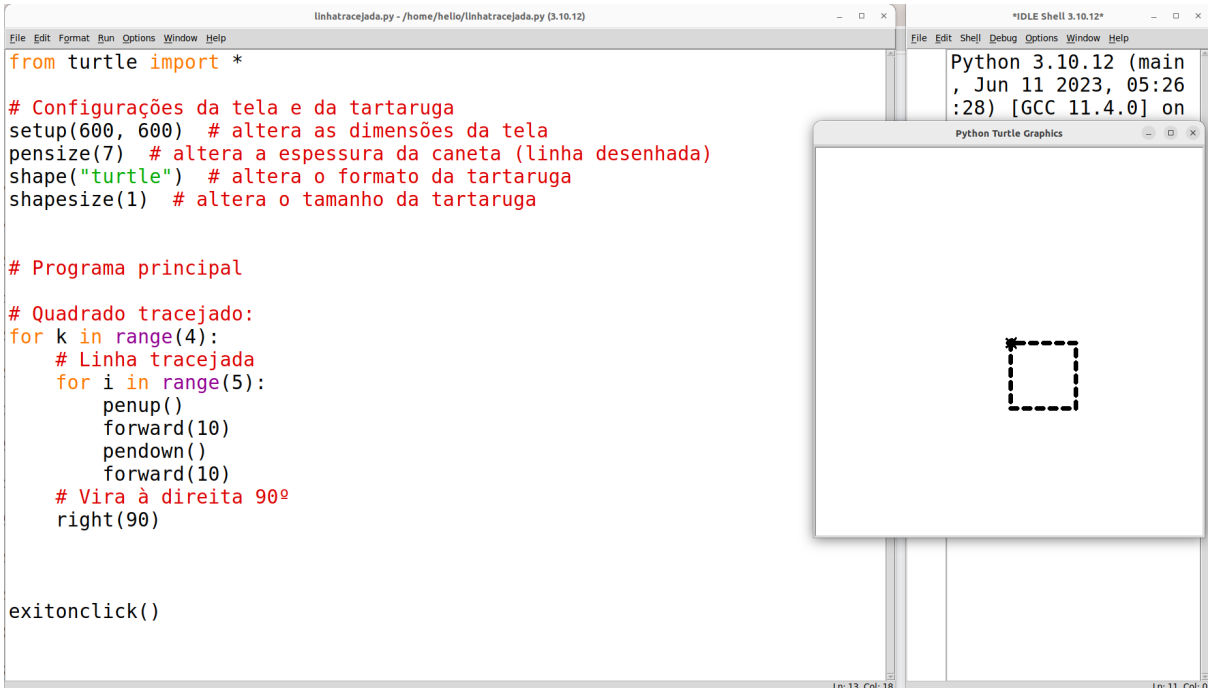


Agora pode ser interessante lançar mais um desafio: peça aos alunos que desenhem um quadrado feito com as linhas tracejadas. Na lousa, mostre o que você quer fazer, deixando clara

a ideia de usar os mesmos comandos para linhas tracejadas, porém repetindo-os 4 vezes e girando 90 graus cada vez. Dê alguns minutos para eles tentarem. Se alguém conseguir, elogie e estimule os demais. Dê a solução:

```
# Quadrado tracejado:
for k in range(4):
    # Linha tracejada
    for i in range(5):
        penup()
        forward(10)
        pendown()
        forward(10)
    # Vira à direita 90°
    right(90)
```

Veja abaixo como fica:



Explique que cada `for` tem seu próprio escopo e que, neste caso, temos dois `for` aninhados (um dentro do outro). Pense como se fossem duas caixas, uma dentro da outra: um `for` dentro do outro. Assim, é importante notar as indentações (espaçamentos). Os 4 comandos que desenham a linha tracejada estão dentro do `for` mais interno, e, portanto, possuem dois espaçamentos (dois *tabs*). Já o `for` mais interno e o comando `right` estão dentro do `for` mais externo, e por isso possuem apenas um espaçamento. Veja as duas figuras abaixo para uma ilustração dos `for` como duas caixas aninhadas. Desenhe na lousa algo parecido para ajudar no entendimento.

```
for k in range(4):
    # Linha tracejada
    for i in range(5):
        penup()
        forward(10)
        pendown()
        forward(10)
    # Vira à direita 90°
    right(90)
```

```
for k in range(4):
    # Linha tracejada
    for i in range(5):
        penup()
        forward(10)
        pendown()
        forward(10)
    # Vira à direita 90°
    right(90)
```

Note também que ambos os `for` devem ter variáveis de laço diferentes. Ainda não usamos diretamente as variáveis de laço, mas todo laço (*loop*) do tipo `for` as possui. Mais adiante veremos mais detalhes sobre variáveis de laço. Por enquanto, pense nelas como simplesmente a letrinha ou palavra que colocamos depois do `for` e antes do `in`. Perceba, portanto, que no laço externo usamos o nome de variável `k`, e, no laço interno, usamos `i`. Contudo, poderíamos usar qualquer outra letra ou nome, desde que não seja uma palavra reservada da linguagem. (Palavras reservadas são aquelas que são usadas em comandos do próprio Python, como *for*, *in*, *if*, *def*, etc.)

É interessante mencionar que, embora seja comum, em algumas situações, termos laços (*loops*) aninhados, isso pode e deve ser evitado sempre que possível. Mas como podemos evitá-los? Utilizando definições de funções para "ensinar" a linguagem um novo comando que nós mesmos criaremos!

Indague os alunos de modo similar ao que segue: *"Vocês perceberam que tivemos que repetir de novo todos aqueles comandos para fazer a linha tracejada? E se tivesse uma maneira da tartaruga **aprender** como fazer a linha tracejada de uma vez por todas, e depois só pedir para ela desenhar de novo sem ter que repetir todos os comandos?"*

Faça uma dinâmica com os alunos: convide um(a) aluno(a) para ficar de pé, e faça de conta que ele(a) é um robô, e que você dará os comandos a ele(a). Peça para ele(a) andar alguns passos à frente, depois girar o corpo, e depois dar mais alguns passos até ficar de frente para a porta da sala. Depois, peça para voltar de onde saiu, e diga algo assim: *"Agora eu quero que você faça de novo os mesmos comandos, mas quero também que você aprenda/memorize esses comandos."*. Repita os "comandos" para que ele(a) chegue até a porta e diga: *"Muito bem, agora você aprendeu o comando 'vá até a porta'. Sempre que eu te pedir 'vá até a porta', você*

deve repetir os comandos exatamente como fez. Vamos testar?". E então, peça para ele(a) voltar ao ponto inicial e diga: *"Fulano(a), 'vá até a porta'"*. O(a) aluno(a) então deve "executar" novamente os comandos, sem você explicar os passos que deve seguir.

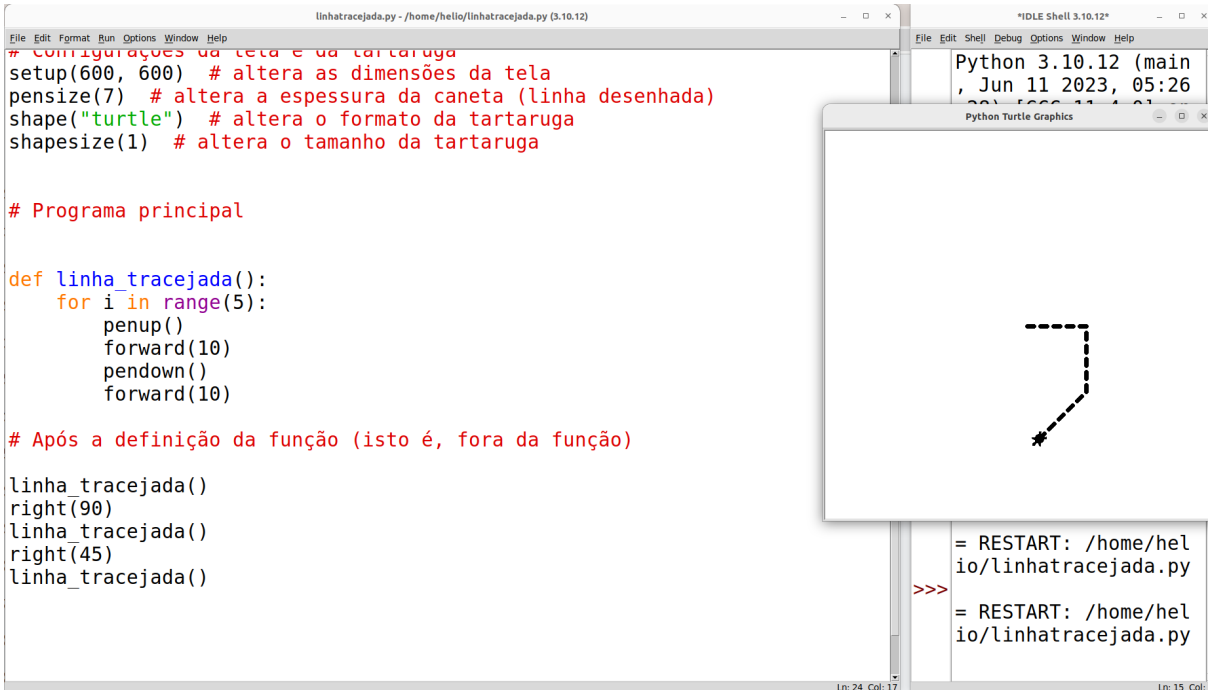
Esta dinâmica tem a intenção de facilitar a compreensão do conceito de **definir uma função** na linguagem, isto é, **fazer a tartaruga aprender um novo comando**. Explique que, da mesma forma que o(a) colega "aprendeu" como "ir até a porta", também podemos fazer a tartaruga aprender sequências de comandos. Mostre que vamos fazer a tartaruga aprender a desenhar uma linha tracejada, como feito anteriormente. Para isso, usaremos o comando `def`:

```
def linha_tracejada():
    for i in range(5):
        penup()
        forward(10)
        pendown()
        forward(10)
```

Perceba que colocamos o código responsável apenas por desenhar a linha tracejada **dentro** da definição da função chamada `linha_tracejada`. Esse nome é a gente que escolhe. Lembrando que os nomes de funções não devem ter espaços, por isso usamos um "_" (*underline*) para juntar as duas palavras. Mas, para que serve uma função? Aí é que está a "mágica": agora podemos chamar a função quantas vezes quisermos pelo seu nome, e assim conseguimos desenhar várias linhas tracejadas. Escreva, após a definição da função (sem indentação, pois não estará dentro da função), algumas chamadas à função, como colocado abaixo:

```
def linha_tracejada():
    for i in range(5):
        penup()
        forward(10)
        pendown()
        forward(10)

# Após a definição da função (isto é, fora da função)
linha_tracejada()
right(90)
linha_tracejada()
right(45)
linha_tracejada()
```

Veja abaixo como fica:



Peça aos alunos que insiram várias vezes `linha_tracejada()`, para que entendam que a tartaruga "aprendeu" um novo comando. Note que devemos incluir o "abre e fecha parênteses" (`()`) na frente do nome da função.

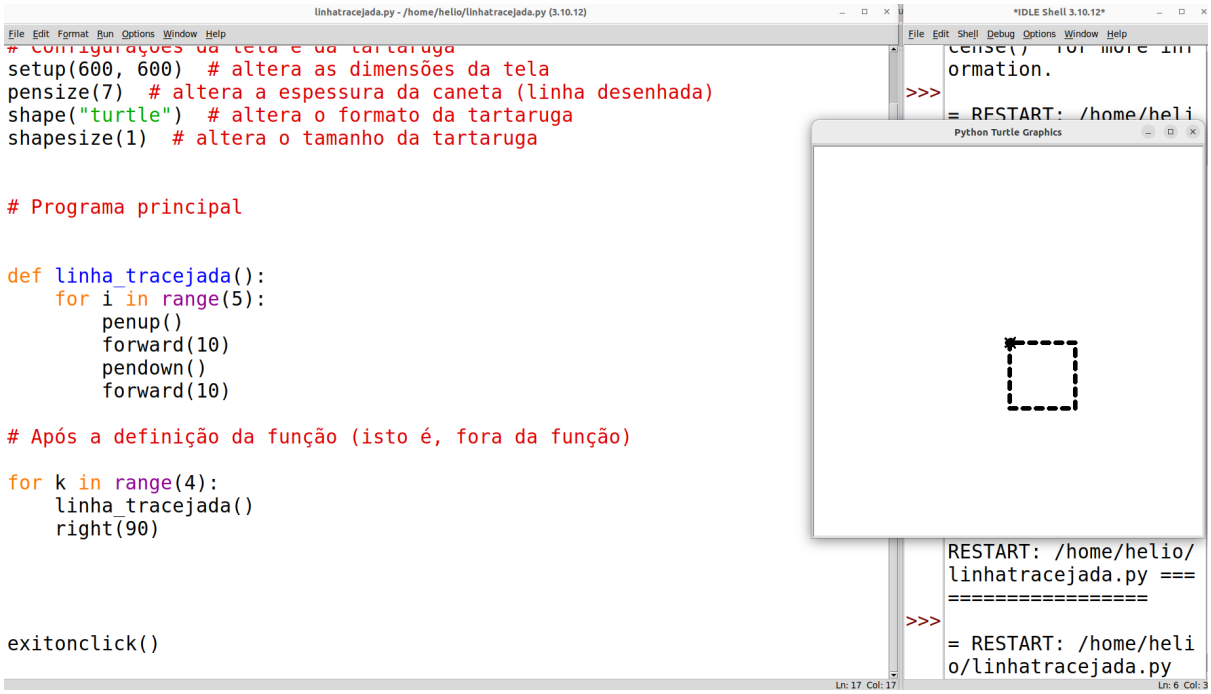
Note que pudemos desenhar três linhas tracejadas simplesmente **chamando** (ou **invocando**) a função três vezes. Ou seja, definimos a função uma única vez, e agora podemos chamá-la (invocá-la) quantas vezes quisermos.

Perceba também que, ao definirmos uma função com o comando `def`, criamos um novo bloco, isto é, um novo escopo. Logo, tudo o que está **dentro** da função deve estar indentado, isto é, com um espaçamento. Perceba que o `for`, dentro da função, continua sendo um bloco por si próprio. Logo, o que está dentro do `for` continua tendo um espaçamento a mais também.

> **Dica:** não se esqueça que a indentação dos comandos no mesmo nível (isto é, dentro do mesmo escopo/bloco) deve possuir o mesmo espaçamento. Caso contrário, o Python emitirá um erro de sintaxe.

Agora podemos refazer nosso quadrado usando a função `linha_tracejada`, ficando assim:

```
for k in range(4):
    linha_tracejada()
    right(90)
```



Introduziremos agora a ideia de que uma função pode ter **parâmetros de entrada**, fazendo algumas modificações à função `linha_tracejada`.

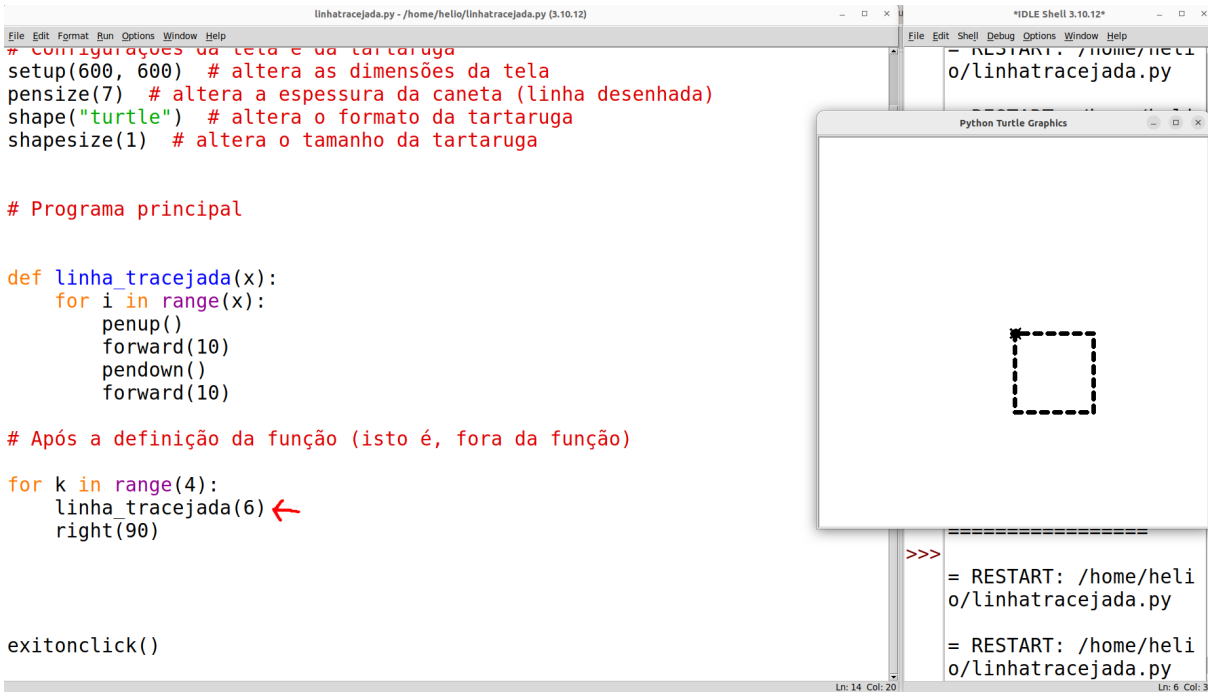
Para isso, pergunte aos alunos o seguinte: *"E se quiséssemos ensinar o programa a desenhar linhas tracejadas, mas de modo que pudéssemos dizer a ele quantos traços devem ser feitos?"* Dê exemplos na lousa de chamadas à função `linha_tracejada`, mas passando para ela a quantidade de traços. Por exemplo, escreva na lousa: `linha_tracejada(6)`, e desenhe uma linha com 6 traços. Em seguida, escreva na lousa `linha_tracejada(7)`, e desenhe uma linha com 7 traços. Faça quantos exemplos achar necessário. Em seguida, mostre como podemos modificar nossa função `linha_tracejada` no Python para que receba a quantidade de traços:

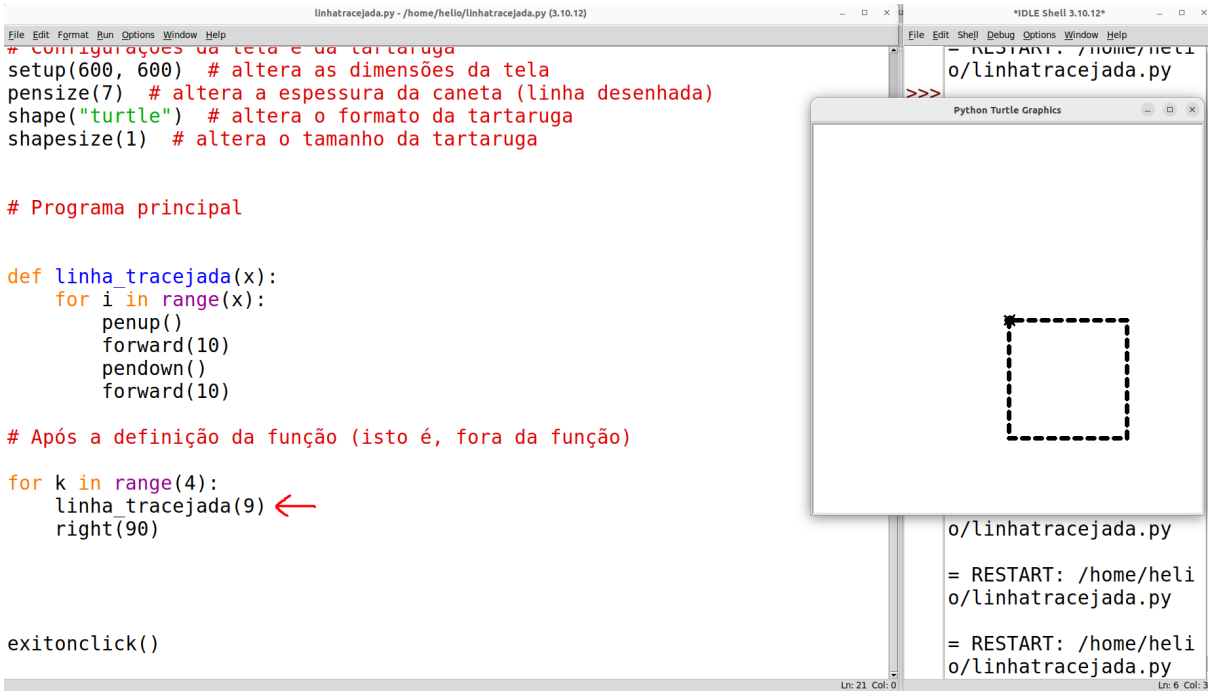
```
def linha_tracejada(x):
    for i in range(x):
        penup()
        forward(10)
        pendown()
        forward(10)
```

O que fizemos? Definimos que a função `linha_tracejada` recebe por parâmetro um número que será colocado em uma variável `x`. Explique que uma variável é como se fosse um espaço, ou uma "caixinha", na qual colocamos um valor (e.g. um número). Essa caixinha possui um nome, que é uma letra ou palavra que escolhemos. Assim, dentro dessa variável `x` colocaremos a quantidade de vezes que um traço será desenhado. Mas quando colocamos um valor na variável `x`? No momento em que chamamos a função! Quando chamarmos a função, não mais escreveremos `linha_tracejada()`, mas sim, por exemplo, `linha_tracejada(6)`, que desenhará uma linha com 6 traços. Assim, a função será executada colocando o número `6`

sempre que aparecer o **x**. Note que a variável **x** é usada dentro do `range`, que define a quantidade de vezes que o `for` será executado. Logo, quando ***x=6***, o `for` (e tudo o que estiver dentro dele) executará 6 vezes.

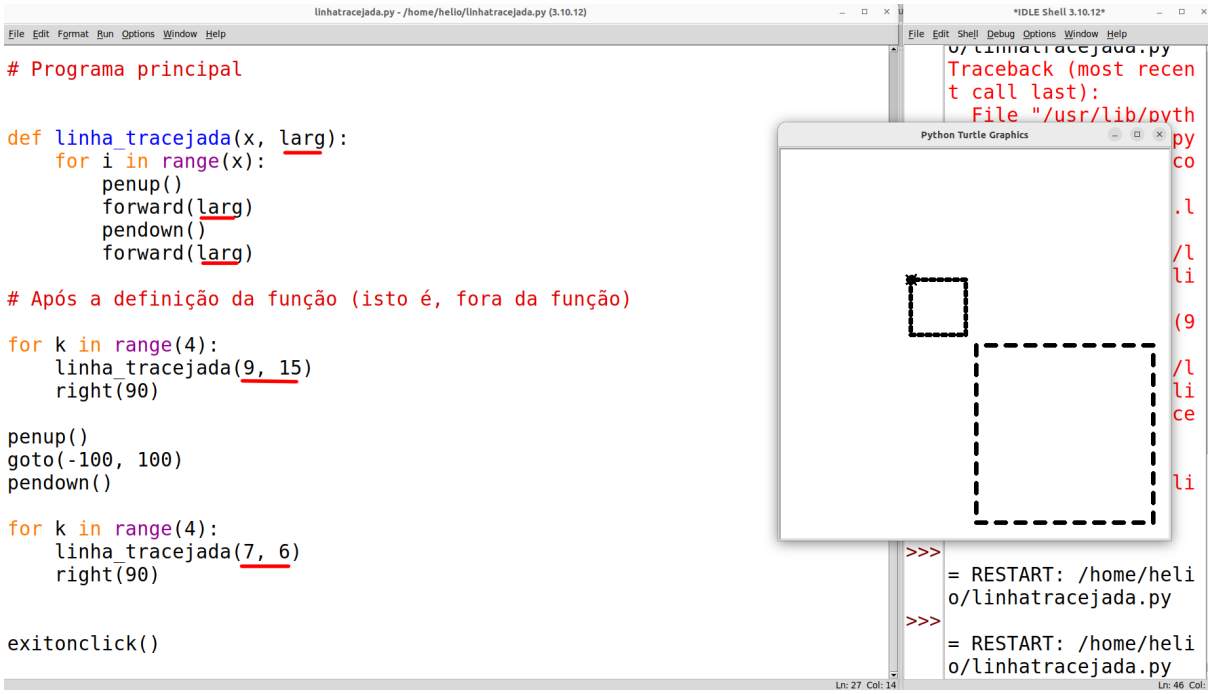
Altere na chamada da função o valor do parâmetro (variável) passado, colocando diferentes valores, como 9, 3, 5, etc. Veja abaixo alguns exemplos.





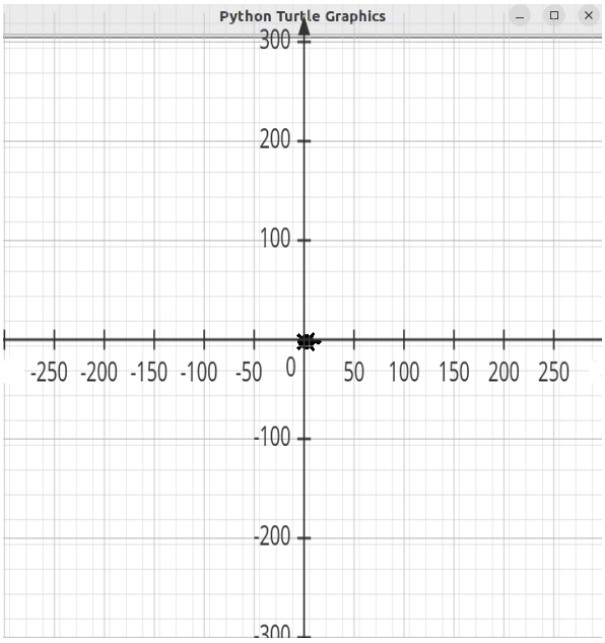
Pronto, agora podemos fazer linhas tracejadas com diferentes quantidades de traços, e assim também quadrados tracejados de diferentes tamanhos. Essa é a “mágica” das funções!

Seguindo a mesma linha de raciocínio, é possível também definir um segundo parâmetro: a largura de cada traço. Basta repetir os mesmos passos para criar um parâmetro chamado `larg`. Veja abaixo como ficaria:



Faça testes com diferentes valores para `x` e `larg`, deixando claro o quanto é interessante parametrizar funções, pois permite criar diferentes formas de linhas tracejadas com pouco esforço.

> **Nota:** os comandos `penup(); goto(-100, 100); pendown()` entre os dois laços servem para fazer a tartaruga se deslocar para uma outra parte da tela sem desenhar. O comando `goto` (do inglês, *"go to"*, que significa "vá para"), especificamente, faz a tartaruga ir diretamente até as coordenadas *x=-100, y=100*. Se desejar, esse pode ser um gancho para discutir o conceito de **plano cartesiano**, pois a tela do Python Turtle, onde a tartaruga desenha, é um plano cartesiano, cujo centro da tela é a posição (0,0). Veja a figura abaixo para uma ilustração de como os eixos *x* e *y* estão posicionados na tela da tartaruga.

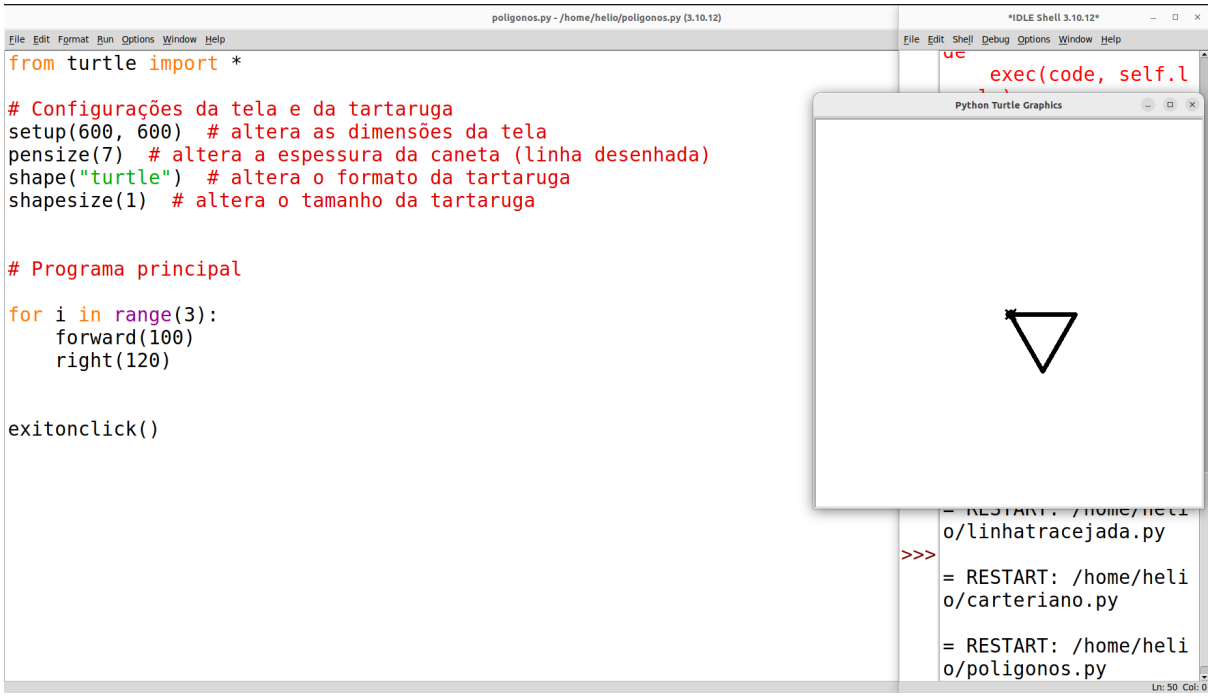


Polígonos regulares e ângulos

Salve o arquivo atual e peça para criarem um novo arquivo (*File → New*). Salve esse novo arquivo com o nome *poligonos.py*. Copie também as primeiras linhas, de configuração, para o novo arquivo.

Vamos agora fazer os alunos brincarem um pouco com diferentes polígonos regulares. Explique que o quadrado é um polígono de 4 lados, mas que existem outras formas geométricas com diferentes quantidades de lados. Pergunte aos alunos se eles conhecem algum outro polígono. Provavelmente dirão “triângulo”, ou talvez até mesmo “hexágono”, etc. Diga que vamos desenhar um triângulo, e faça junto com eles:

```
for i in range(3):      (questione, "por que 3"?)
    forward(100)
    right(120)          (diga que em breve você vai explicar o porquê de ser 120)
```

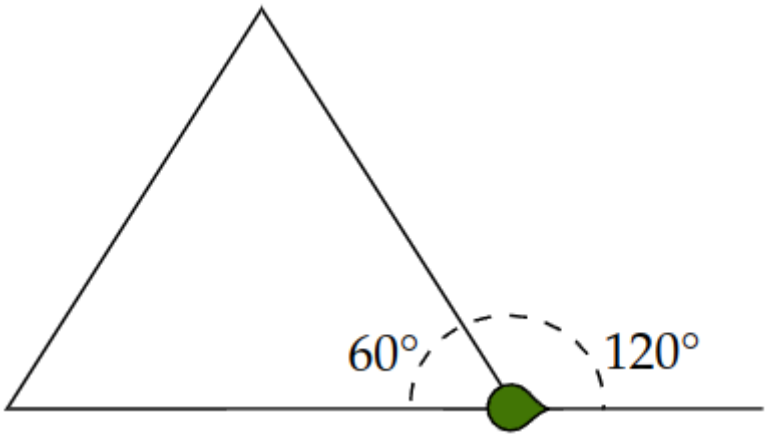


Pergunte: *"Por que '`range(3)`'?"*. Deve ser, neste ponto, evidente, que se trata da quantidade de lados do triângulo. *"E quanto ao ângulo, por que 120?"* Deixe em mistério por enquanto e explique que logo entenderão o porquê do 120 para o ângulo do triângulo, e o porquê do 90 para o ângulo do quadrado.

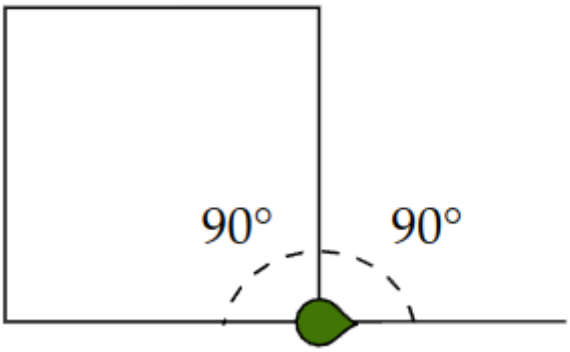
Peça para executarem. Agora lance um **desafio**: *"E se eu quiser fazer um hexágono? Um hexágono tem 6 lados. Como eu poderia começar?"* Deixe-os começarem e ajude somente aqueles que não entenderam. Logo, alguns deles vão tentar algo como '`for i in range(6): forward(100); right(...)`', mas provavelmente não conseguirão inserir o ângulo correto. Escreva na lousa o código para cada um dos polígonos que já conseguiram fazer:

```
for i in range(4): forward(100); right(90)  → Quadrado (desenhe o quadrado)

for i in range(3): forward(100); right(120)  → Triângulo (desenhe o triângulo)

for i in range(6): forward(100); right(???) → Hexágono (desenhe o hexágono)
```

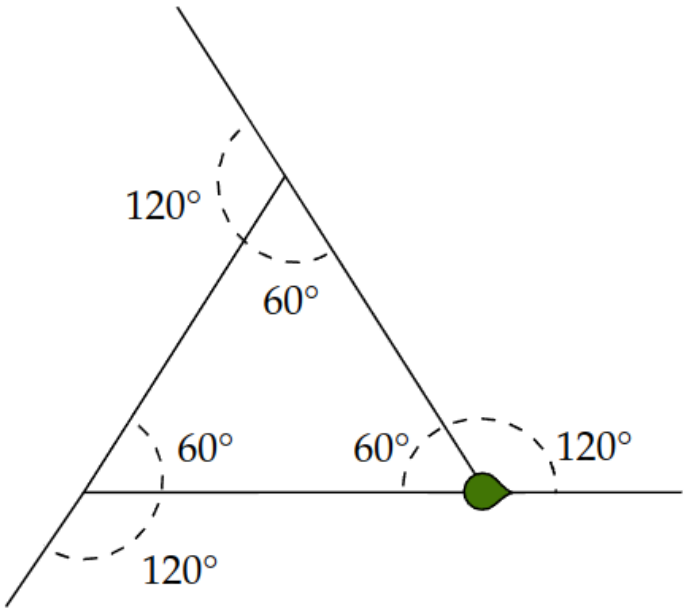
Neste ponto os alunos ficarão bem curiosos para descobrirem qual o ângulo correto para o hexágono. É uma oportunidade para, na lousa, explicar como funcionam os ângulos de um polígono. Por exemplo, desenhe um triângulo e mostre quantos graus a tartaruga tem que girar após traçar um lado. Com isto, é possível explicar aos alunos a questão dos **ângulos suplementares** e dos **ângulos internos** de um polígono regular. Mostre que os ângulos internos de cada ponta do triângulo têm 60º, e que a tartaruga tem que girar 120º, pois 180º - 60º = 120º, que é o ângulo suplementar de 60º.



Faça o mesmo para o quadrado, mostrando que o ângulo suplementar de 90º é 90º.

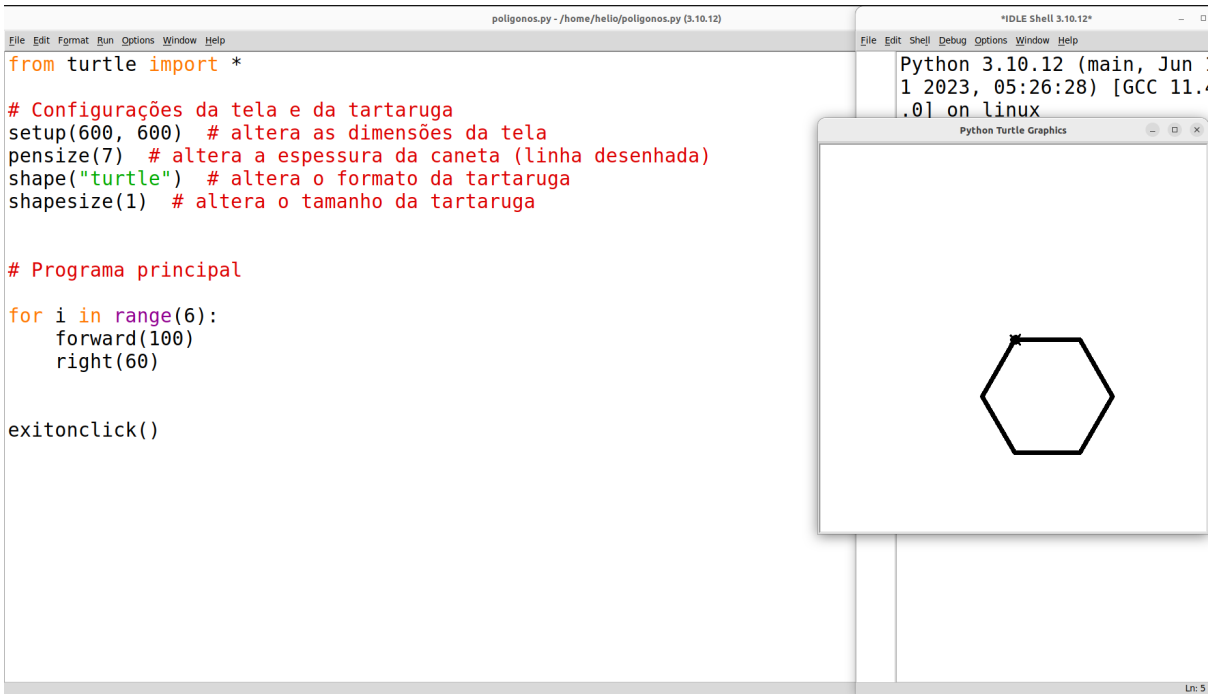


Mostre também que a soma dos ângulos internos sempre totaliza 180º, e que se somarmos os ângulos suplementares dos ângulos internos teremos 360º (por exemplo, 120 + 120 + 120, no caso do triângulo). Assim, você pode chegar à conclusão de que para 4 lados temos que girar 90º, pois 360º/4 = 90º, e que para 3 lados temos que girar 120º, pois 360º/3 = 120º.

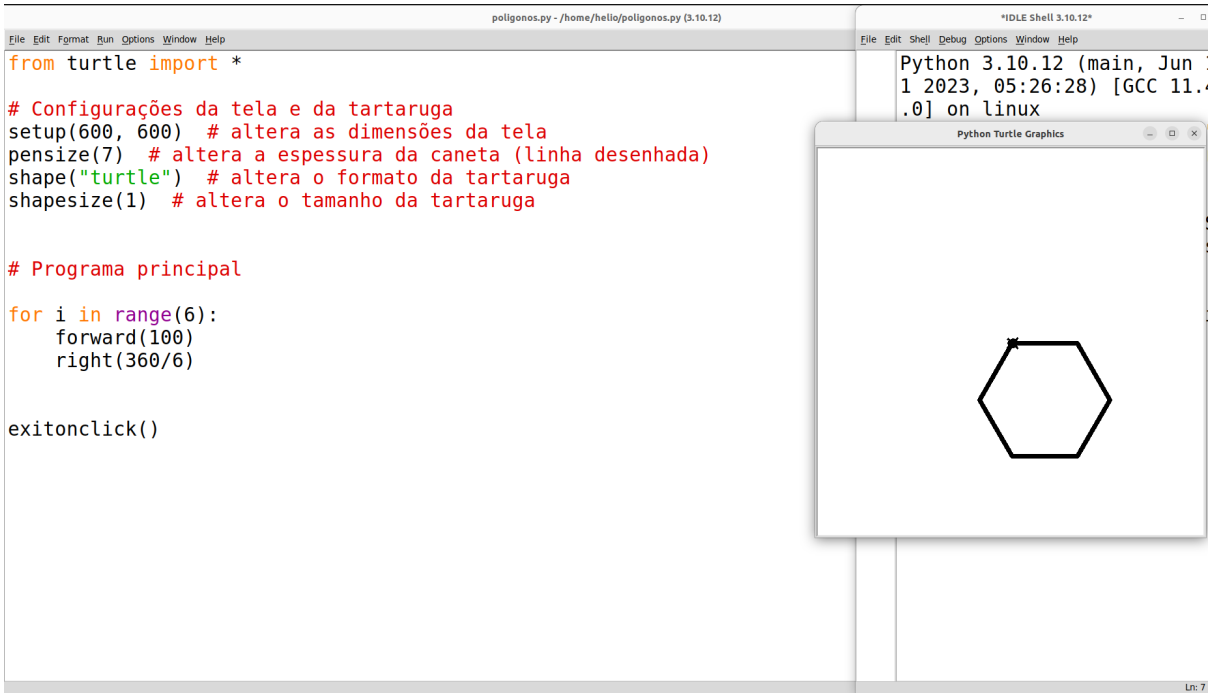


Tendo evidenciado esse padrão/fórmula ($360/lados$), peça aos alunos que calculem o ângulo que a tartaruga tem que girar para o hexágono. Deve-se chegar à conclusão que é 360º/6 = 60º. Feito isso, complete o código e peça que os alunos executem:

```
for i in range(6):
    forward(100)
    right(60)
```



Pode ser interessante também mostrar que é possível, na programação, colocar uma **expressão aritmética** no lugar de um valor. Por exemplo, em vez de colocar `right(60)`, podemos colocar `right(360/6)` (vire à direita por 360 dividido por 6 graus). Veja abaixo:



Uma vez que isso tenha sido compreendido, lance o desafio para os alunos fazerem o pentágono, o heptágono e o octógono. Elogie e recompense os que conseguirem desenhar mais. Depois, escreva na lousa as soluções dos desafios propostos:

```
for i in range(4): forward(100); right(90)  → Quadrado
for i in range(3): forward(100); right(120)  → Triângulo
for i in range(6): forward(100); right(60) → Hexágono
for i in range(5): forward(100); right(72) → Pentágono
for i in range(7): forward(100); right(51) → Heptágono
for i in range(8): forward(100); right(45) → Octógono
```

Pergunte: *"E se eu quiser fazer um hexágono maior, o que tenho que fazer?"*. Espere pela resposta e mostre que basta aumentar a distância com que se anda para frente:

```
for i in range(6): forward(150); right(60) → Hexágono
```

Agora podemos trabalhar com os alunos a habilidade de **identificação de padrões**, uma das habilidades relacionadas ao **pensamento computacional**. Comece a raciocinar com os alunos da seguinte forma: *"O que tem de comum nos códigos de todos esses polígonos?"*. Vá termo por termo de cada linha perguntando: *"Isto muda?"*. Por exemplo:

```
for i in range(4): forward(100); right(90)
for i in range ...
for i in range ...
```

```
...
```

Pergunte: *"Dentre todos eles, o 'for i in range' muda?"*. A resposta será **"não"**.

```
for i in range(4): forward(100); right(90)
for i in range(3) ...
for i in range(6) ...
...
```

Pergunte: *"Dentre todos eles, esse número muda (se referindo à quantidade de lados)?"*. A resposta deve ser **"sim"**.

Abaixo de todas as linhas, copie cada termo que não muda, e, no lugar dos termos que mudam, escreva um ponto de interrogação (ou outro símbolo que represente algo que não sabemos, ou que pode acomodar algum valor, como uma "caixinha" ou "quadradinho"). No final, essa última linha ficará assim:

```
for i in range(?): forward(?); right(?)
```

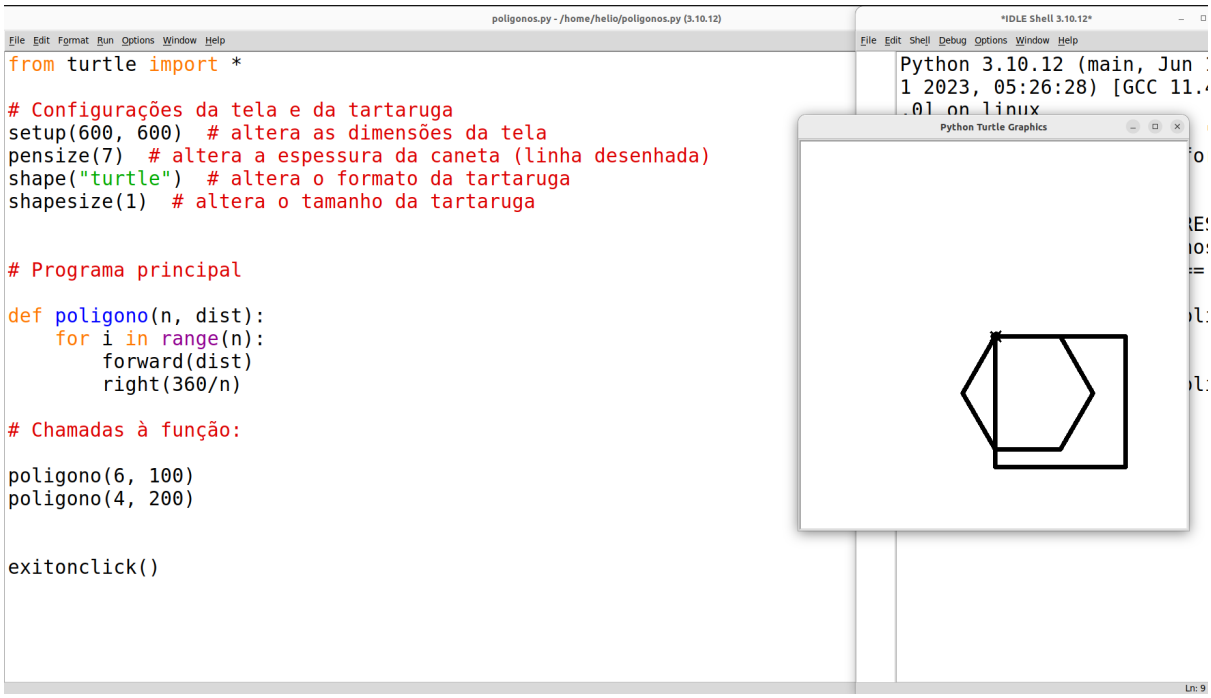
Frente a cada interrogação, questione: *"O que vai nessa interrogação?"*, ou *"O que podemos colocar para substituir a interrogação?"*. Reflita com os alunos, com base nos exemplos de polígonos escritos acima, o que cada número no lugar da interrogação representa. Tente com eles chegar à conclusão de que: a primeira interrogação diz respeito à ***quantidade de lados***; a segunda diz respeito ao ***tamanho dos lados***; e a última diz respeito à operação ***360÷(quantidade de lados)***. ao final, você deve demonstrar para os alunos que aquela expressão se tornará uma **fórmula** para desenhar qualquer polígono, desde que se saiba a quantidade de lados e o tamanho dos lados. Troque as interrogações por letras ou palavras da seguinte forma:

```
for i in range(n): forward(dist); right(360/n)
```

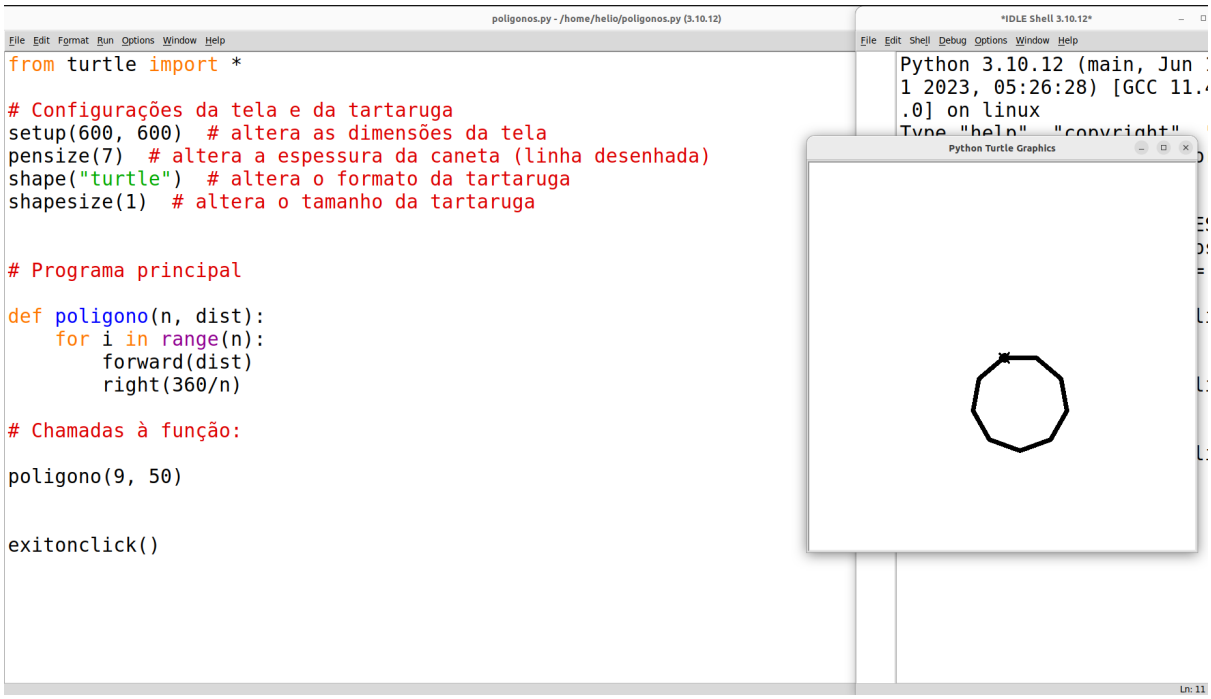
Aqui temos uma ótima oportunidade de falar mais um pouco sobre o conceito de **variáveis**. Explique que uma variável é como se fosse uma caixinha com um nome escrito do lado de fora, mas que, dentro da caixa, tem um número (valor). Neste caso, cabe a nós definir o número que será posto dentro dessa caixinha. Para isso, vamos precisar definir **uma função** que recebe **parâmetros (variáveis) de entrada**. Em outras palavras, temos que fazer com que a tartaruga **aprenda** um novo comando chamado `poligono`, porém, sempre que formos chamar (invocar) essa função, temos que informá-la quais são os valores que queremos para essas variáveis de entrada. Para isso, criaremos uma nova **função** com 2 (duas) variáveis de entrada: `n` e `dist`, onde `n` representa a quantidade de lados do polígono, e `dist` representa o tamanho de cada lado, ou seja, a distância que a tartaruga deve percorrer para desenhar cada lado do polígono. Veja abaixo:

```
def poligono(n, dist):
    for i in range(n):
        forward(dist)
```

```
        right(360/n)

# Chamadas à função:
poligono(6, 100)
poligono(4, 200)
```



Agora peça a eles que tentem pedir à tartaruga que desenhe um polígono de 9 lados. Veja abaixo:



Solicite aos alunos que peçam à tartaruga que desenhe polígonos de diferentes lados com a nova função.

Um gancho para introduzir funções matemáticas

Lance mais um **desafio** aos alunos: peça que façam a tartaruga aprender como desenhar um triângulo, porém utilizando agora a função `poligono` que acabamos de definir. Para isto, escreva na lousa alguns exemplos de como criar um triângulo:

```
poligono(3, 50)
poligono(3, 100)
poligono(3, 150)
```

Mais uma vez, faça o procedimento de verificar quais os termos que mudaram e aqueles que permanecem inalterados em cada um dos exemplos. O resultado ficará assim:

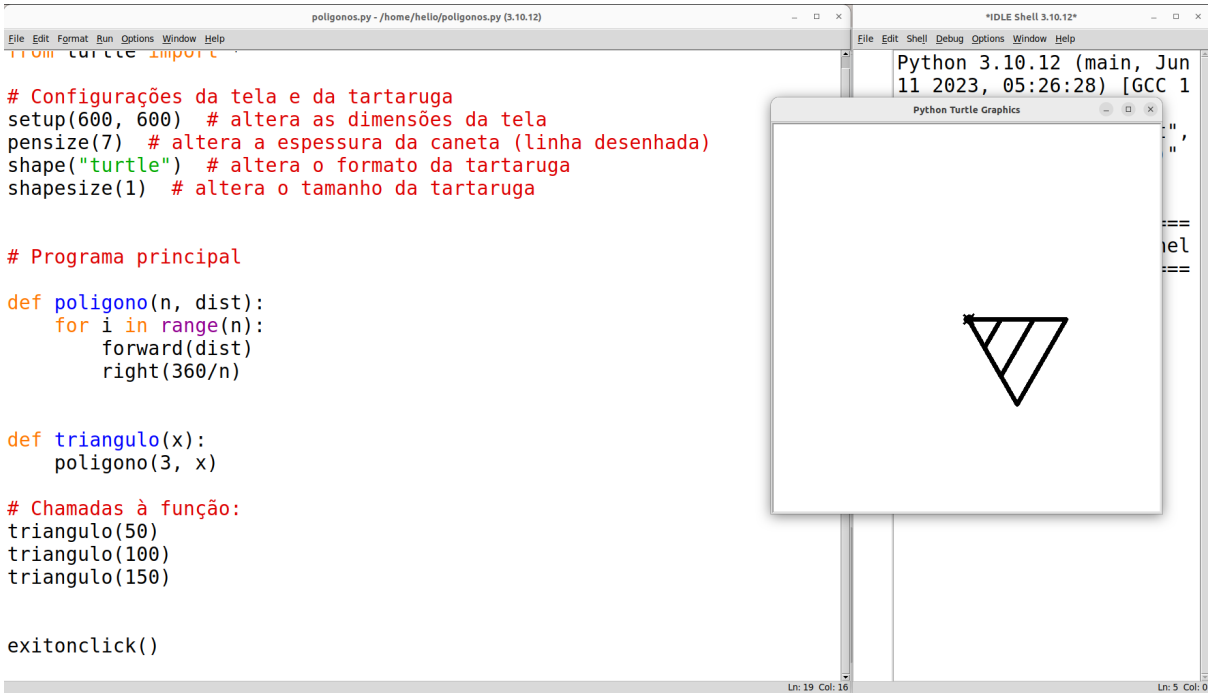
```
poligono(3, ??)
```

Pergunte aos alunos o que representa a interrogação. Deve-se chegar à conclusão de que se trata do **tamanho dos lados** do triângulo. Atribua um nome a essa "incógnita", por exemplo, *x*:

```
poligono(3, x)
```

Agora peça aos alunos que escrevam a função `triangulo`. Espere uns minutos (para eles tentarem) e depois mostre a solução:

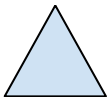
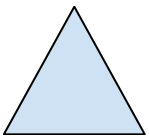
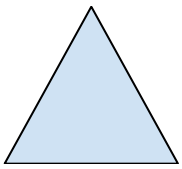
```
def triangulo(x):
      poligono(3, x)

# Chamadas
triangulo(50)
triangulo(100)
triangulo(200)
```



Note que temos duas funções definidas: `poligono`, que já tínhamos definido anteriormente, e a nova função `triangulo`, que utiliza a função `poligono`, mas que só possui um único parâmetro (variável) de entrada: **`x`**, que define o tamanho dos lados do triângulo.

Assegure-se de que eles percebam que diferentes números que você passa como variável **`x`** para a função `triangulo` resultam em triângulos de diferentes tamanhos. Esta é uma oportunidade para iniciar a explicação sobre **funções matemáticas**:

Faça na lousa uma tabela mais ou menos assim:

x	triangulo(x)
20	
40	
60	
80	

Perceba que se trata de uma tabela de função, cujo *domínio* é um número e cuja *imagem* é uma **forma geométrica**! Acreditamos que essa seja uma maneira interessante de explicar como funcionam as funções matemáticas, conforme também constataram autores como Schanzer, Felleisen e Krishnamurthi[8], dentre outros pesquisadores, os quais propuseram uma abordagem para o ensino de álgebra com programação chamada *Bootstrap.*

Você pode mostrar como a ideia é parecida com funções que só envolvem números. Por exemplo *f(x) = 2x*:

x	f(x)
1	2
2	4
3	6
4	8

Desafio 1: como um desafio um pouco mais interessante, peça aos alunos que definam uma função chamada `poligono_tracejado`, que desenha um polígono usando a linha tracejada definida anteriormente. Dica: A ideia é parecida com o quadrado tracejado que fizemos anteriormente — basta trocar o comando `forward` pela chamada à função `linha_tracejada`.

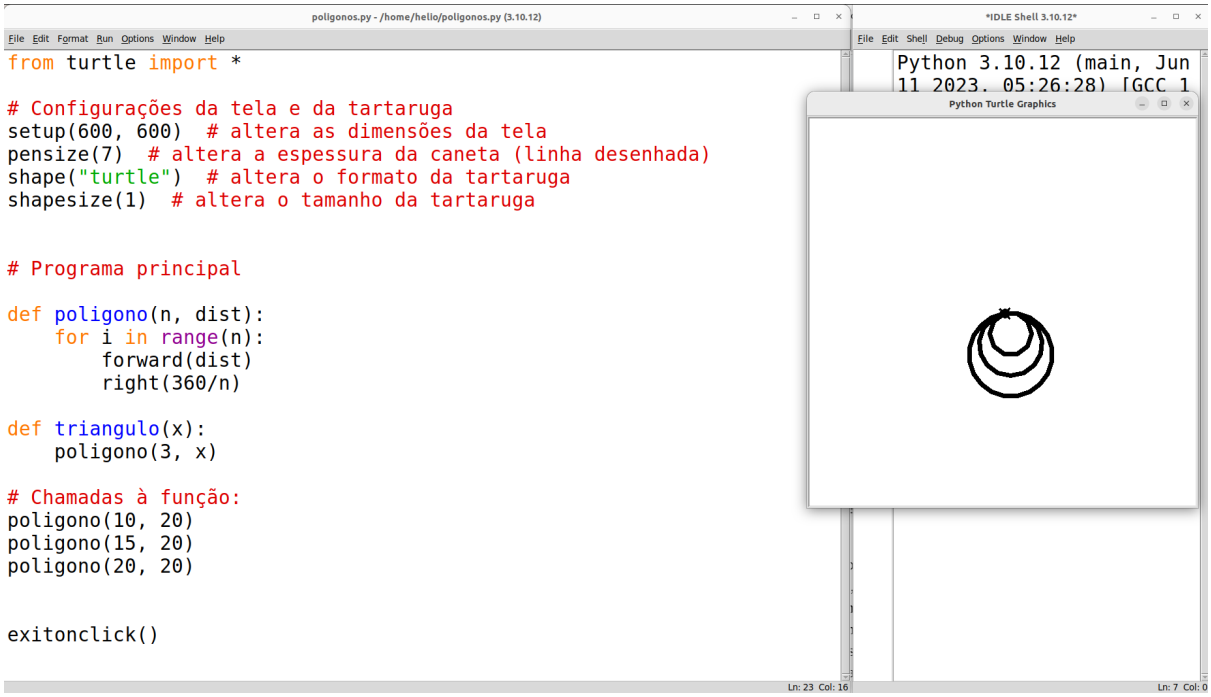
Desafio 2: peça aos alunos que criem as funções `quadrado`, `hexagono` e `pentagono`, do mesmo modo que fizeram com `triangulo`.

De polígonos para círculos

Peça aos alunos que tentem criar polígonos com mais lados, por exemplo:

```
poligono(10, 20)
poligono(15, 20)
poligono(20, 20)
```

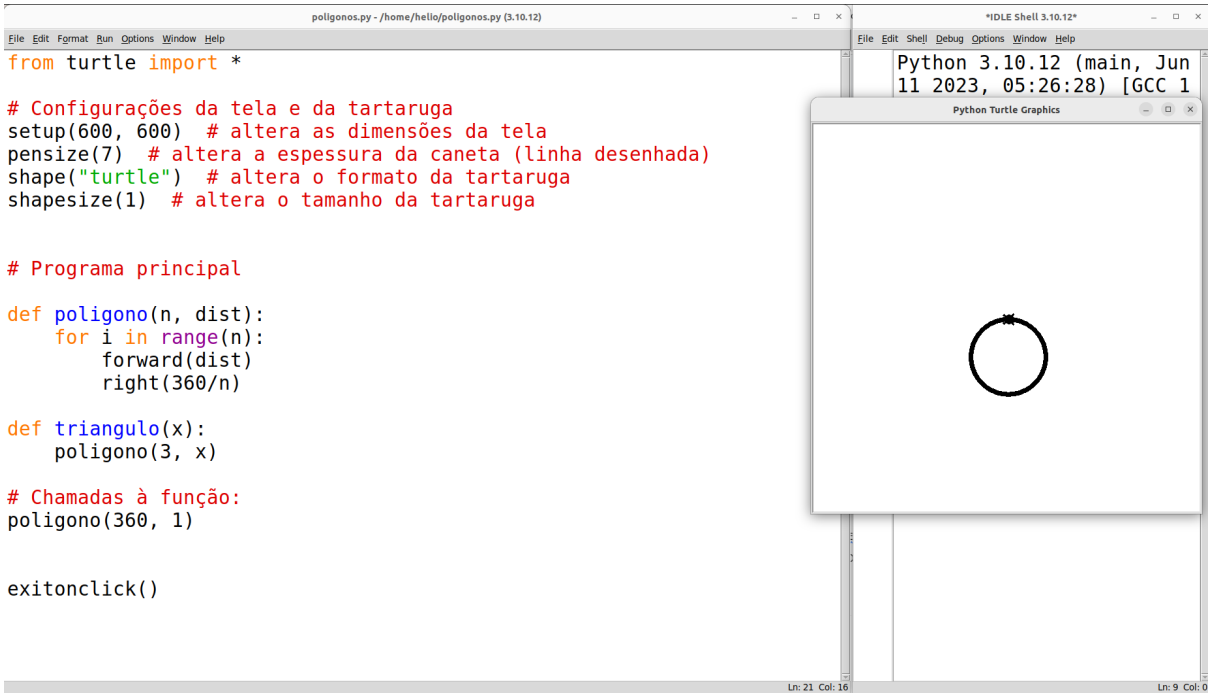
[8] SCHANZER, Emmanuel et al. Transferring skills at solving word problems from computing to algebra through Bootstrap. In: Proceedings of the 46th ACM Technical symposium on computer science education. 2015. p. 616-621.



Por si próprios, os alunos começarão a perceber que, quanto mais lados tiver um polígono, mais parecido ele ficará com um círculo! Assim, pergunte: *"Então, se eu quiser fazer um círculo perfeito, preciso fazer o quê?"*. Instigue-os a responderem que basta colocar mais lados. Não reprima se algum aluno disser algo do tipo: *"É só fazer 1 milhão de lados"*. Ele(a) não estará errado, por isso concorde com ele(a) — quanto mais lados, mais perfeito será o círculo. No entanto, deixe claro que levaria muito tempo para desenhar 1 milhão de lados, e por isso temos que escolher um número menor.

Explique a eles que todo círculo é um arco de 360º. Esclareça o que é um arco, desenhando arcos com diferentes ângulos na lousa, e então mostre por que o círculo tem 360º. Portanto, mostre que seria interessante se pudéssemos fazer o círculo por meio de um polígono com 360 lados. Neste ponto, alguns dos alunos já terão tentado fazer o polígono, porém com tamanhos de lados muito grandes. Não reprima esses alunos, apenas explique que os tamanhos dos lados também devem ser bem pequenos, ou o círculo ficará grande demais para caber na tela. Crie com eles o seguinte código:

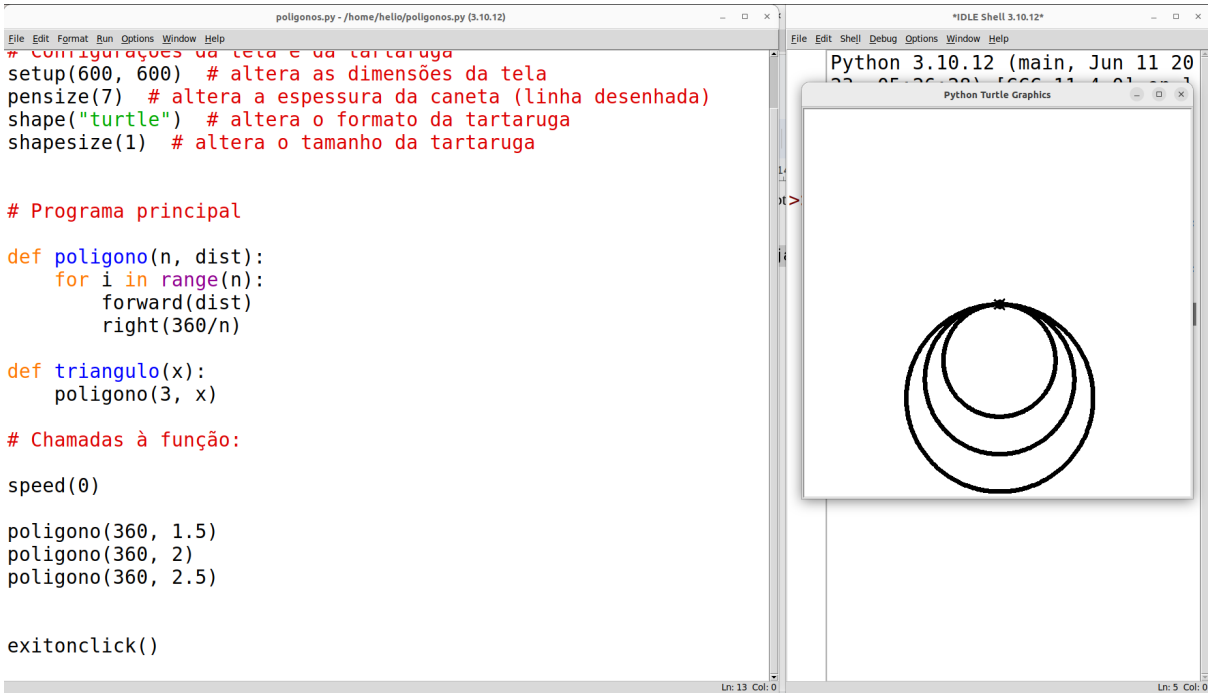
```
poligono(360, 1)
```



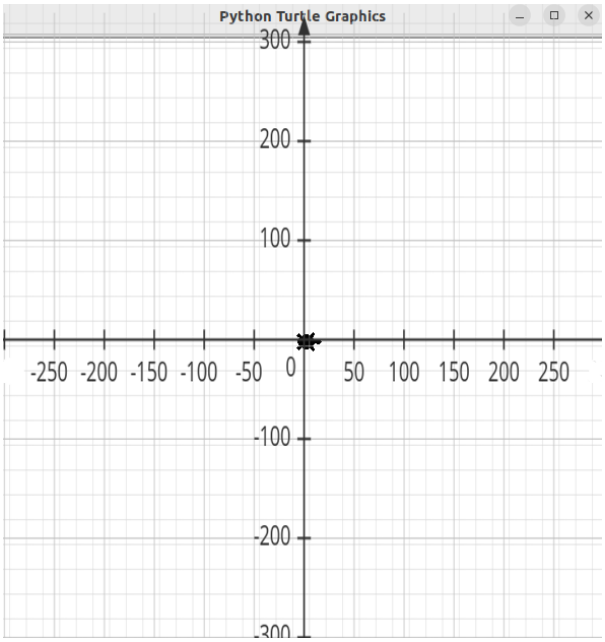
Dica: É possível aumentar a velocidade da tartaruga colocando o seguinte comando antes das chamadas à função `poligono`: `speed(0)`. Para voltar à velocidade normal, inclua o comando `speed(6)`. Esse comando `speed` aceita como parâmetro um número entre 0 e 10, sendo 1 o mais lento, e mais rápido à medida que aumentamos esse número. Também é possível passar o número 0, que é a velocidade mais rápida possível.

Peça para os alunos aumentarem aos poucos o tamanho dos lados, visando obter círculos maiores.

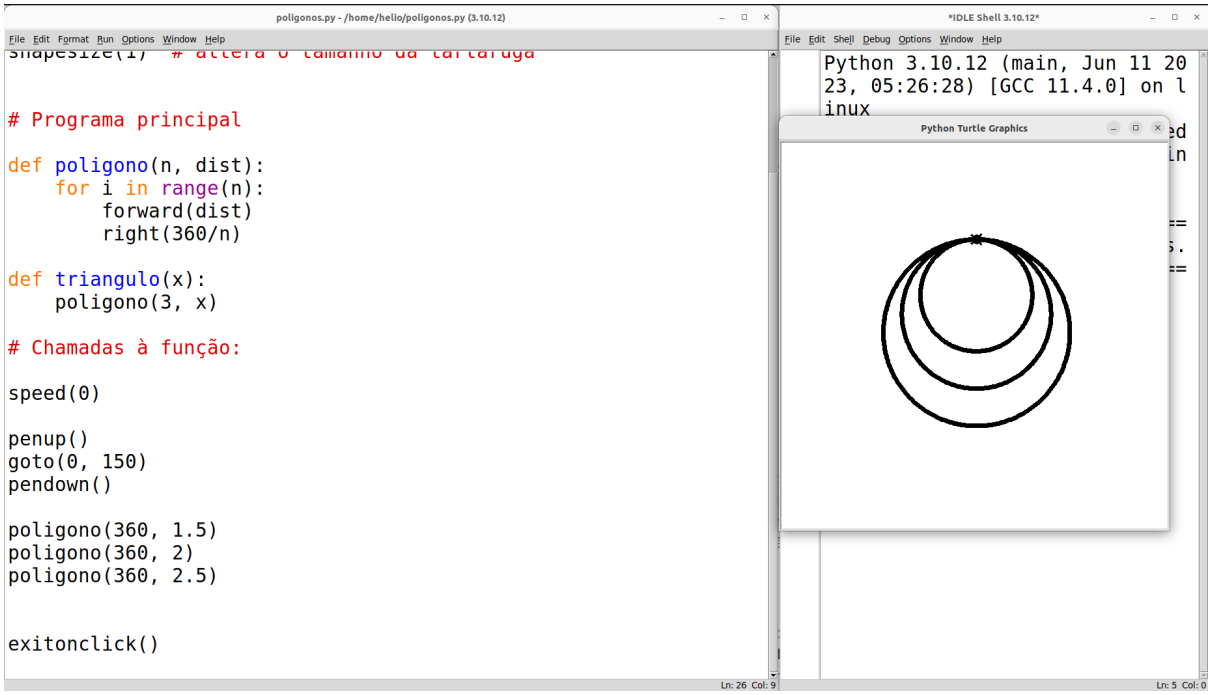
```
poligono(360, 1.5)   # (explique que 1.5 significa 1,5, ou seja, "1 e meio")
poligono(360, 2)
poligono(360, 2.5)
```



Talvez alguns vão indagar sobre o fato de o círculo maior ser tão grande que quase "sai da tela". Para corrigir esse problema, adicione um código antes dos blocos que desenham os círculos para levar a tartaruga para a parte mais acima da tela. Para isso, podemos usar o comando `goto` (do inglês, *"go to"*, que significa "vá para"). Como já explicado na seção **Linhas tracejadas,** esse comando recebe dois parâmetros, o primeiro sendo a coordenada *x* e o segundo a coordenada *y*. Se for utilizar esse comando, é interessante explicar sobre o **plano cartesiano**, e como o ponto exatamente no meio da tela é o ponto (0,0), e que a tela possui dimensões de 600 x 600 pixels (conforme configuramos por meio do comando `setup` no começo do arquivo). Portanto, o ponto mais à esquerda é determinado por *x=-300,* o mais à direita por *x=300*, o ponto na parte superior da tela é *y=300* e na parte inferior é *y=-300*. Veja abaixo uma ilustração do plano cartesiano sobreposto à tela:



Se quisermos, portanto, ir para mais para cima na tela, mantendo-se o eixo *x* centralizado, basta acrescentarmos o comando `goto(0, 150)`, o que faria a tartaruga se mover imediatamente para o ponto (*x=0*, *y=150*). Não se esqueça também de colocar um `penup` logo antes, e um `pendown` logo depois, para que a tartaruga não desenhe no chão enquanto estiver se deslocando para a posição desejada. Veja abaixo como fica:



Agora é o momento de explicar que, geralmente, não desenhamos círculos baseados no tamanho do lado do polígono de 360 lados, mas sim com base no **raio do círculo**. Explique que sempre que queremos desenhar um círculo, usamos o raio. Se você tiver um compasso grande que possa ser usado na lousa, é o modo perfeito de explicar. Se você não tiver um compasso grande, mas tiver compassos pequenos, distribua aos alunos e peça a todos que façam um círculo do mesmo tamanho — assim eles perceberão que conseguem fazer sempre o mesmo círculo se souberem o valor do raio.

Outra alternativa é, caso haja apenas um compasso, fazer uma demonstração em um papel sobre uma mesa, chamando os alunos para observarem em volta da mesa. Caso não disponha de um compasso, use a criatividade para demonstrar que é possível desenhar círculos iguais se sabemos o tamanho do raio. Por exemplo, trace duas linhas perpendiculares, isto é, com ângulo de 90º entre elas, uma na vertical e outra na horizontal, e desenhe o arco de 90º em torno dessas linhas; faça o mesmo para os lados opostos de cada linha (formando uma cruz, como no plano cartesiano), e desenhe os outros arcos de 90º até formar uma circunferência. Com isso, demonstre que as 4 linhas que saem do centro têm o mesmo comprimento, que é o raio.

Portanto, explique que o ideal seria se pudéssemos ordenar à tartaruga algo como: *"Desenhe um círculo de raio 50"*.

> **Opcional:** Aqui pode ser interessante também, se houver espaço na sala, fazer uma **dinâmica de grupo** na qual um aluno (ou o próprio professor) fica parado em pé numa

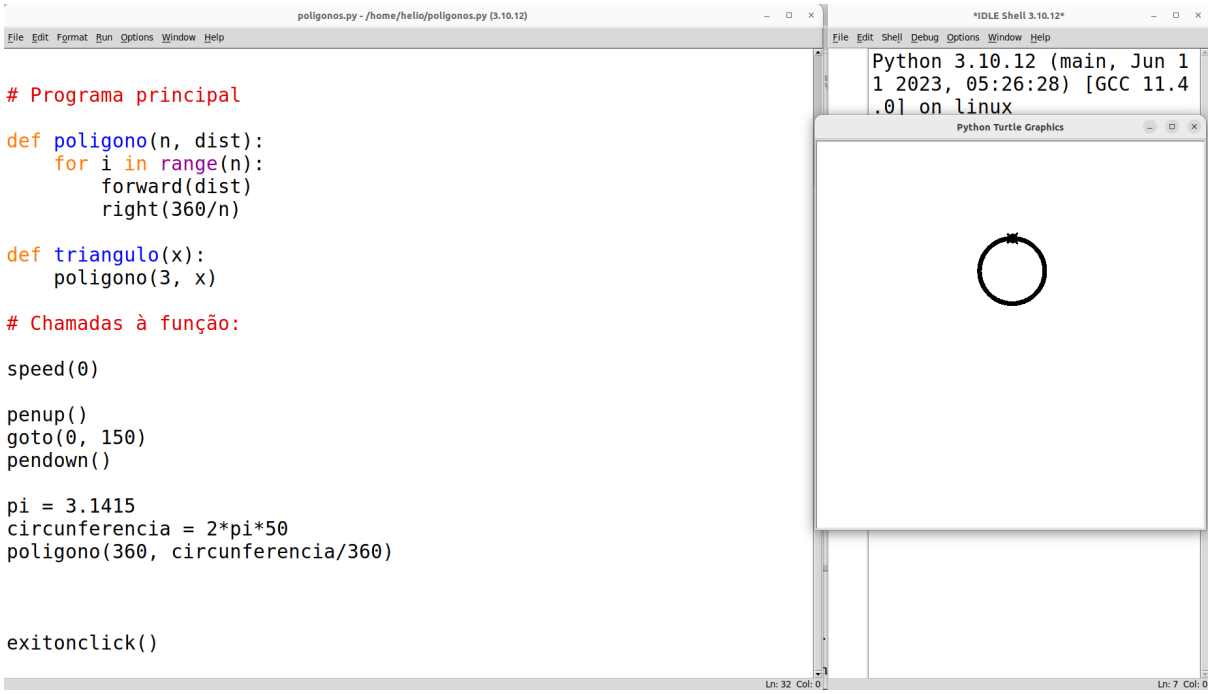
posição que você chamaria de centro, e outro aluno tomaria uma distância de alguns passos deste. Então pediria para o segundo aluno andar em volta do primeiro, de modo que não fique nem mais próximo nem mais distante do centro. Do mesmo modo que a dinâmica de grupo explicada na **Introdução**, você pode fazer de conta que um dos alunos é a tartaruga que recebe instruções. Então, depois de ter aprendido, a turma pode ordenar: *"Fulano(a), faça um círculo com raio de 4 passos"*, ou *"Fulano(a), faça um círculo com raio de 2 passos"*, etc.

"Mas ainda temos um problema: como podemos descobrir o tamanho do lado do polígono/círculo com base no tamanho do raio?". Aqui é o momento em que você deve explicar sobre o comprimento da circunferência, usando a fórmula $2 \times \pi \times r$. Uma forma de explicar isso é usar uma animação, como a que se encontra em http://www.ajudaalunos.com/Quiz_mat/circulo_html/imagens/circulo_pi4.gif.

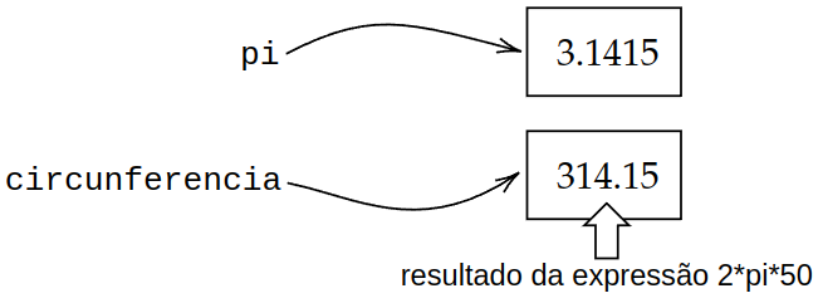
Assim, mostre que sabemos o quanto temos que fazer a tartaruga andar para que desenhe um círculo: basta calcular *circunferencia = 2*pi*raio*. Explique que *pi* é uma constante igual a 3,1415 (o que dá para perceber de modo bem interessante se você mostrar uma animação como a de cima).

"Mas ainda temos um problema: a fórmula do comprimento da circunferência nos mostra apenas a quantidade total de passos que devem ser desenhados para fazer o círculo, mas nós queremos saber apenas a quantidade de passos em cada lado do polígono". Raciocine com eles para que se chegue à conclusão de que, se temos 360 lados, e o total que a tartaruga deve andar para desenhar o círculo é calculado pelo comprimento da circunferência, então temos que fazer uma divisão: temos que dividir o valor de *circunferencia* por 360. Tendo raciocinado assim com os alunos, desenha um círculo de raio 50 com o seguinte comando:

```
pi = 3.1415
circunferencia = 2*pi*50
poligono(360, circunferencia/360)
```



Aqui temos algo novo também: definição de variáveis que não são parâmetros de função. Note que definimos uma variável chamada `pi` para guardar o valor `3.1415`. Em Python, usa-se '.' (ponto) em vez de ',' (vírgula) em números "quebrados"/decimais. Também definimos uma variável `circunferencia` que guarda o resultado do cálculo `2*pi*50`, tal que `50` é o raio que desejamos para o círculo. Novamente, explique que variáveis são como "caixinhas" em que podemos guardar valores. Quando colocamos o símbolo "=" após o nome da variável, e em seguida colocamos um valor ou uma expressão, então o valor ou o resultado da expressão será guardada dentro da "caixinha" da variável. O nome disso é **atribuição de variável**, isto é, atribuímos um valor a uma variável. A figura abaixo ilustra como funcionam as variáveis e o valor que elas armazenam após a atribuição.

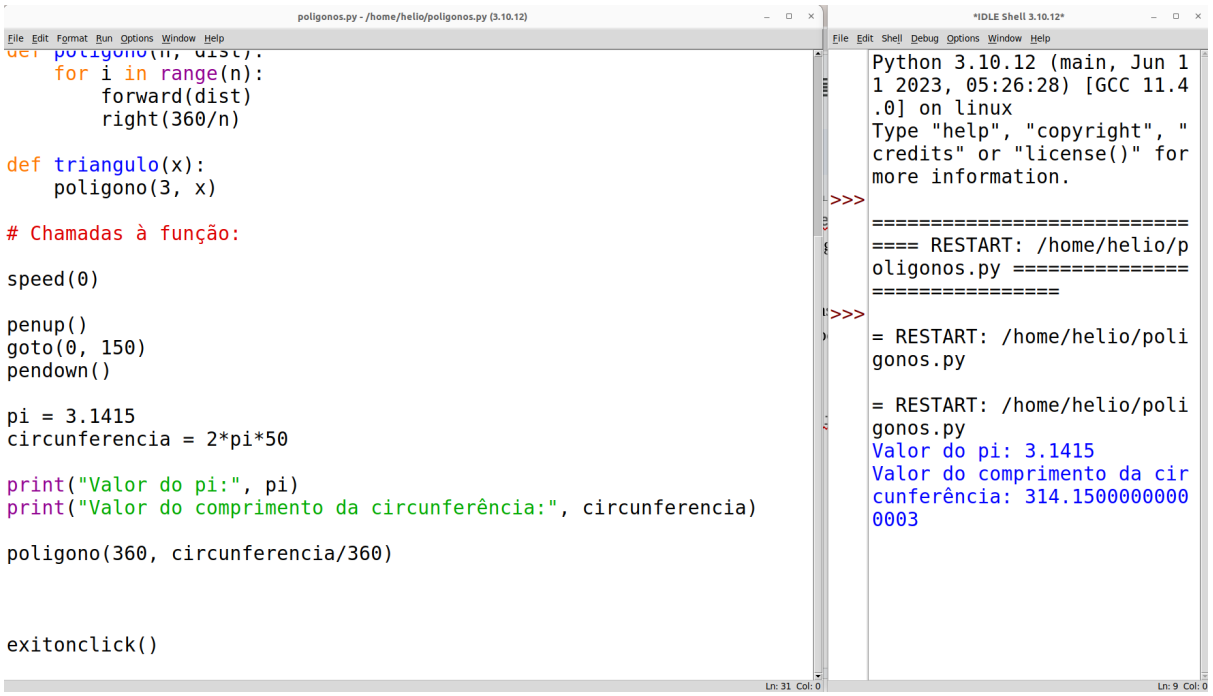


Usamos então a variável `circunferencia` na chamada à função `poligono` para realizar o cálculo da distância que a tartaruga deverá percorrer a cada pedacinho do círculo.

Mostre também que é possível escrever na tela o valor das variáveis. Para isso, basta usar o comando **`print`**, que já tínhamos visto na seção **Introdução ao ambiente**. Coloque esses **`print`**'s antes da chamada a `poligono`:

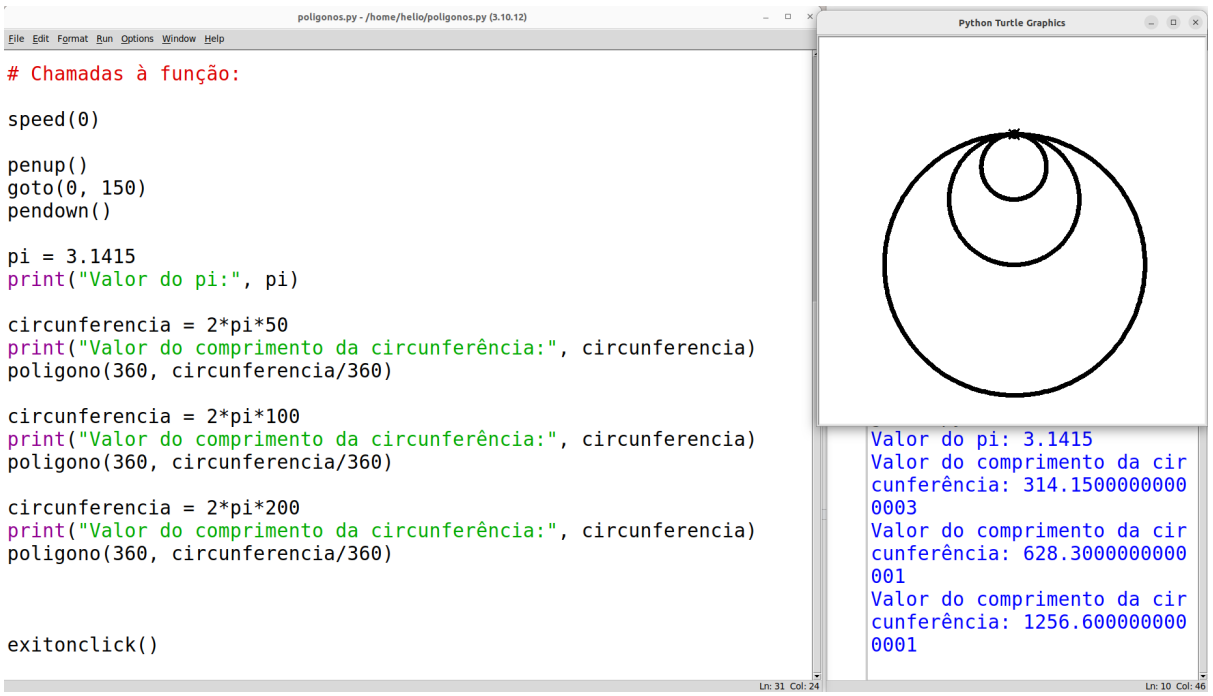
```
print("Valor do pi:", pi)
```

```
print("Valor do comprimento da circunferência:", circunferencia)
```



Note que o resultado dos `print`'s aparece na janela do *shell*, e não na janela da tartaruga.

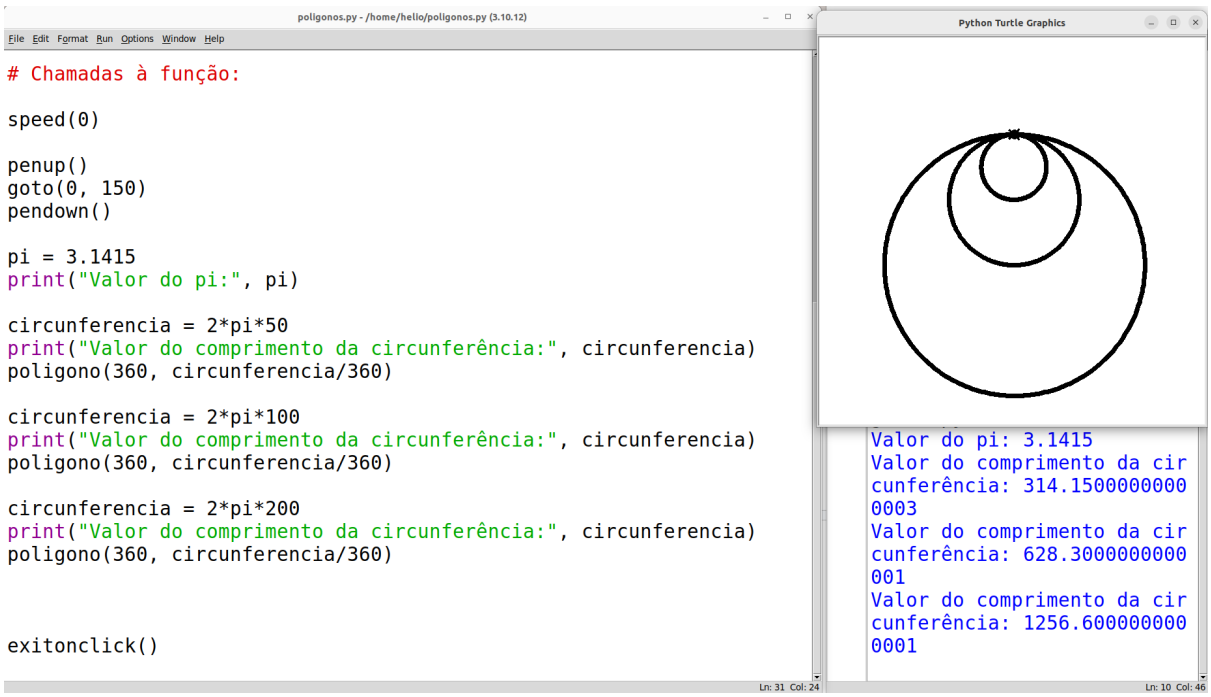
Questione então os alunos sobre a dificuldade em ter que escrever todo esse procedimento quando queremos desenhar um novo círculo, pois note que temos que temos que repetir o cálculo da circunferência e a chamada à função polígono de uma forma específica todas as vezes que quisermos desenhar um novo círculo. Veja abaixo como precisamos proceder para desenhar dois novos círculos, um com raio de 100 e outro com raio de 200.



Note que temos que repetir várias vezes o código, com exceção da definição da variável `pi`, que é um exemplo do que chamamos de **constante** (pois seu valor nunca é alterado durante a execução do programa)**.**

Pergunte como poderíamos tornar isso mais fácil, tentando lembrá-los de que já foi feito algo parecido antes. Se não lembrarem, pergunte como eles fizeram para desenhar o triângulo sem precisar ficar chamando a função `poligono` o tempo todo e colocando o número 3 para o número de lados. Assim, lembre-os que podemos **criar funções**, isto é, podemos **ensinar** a tartaruga a executar um conjunto de comandos, de modo que ela possa repeti-los sempre que desejarmos. Escreva na lousa alguns exemplos de círculos, demonstrando o que muda e o que fica inalterado:

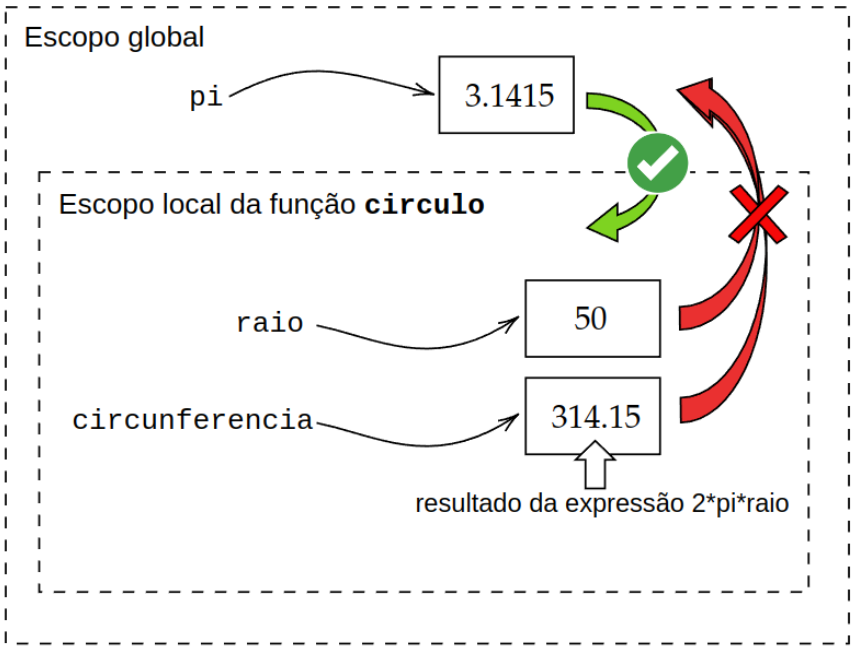
```
circunferencia = 2*pi*50
print("Valor do comprimento da circunferência:", circunferencia)
poligono(360, circunferencia/360)

circunferencia = 2*pi*100
print("Valor do comprimento da circunferência:", circunferencia)
poligono(360, circunferencia/360)

circunferencia = 2*pi*200
print("Valor do comprimento da circunferência:", circunferencia)
poligono(360, circunferencia/360)
```



Note que somente o valor referente ao raio muda em cada caso. Dê alguns minutos aos alunos para que tentem definir a função `circulo`. Parabenize os que conseguirem, estimule os que se esforçaram, e então mostre a solução:

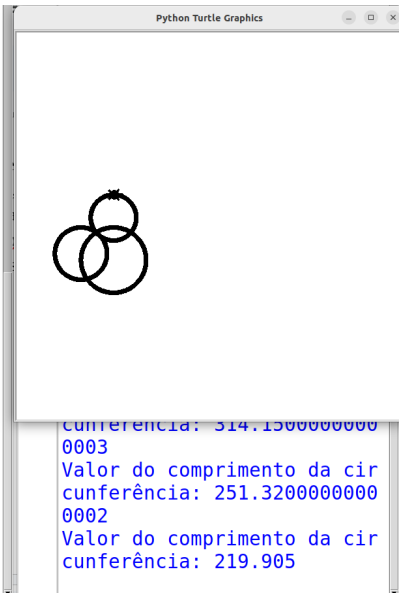
```
def circulo(raio):
    circunferencia = 2*pi*raio
    poligono(360, circunferencia/360)
```

Note que a variável `pi` não é passada por parâmetro, como o raio, pois podemos pegar ela do **escopo global** do nosso programa. Todas as variáveis que são definidas **fora** das definições de funções podem ser acessadas dentro de qualquer função, e são chamadas de **variáveis globais**. É o caso do `pi`, que é uma constante. Por outro lado, há variáveis que são definidas **dentro** da definição de uma função, que são chamadas de **variáveis locais**. Essas podem somente ser acessadas dentro da definição da própria função, mas nunca fora dela ou dentro de outras funções. A variável `circunferencia` é um exemplo de variável local, pois é definida somente dentro do escopo da função `circulo`. Além disso, todo parâmetro de uma função também é uma variável local dela. Portanto, o parâmetro `raio` também é uma variável local de `circulo`.



Mostre para os alunos que é possível desenhar círculos de diferentes tamanhos em lugares diferentes da tela. Basta levantar a caneta, mover a tartaruga para o lugar desejado, abaixar a caneta e então desenhar. Para isso, definimos ainda uma outra função responsável apenas por mover a tartaruga a uma posição sem mover, usando a sequência de comandos `penup(); goto(x, y); pendown()`. Esse é mais um exemplo de como podemos tornar nosso código mais modularizado e menos repetitivo.

```
pi = 3.1415
print("Valor do pi:", pi)

def vai_para_posicao_sem_riscar(x, y):
    penup()
    goto(x, y)
    pendown()

def circulo(raio):
    circunferencia = 2*pi*raio
    print("Valor do comprimento da circunferência:", circunferencia)
    poligono(360, circunferencia/360)

# Chamadas à função circulo
vai_para_posicao_sem_riscar(-150, 0)
circulo(50)
vai_para_posicao_sem_riscar(-200, 0)
circulo(40)
vai_para_posicao_sem_riscar(-150, 50)
circulo(35)



exitonclick()
```



Plotagem de gráficos de funções no plano cartesiano

Uma vez compreendidos inicialmente os conceitos de funções e plano cartesiano, podemos explorar a ideia de plotar gráficos de funções. Primeiramente, é interessante configurar a escala do plano cartesiano e adicionar um plano de fundo à tela da tartaruga que remete ao plano cartesiano.

Baixe a imagem da seguinte URL: https://helioh2.github.io/turtle/plano_cartesiano.png (clique com o botão direito do mouse e escolha a opção "Salvar imagem como…"). Copie-a para a mesma pasta em que você estiver salvando seus arquivos de código-fonte do Python.

Em seguida, crie um novo arquivo no IDLE, e então copie o seguinte código no início desse arquivo.

```
# Configurações da tela e da tartaruga
setup(800, 800)  # altera as dimensões da tela para 800x800
# Configura eixos do plano cartesiano para ir de -60 a 60
setworldcoordinates(-60, -60, 60, 60)
# Carrega a imagem de fundo na forma de plano cartesiano
bgpic("plano_cartesiano.png")
width(6)  # altera a espessura da caneta (linha desenhada)
shape("turtle")  # altera o formato da tartaruga
shapesize(1)  # altera o tamanho da tartaruga
color("red")
```

Agora, peça que os alunos criem uma função matemática simples: $f(x) = x$. Para isso, basta definir uma função chamada `f` no Python, com um único parâmetro `x`. Dentro da função, simplesmente retornaremos o valor de `x`.

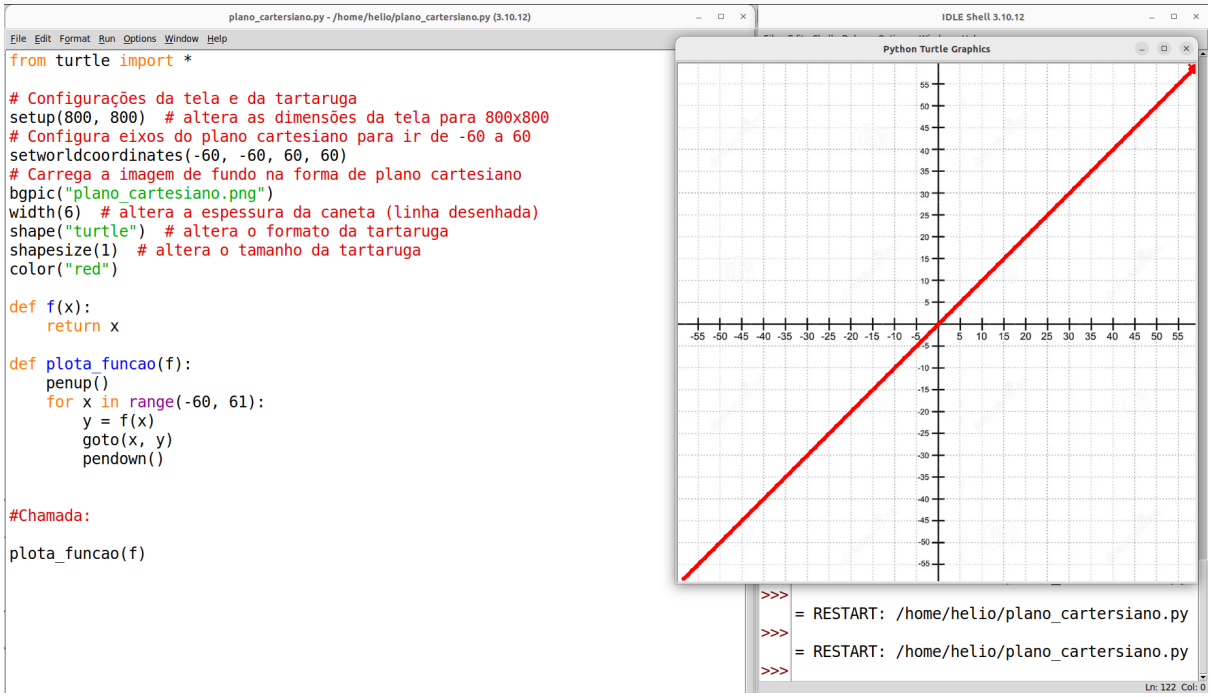
```
def f(x):
    return x
```

Agora, iremos escrever uma função que plota uma função qualquer que possua um único parâmetro `x`. Portanto, criaremos uma função chamada `plota_funcao` que recebe por parâmetro uma outra função. É isso mesmo! Vimos anteriormente que é possível passar valores numéricos para dentro de uma função por parâmetro. Mas em Python e em outras linguagens que possuem suporte à **programação funcional**, funções são objetos de alta ordem, podendo ser passados por parâmetro como se fossem valores. Não é necessário explicar essa informação mais teórica se não quiser aos alunos. Apenas explique que é possível passar funções para dentro de funções, e vamos compreender isso melhor assim que testarmos isso no código.

O funcionamento da função `plota_funcao` é o seguinte: usa-se um laço do tipo for para pegar todos os valores possíveis de `x` entre -60 e 60, que é a escala do eixo-x do plano cartesiano que configuramos. Para cada valor de `x`, então, calculamos o valor de `y` chamando a função `f`. Em seguida, usamos o comando `goto(x, y)` para fazer a tartaruga se mover até aquela posição.

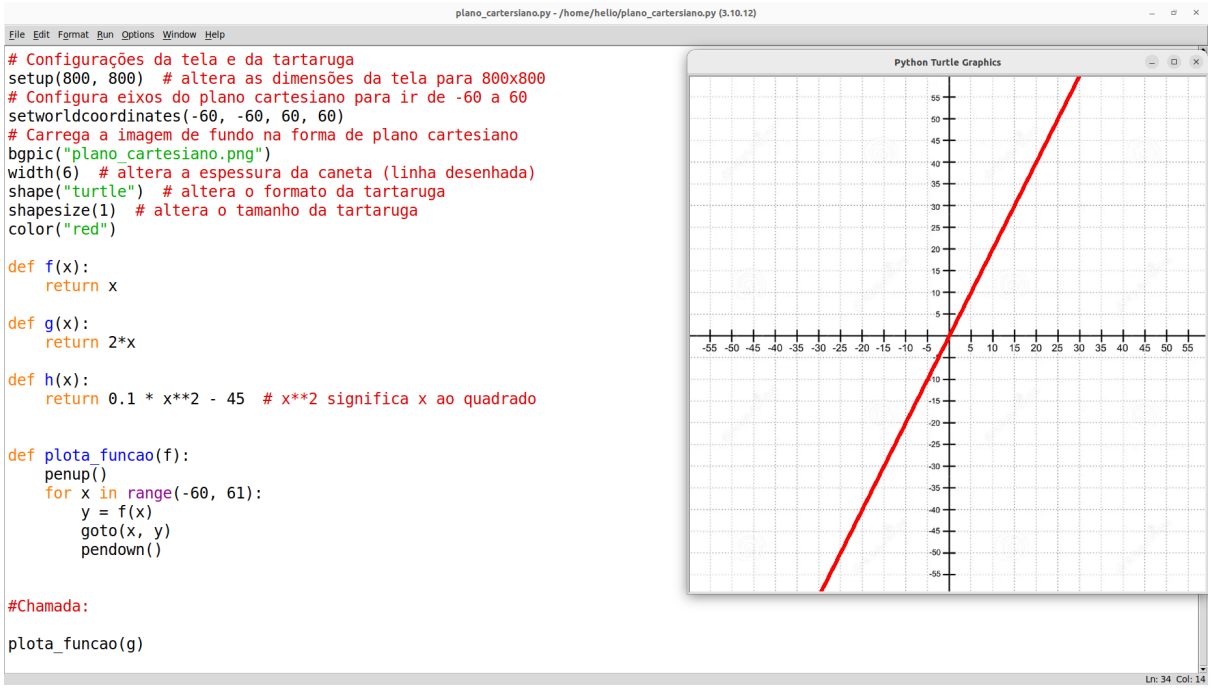
```
def plota_funcao(f):
    penup()
    for x in range(-60, 61):
```

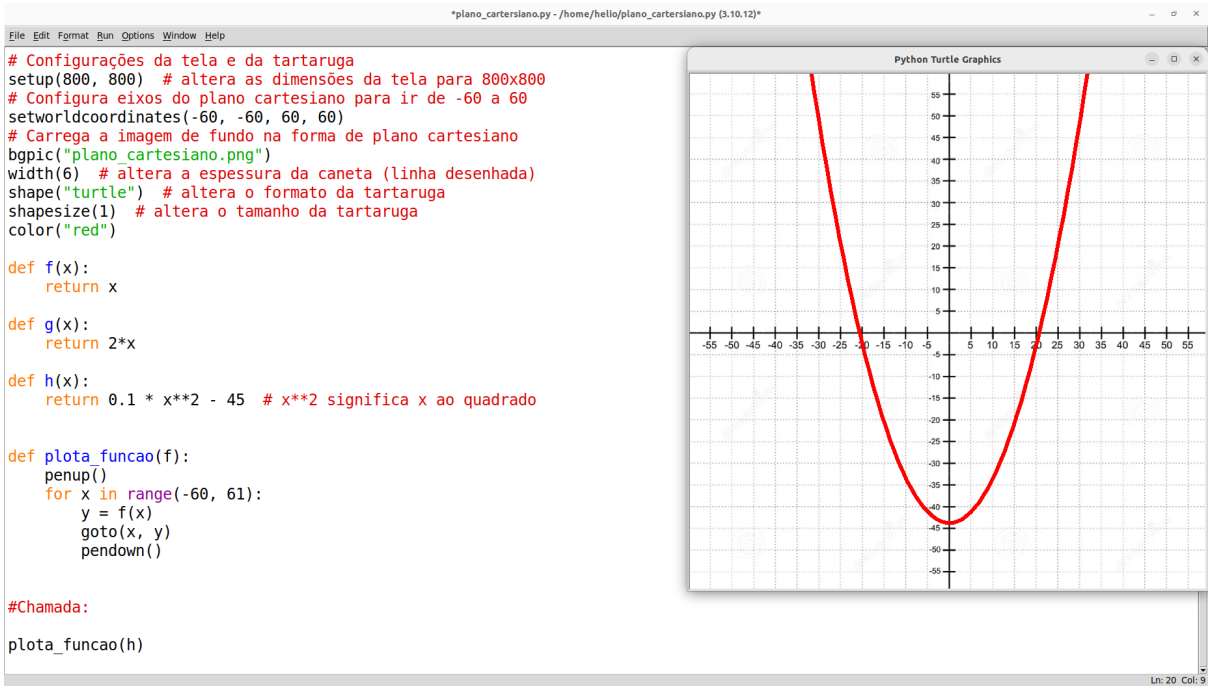
```
        y = f(x)
        goto(x, y)
        pendown()

#Chamada:
plota_funcao(f)
```



Agora podemos criar novas funções e passar elas para `plota_funcao`. Veja alguns exemplos:

```
def g(x):
    return 2*x

def h(x):
    return 0.1 * x**2 - 45  # x**2 significa x ao quadrado

# (...)

# Chamadas
plota_funcao(g)
plota_funcao(h)
```



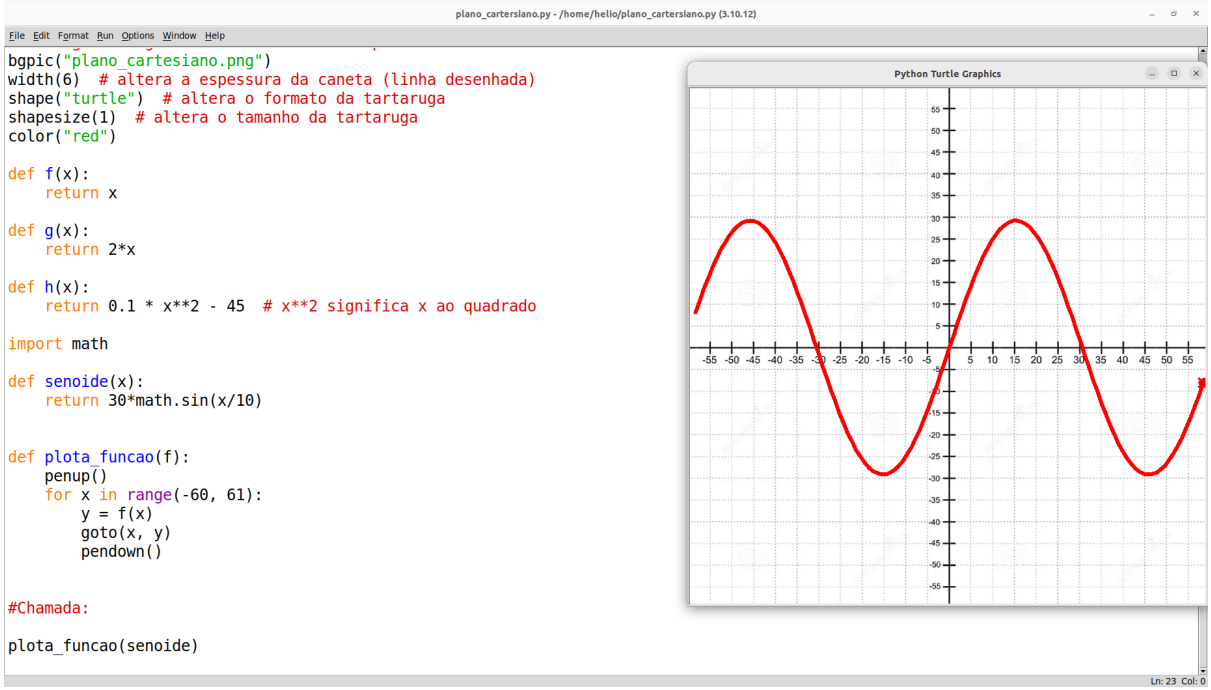


A função `g` é descrita matematicamente como: $g(x) = 2x$, sendo uma função de primeiro grau (assim como a `f`), plotada portanto como uma reta, mas com inclinação menor que a função `f`.

A função `h` é descrita matematicamente como $h(x) = 0{,}1x^2 - 45$, que é uma função de segundo grau, portanto plotada como uma parábola.

Podemos plotar também funções trigonométricas, como o *seno*. Basta importar o módulo `math` usando `import math`. Esse módulo contém diversas funções matemáticas prontas.

```
import math

def senoide(x):
    return 30*math.sin(x/10)

# (...)

# Chamadas
plota_funcao(senoide)
```



A função `senoide` acima seria descrita matematicamente como segue:

$$senoide(x) = 30 \times sen(x/10).$$

Os valores constantes multiplicados/divididos pelo *seno* e pelo x foram ajustados para se adequar à escala de nosso plano cartesiano. Fique à vontade para alterar esses valores para criar diferentes senóides.

Fique à vontade para testar também outras funções matemáticas. Algumas outras sugestões: $f(x) = 5$ (função constante); $f(x) = 0{,}001 \times x^3$ (função de terceiro grau).

Parabéns Professor(a)!! Se você seguiu esses planos de aula, temos esperança de que foi bastante proveitoso para os alunos. Conte-nos como foi!!

Indo além: estrelas, espirais e fractais

É possível criar uma grande variedade de desenhos com o Turtle, mas há três tipos de figuras que valem a pena mencionar caso você queira se aprofundar e, talvez, levar para seus alunos. Um desses tipos são as **estrelas**. A lógica do desenho de uma estrela não é muito diferente da lógica do desenho de polígonos regulares: repete-se uma certa quantidade de vezes um movimento para frente e um giro em determinado ângulo.

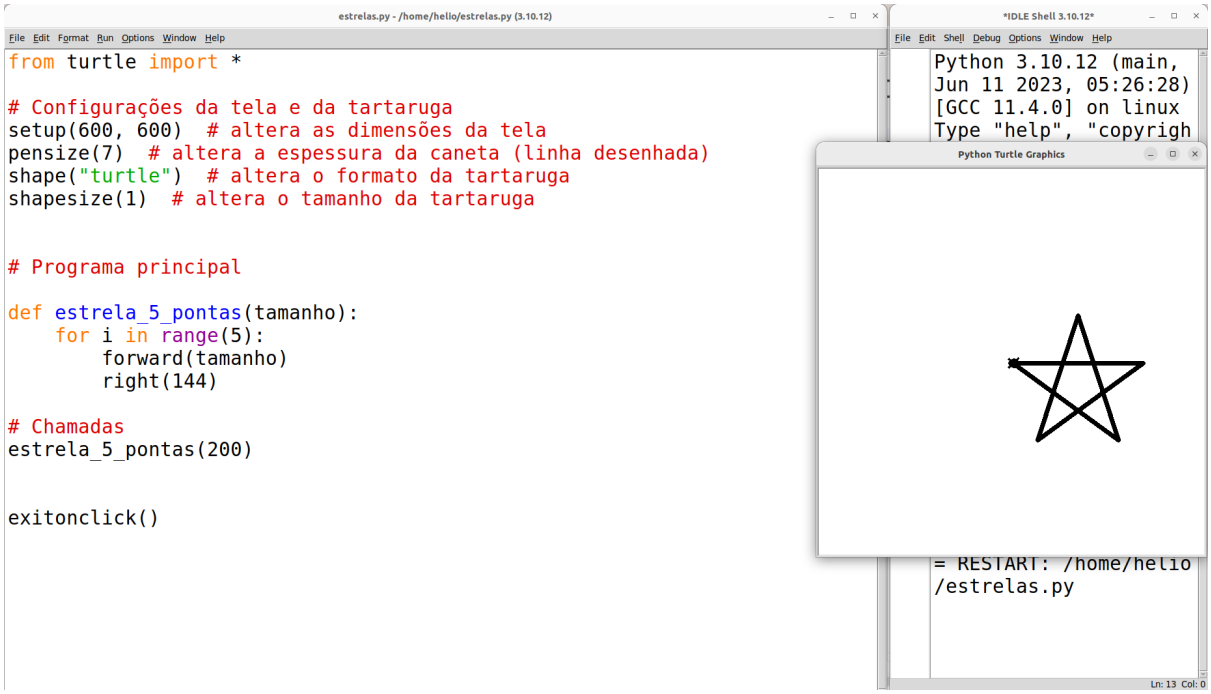
Outro tipo de figura interessante são as **espirais**, que também consistem em repetições de movimentos e giros, mas alterando-se o ângulo de giro e/ou a distância percorrida a cada repetição. É possível desenhar espirais baseadas nos mais diversos polígonos regulares, incluindo espirais circulares.

Por fim, o tipo mais intrigante de figuras são os **fractais**, que consistem em figuras geradas recursivamente, podendo inclusive ser geradas por tempo indeterminado. Muitos fractais simulam padrões presentes na natureza, como o crescimento de plantas e o formato de flocos de neve.

Estrelas

Há dois tipos de estrelas no que diz respeito à lógica de seu desenho: com quantidade ímpar de pontas e com quantidade par de pontas. Vejamos primeiramente como podemos fazer uma estrela com quantidade de pontas ímpar, começando com a estrela de 5 pontas.

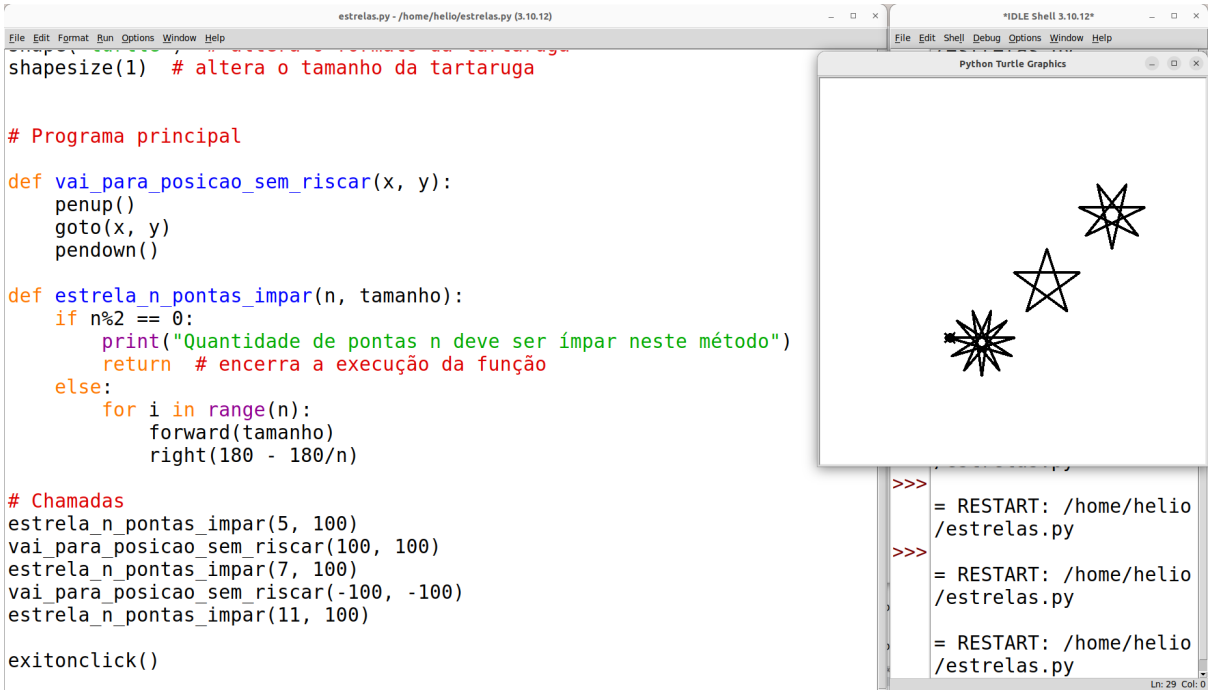
```
def estrela_5_pontas(tamanho):
    for i in range(5):
        forward(tamanho)
        right(144)

# Chamada
estrela_5_pontas(200)
```



Perceba que a construção lógica é idêntica ao desenho de um pentágono, mas com ângulo de 144°. Mas, de onde tiramos esse ângulo? Basta sabermos que o *ângulo interno* de cada ponta da estrela de 5 pontas tem 36º (que deriva do cálculo *180°/5*). Logo, seu *ângulo suplementar* é de *180° - 36° = 144°* (que é o ângulo que a tartaruga precisa virar para formar as pontas). A partir disso, podemos derivar uma fórmula geral para calcularmos o ângulo suplementar de uma estrela de *n* pontas: *ângulo = 180º - (180° / n)*. Assim podemos testar a fórmula para estrelas com diferentes quantidades ímpares de pontas, e chegamos à solução para qualquer número ímpar maior ou igual a 5.

```
def vai_para_posicao_sem_riscar(x, y):
    penup()
    goto(x, y)
    pendown()

def estrela_n_pontas_impar(n, tamanho):
    if n%2 == 0:
        print("Quantidade de pontas n deve ser ímpar neste
método")
    else:
        for i in range(n):
            forward(tamanho)
            right(180 - 180/n)

# Chamadas
estrela_n_pontas_impar(5, 100)
vai_para_posicao_sem_riscar(100, 100)
```

```
estrela_n_pontas_impar(7, 100)
vai_para_posicao_sem_riscar(-100, -100)
estrela_n_pontas_impar(11, 100)
```



Perceba que foi feito um comando condicional (**`if`***)* para verificar se `n` é par — i.e., se o resto da divisão de `n` por 2 é igual a zero. Se for par, imprime-se: *"Quantidade de pontas n deve ser ímpar neste método"*. Caso contrário (**`else`**), desenha-se a estrela. Aqui vemos, pela primeira vez neste material, um comando condicional, o famoso **`if... else...`**, que em português significa "se… senão…". A ideia é muito simples, como acabamos de ver: coloca-se na frente do **`if`** uma condição. Se ela for verdadeira, executa-se o código que está dentro do escopo do **`if`** ("se"). Senão (**`else`**), executa-se o que estiver dentro do escopo do **`else`**. Veremos mais exemplos mais adiante, então logo isso se tornará natural.

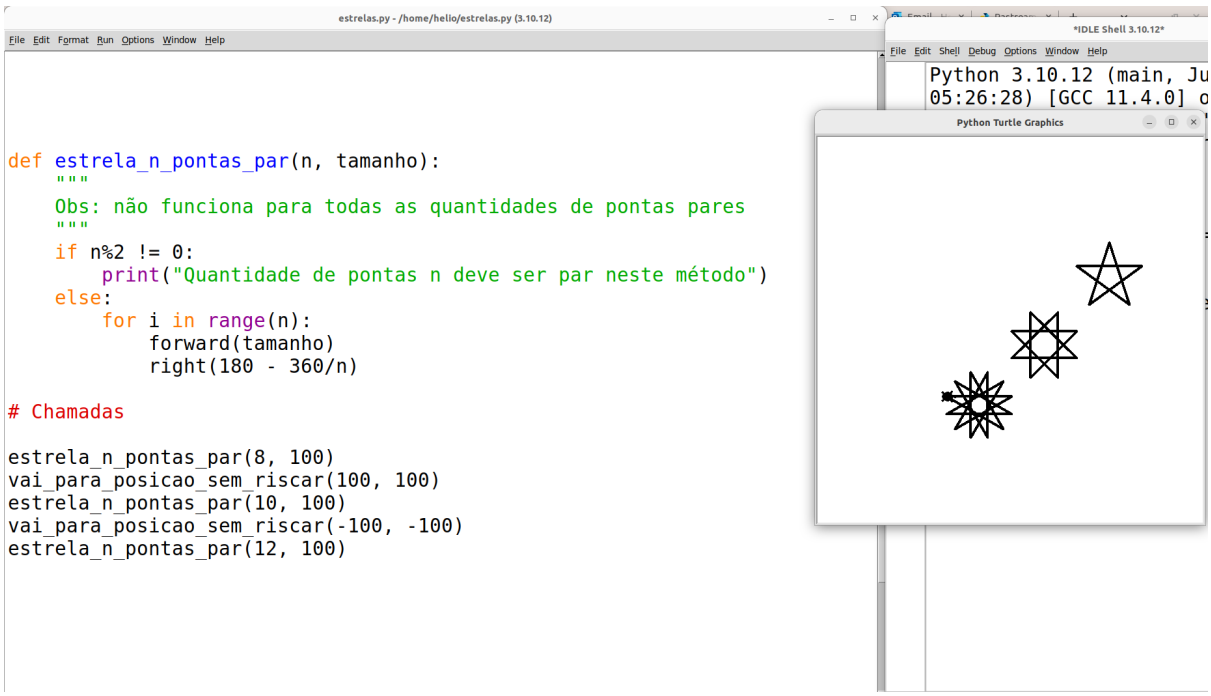
Note também como fazemos para usar a condição de igualdade: usando "==", e não "=". É assim em muitas linguagens de programação pois o "=" serve para fazer a atribuição de variáveis (como já vimos anteriormente), e não para comparação de valores.

Por fim, perceba que o laço **`for`** está dentro do escopo do **`else`**, e não do **`if`**. Portanto, ele só é executado quando `n` não é par, ou seja, quando é ímpar.

Quanto às estrelas com quantidade par de pontas, é um caso um pouco mais complexo. Tome, por exemplo, as estrelas de 6 e 8 pontas. Note que uma forma de desenhar a de 6 pontas é sobrepôr dois triângulos equiláteros rotacionados, e a de 8 pontas pode ser desenhada sobrepondo dois quadrados também rotacionados. No entanto, seguir essa lógica resultaria em estrelas muito "largas" à medida que a quantidade de pontas aumenta.

Alguns casos de estrelas com quantidade par de pontas podem ser desenhadas com uma variante da fórmula para estrelas com pontas ímpares. Basta calcular o ângulo usando `180 - 360/n`. Essa fórmula funciona para a estrela de 8 pontas e de 12 pontas, mas não funciona para de 6 e de 10, por exemplo. Veja abaixo:

```
def estrela_n_pontas_par(n, tamanho):
    """
    Obs: não funciona para todas as quantidades de pontas pares
    """
    if n%2 != 0:
        print("Quantidade de pontas n deve ser par neste método")
    else:
        for i in range(n):
            forward(tamanho)
            right(180 - 360/n)
```

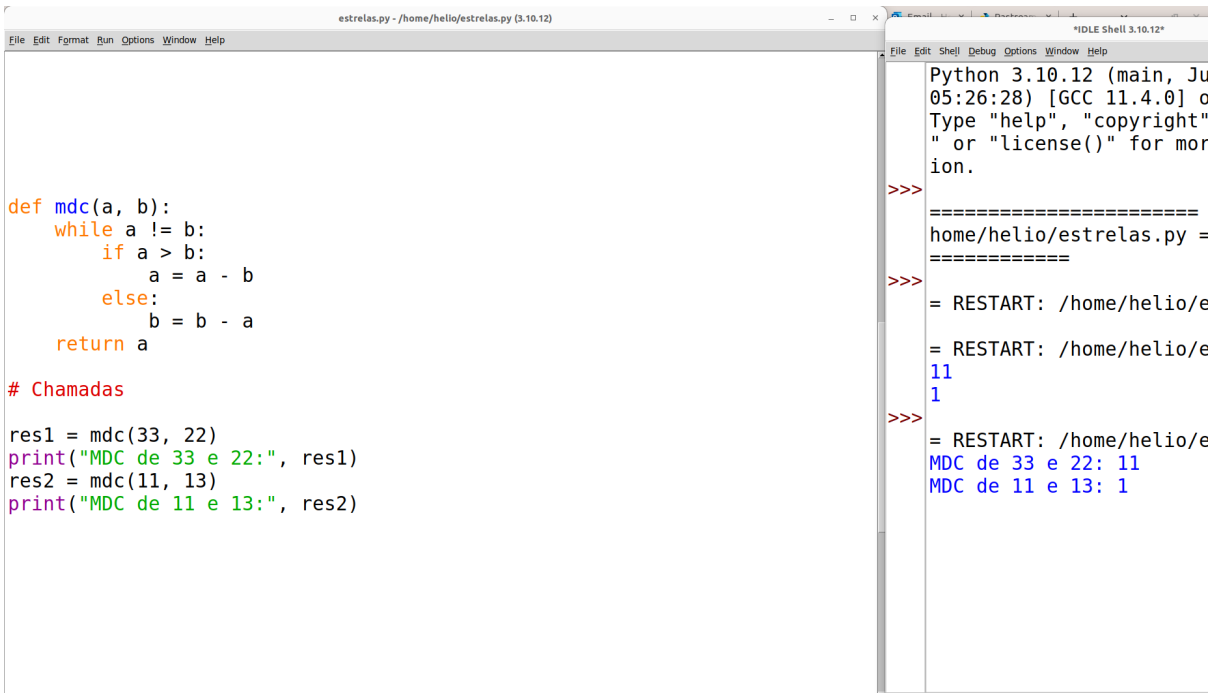


Note que quando *n=10*, esse método desenha uma estrela de 5 pontas.

Uma solução mais sofisticada, capaz de desenhar a maioria das estrelas, tanto ímpares quanto pares, é utilizar o cálculo do Máximo Divisor Comum (MDC) para procurar um número coprimo da quantidade de pontas. Dois números são coprimos entre si se não existe um divisor comum entre eles além do próprio número 1. Uma vez encontrado um coprimo (`p`) da quantidade de pontas (`n`) desejada, basta calcular o ângulo como `(360/n) * p`. Portanto, precisaremos primeiramente implementar (codificar) um algoritmo que permita fazer o cálculo do MDC: o algoritmo de Euclides, um dos algoritmos mais antigos, conhecido desde por volta de 300 a.C. O algoritmo de Euclides é simples: ele toma como entrada dois números inteiros

(`a` e `b`), e, a cada iteração (repetição), subtrai o maior pelo menor e substitui o maior pelo resultado da subtração. Repete-se esse processo até que `a==b` (o valor de `a` seja igual o valor de `b`). Abaixo segue a implementação do algoritmo em Python:

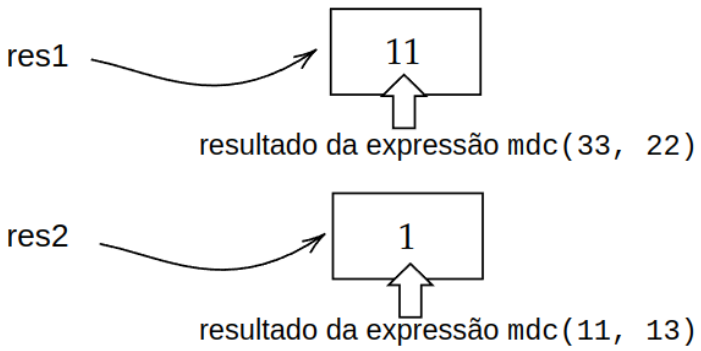
```
def mdc(a, b):
    while a != b:
        if a > b:
            a = a - b
        else:
            b = b - a
    return a

# Chamadas
res1 = mdc(33, 22)
print("MDC de 33 e 22:", res1)
res2 = mdc(11, 13)
print("MDC de 11 e 13:", res2)
```



Temos algumas coisas novas aqui também. Primeiro, temos o comando **`while`** ("enquanto"), que, assim como o **`for`**, cria um laço de repetição. No entanto, aqui definimos uma condição (assim como no **`if`**) que deverá ser verdadeira para que o laço seja executado a cada ciclo. No caso em tela, a lógica é a seguinte: **Enquanto** (**`while`**) `a` for diferente ("`!=`") de `b`, faça o que está dentro do escopo do **`while`**. O que é feito dentro do escopo do **`while`**? Um novo condicional da seguinte forma: Se (**`if`**) `a` for maior que `b`, então o valor de `a` é atualizado com

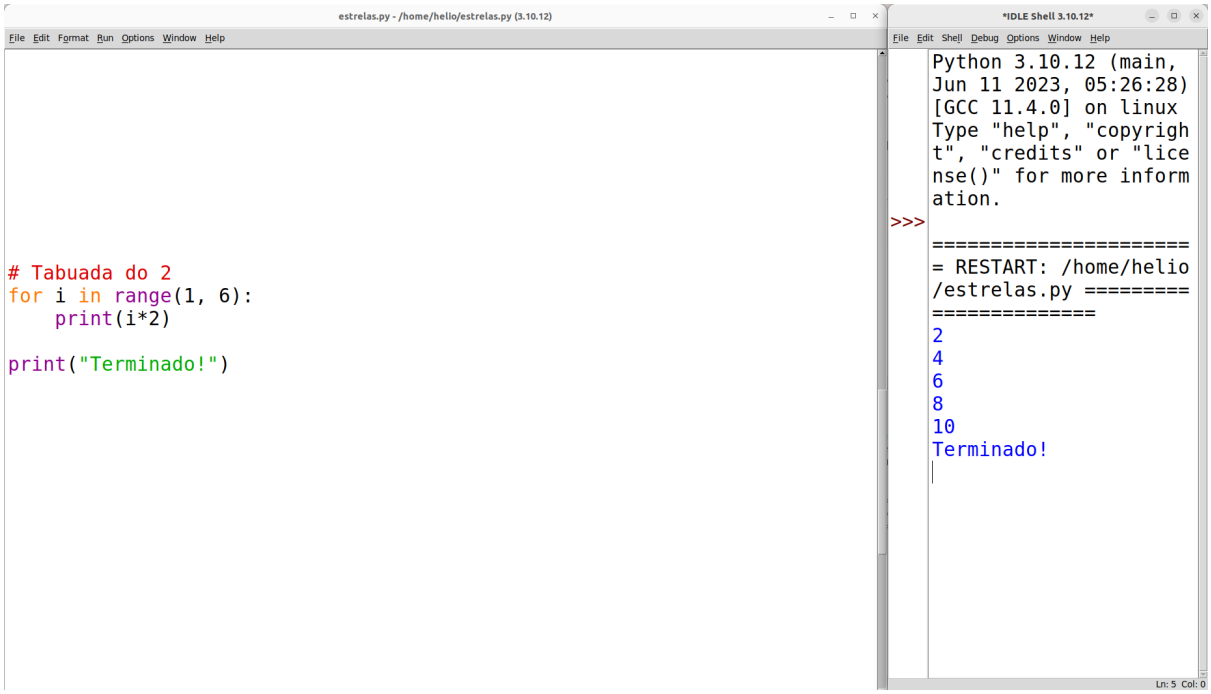
o valor resultante do cálculo `a - b`. Senão (**`else`**) — ou seja, se `a` for menor que `b` — então o valor de `b` é atualizado com o valor resultante do cálculo `b - a`.

Em algum momento, à medida que executamos esse processo, o valor de `a` será igual a `b`. Quando isso acontecer, a condição do **`while`** se tornará falsa — pois "`a` ser igual a `b`" é o oposto de "`a` ser diferente de `b`". Nesse caso, o algoritmo não entrará novamente no escopo do **`while`**, pulando a linha de execução para depois do **`while`**, isto é, para o comando **`return`** `a` (perceba que este comando está fora do escopo do **`while`**). O comando **`return`** faz com que a função retorne, ao final, um valor para a parte do código que a chamou. Quando a função retorna um valor, é possível utilizar esse valor de retorno em outros lugares do código, como fizemos nas chamadas de função no código acima. Perceba que criamos duas variáveis, `res1` e `res2`. A variável `res1` recebe o valor resultante (retornado) pela chamada à função `mdc(33, 22)`, e a variável `res2` recebe o valor resultante (retornado) pela chamada à função `mdc(11, 13)`. Depois, imprimimos, usando o **`print`**, esses valores, que já estarão guardados dentro das respectivas variáveis. Veja na figura abaixo uma ilustração das variáveis `res1` e `res2` recebendo o valor resultante das chamadas.



Uma vez definido o algoritmo que calcula o MDC, podemos definir a função `estrela` da seguinte forma: testa-se cada número entre 2 e *n/2* (quociente da divisão de *n* por 2, que em Python é dado por `n//2`), de modo a verificar se algum desses número é coprimo de *n*. Para testar todos os números entre 2 e *n*/2, podemos usar o **`while`**, mas também podemos usar o **`for`**, porém de uma forma um pouco mais detalhada. Até aqui, neste material, utilizamos o **`for`** simplesmente como uma forma de repetir os mesmos comandos por uma determinada quantidade de vezes. No entanto, o **`for`** possui um uso mais amplo, o qual iremos "revelar" agora. Veja o exemplo abaixo:

```
# Tabuada do 2
for i in range(1, 6):
    print(i*2)

print("Terminado!")
```



Perceba que passamos para o `range` dois números. O **primeiro** significa que a variável `i` (que chamamos de **variável de laço**) será criada e definida com o valor inicial 1. Então, dentro do escopo do `for`, usamos o valor dessa variável `i` e o multiplicamos por 2, imprimindo o valor resultante. Em seguida, na próxima iteração (i.e., na próxima repetição do laço), a variável `i` será atualizada para `i+1`, ou seja, será atualizada (incrementada) para o valor 2. Na iteração seguinte, será incrementada para 3, depois para 4, e depois para 5. A cada vez, é executado o que está dentro do `for`. Quando a variável `i` tiver que ser incrementada para 6, então chegamos ao limite — que definimos como o **segundo** parâmetro do `range`. Nesse caso, não se entrará mais no escopo do `for`, fazendo com que a execução pule para a próxima linha de código após (e fora) do `for`. Portanto, a forma geral do `for` é a seguinte:

```
for <nome da variável de laço> in range(<valor inicial>, <limite>, <opcional: passo>):

    ## Faz algo com a variável de laço
```

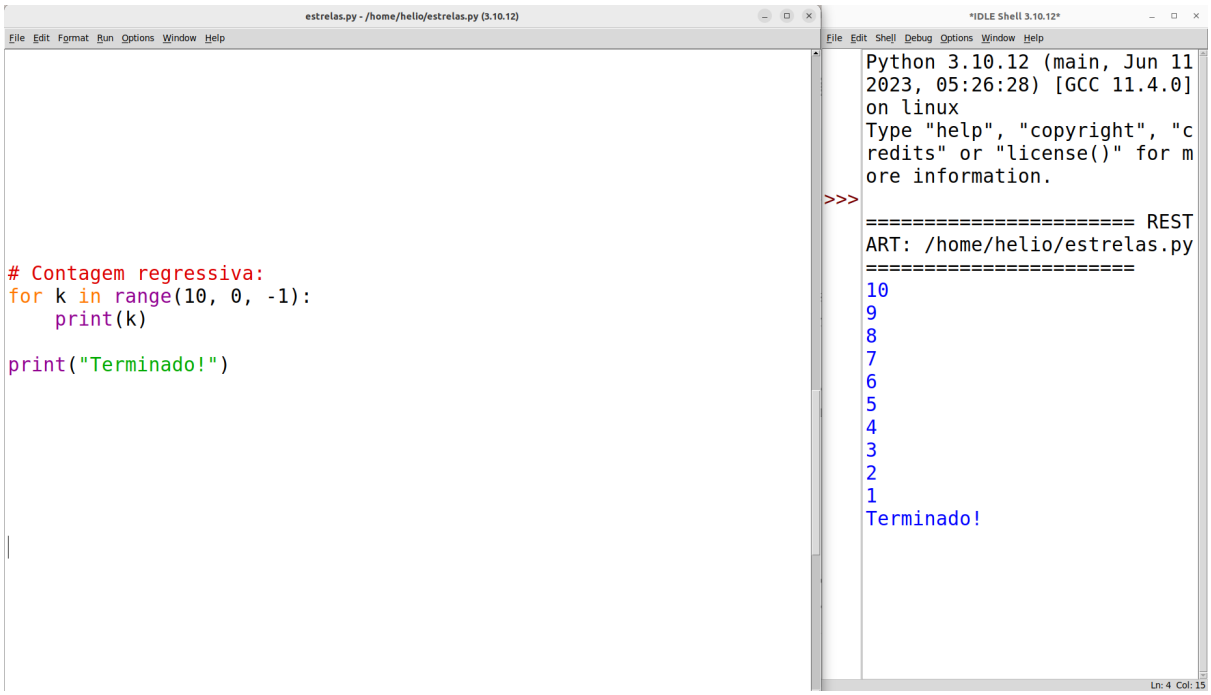
Note que temos a possibilidade também de incluir um **terceiro** parâmetro ao `range`: o passo. Quando não colocamos um terceiro parâmetro, a variável de laço é incrementada com *passo=1* — ou seja, incrementa de 1 em 1 — como no exemplo da tabuada: note que inicialmente *i=1*, depois *i=2*, depois *i=3*, etc. No entanto, podemos definir um passo, que, por exemplo, incrementa de 2 em 2. Como abaixo:

```
for k in range(2, 12, 2):
    print(k)
```

Note que usamos um nome diferente para a variável de laço (`k`). Assim como as demais variáveis (parâmetros de função, variáveis locais e variáveis globais), podemos dar o nome que quisermos à variável de laço. Execute o código acima no Python e note que ele dará o mesmo resultado do código anterior, pois agora `k` será inicializado com valor 2, que então será incrementado de 2 em 2, pois definimos o terceiro parâmetro, que é o passo, com 2. Assim, inicialmente *k=2*, depois *k=4*, depois *k=6*, depois *k=8*, depois *k=10*. Quando chegar em 12, o `for` termina sua execução, pois o limite (segundo parâmetro do `range`) é 12.

Podemos também definir o passo de modo que, em vez de incrementar o valor da variável de laço, o decrementamos, isto é, diminuímos seu valor. Para isso, basta colocar um número negativo para o passo, como vemos a seguir:

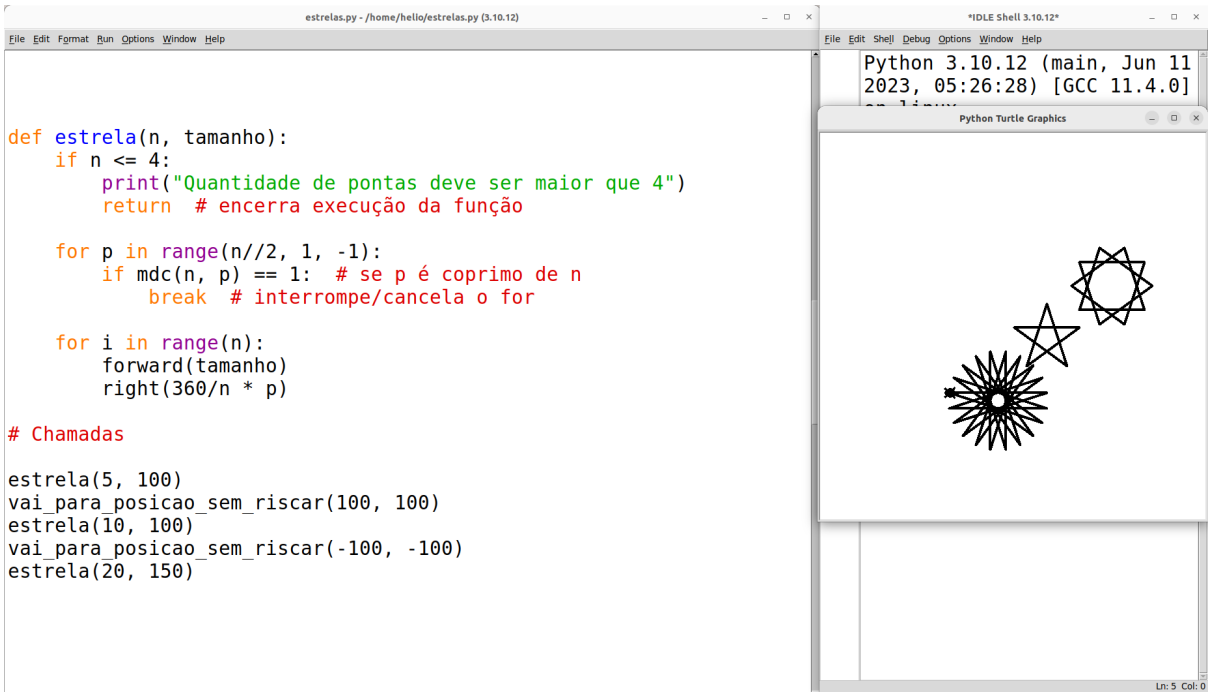
```
# Contagem regressiva:
for k in range(10, 0, -1):
      print(k)
```



Note que, nesse caso, definimos o limite (segundo parâmetro) como 0. Logo, o valor de `k` só vai até o 1. Se quisermos incluir o 0, temos que trocar o segundo parâmetro para -1. Faça o teste você mesmo(a).

Voltando a nossa função `estrela`, a ideia é inicializar uma variável de laço `p` com o valor `n//2` (quociente da divisão de `n` por 2) e, a cada iteração do laço, decrementar este valor, até chegar ao valor 2. Assim, para cada valor de `p`, calcula-se o MDC de `n` e `p` e verifica-se se o MDC é igual a 1. Se for, significa que `n` e `p` são coprimos entre si, e assim podemos usá-los na fórmula para calcular o ângulo de giro da tartaruga. Veja abaixo como fica:

```
def estrela(n, tamanho):
    if n <= 4:
        print("Quantidade de pontas deve ser maior que 4")
        return  # encerra execução da função

    for p in range(n//2, 1, -1):
    # Obs: n//2 pega o quociente da divisão de n por 2
        if mdc(n, p) == 1:  # se p é coprimo de n
            break  # interrompe/cancela o for

    for i in range(n):
        forward(tamanho)
        right(360/n * p)

# Chamadas

estrela(5, 100)
vai_para_posicao_sem_riscar(100, 100)
estrela(10, 100)
vai_para_posicao_sem_riscar(-100, -100)
estrela(20, 150)
```

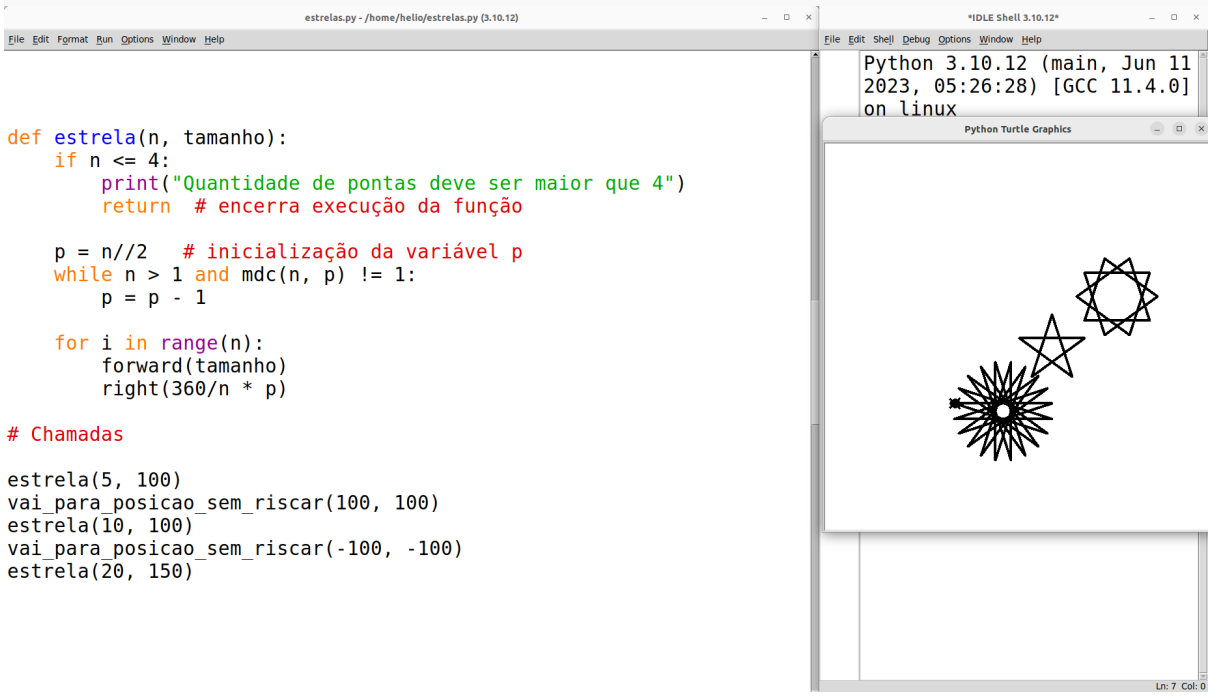


Note que, dentro do `for p in range...`, verificamos se `n` e `p` são coprimos, isto é, se o MDC de `n` e `p` é igual a 1. Se for o caso, usamos o comando **`break`** ("interrompe") para interromper, ou cancelar, a execução do **`for`**. Quando, dentro de um laço, executa-se o **`break`**,

o laço deixa de repetir, passando a linha de execução para a próxima instrução após (e fora) do laço. A ideia é simples: se encontramos um coprimo `p`, não é necessário continuar procurando, por isso interrompe-se a busca.

Uma alternativa, igualmente correta, é usar **`while`** em vez de **`for`** para fazer a busca do coprimo. Veja como fica:

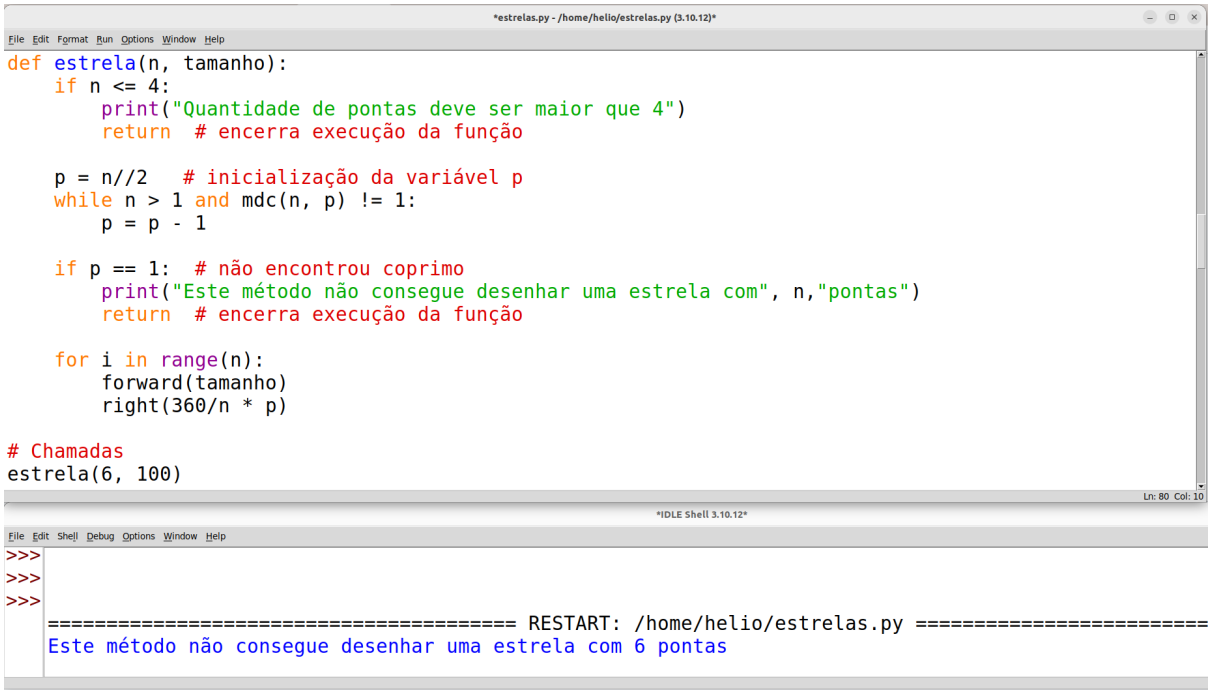
```
def estrela(n, tamanho):
    if n <= 4:
        print("Quantidade de pontas deve ser maior que 4")
        return  # encerra execução da função

    p = n//2   # inicialização da variável p
    while n > 1 and mdc(n, p) != 1:
        p = p - 1

    for i in range(n):
        forward(tamanho)
        right(360/n * p)

# Chamadas
estrela(5, 100)
vai_para_posicao_sem_riscar(100, 100)
estrela(10, 100)
vai_para_posicao_sem_riscar(-100, -100)
estrela(20, 150)
```



Nesse caso, temos que o `while` só continua a executar se atender a duas condições: `n` deve ser maior que 1 (i.e., maior ou igual a 2) **e** ("`and`") o MDC de `p` e `n` deve ser diferente de 1. Se qualquer uma dessas condições não for satisfeita, deve-se "sair" do `while`. Dentro do `while`, `p` é decrementado "manualmente", ou seja, nós mesmos colocamos uma linha de código que realiza a atualização do valor de `p`. Se, em algum momento, o MDC de `p` e `n` resultar em 1, o `while` é interrompido, pois infringirá a segunda condição. Pode acontecer também de um coprimo não ser encontrado. Isso acontece quando `p` chega ao valor de 1, infringindo a primeira condição do `while`; ou seja, todos os números entre `2` e `n//2` foram testados e nenhum deles é coprimo. Esses casos ocorrem nas exceções ao nosso algoritmo, isto é, nos casos de estrelas que nosso algoritmo ainda não consegue desenhar. Um caso conhecido é $n=6$.

Dada a existência desses casos de exceção, é interessante que coloquemos um comando condicional (`if`) logo após a busca pelo coprimo. Basta verificar se, ao final da busca, `p==1`. Se for o caso, imprimimos uma mensagem de erro e damos um `return`. Quando retornamos, mesmo sem retornar nenhum valor, a execução da função é terminada. Desse modo, a função nem mesmo tentará desenhar a estrela. Veja abaixo como fica então essa versão final:

```
def estrela(n, tamanho):
    if n <= 4:
        print("Quantidade de pontas deve ser maior que 4")
        return  # encerra execução da função

    p = n//2   # inicialização da variável p
    while n > 1 and mdc(n, p) != 1:
        p = p - 1

    if p == 1:  # não encontrou coprimo
      print("Este método não consegue desenhar uma estrela com",
n,"pontas")
        return  # encerra execução da função

    for i in range(n):
        forward(tamanho)
        right(360/n * p)
```

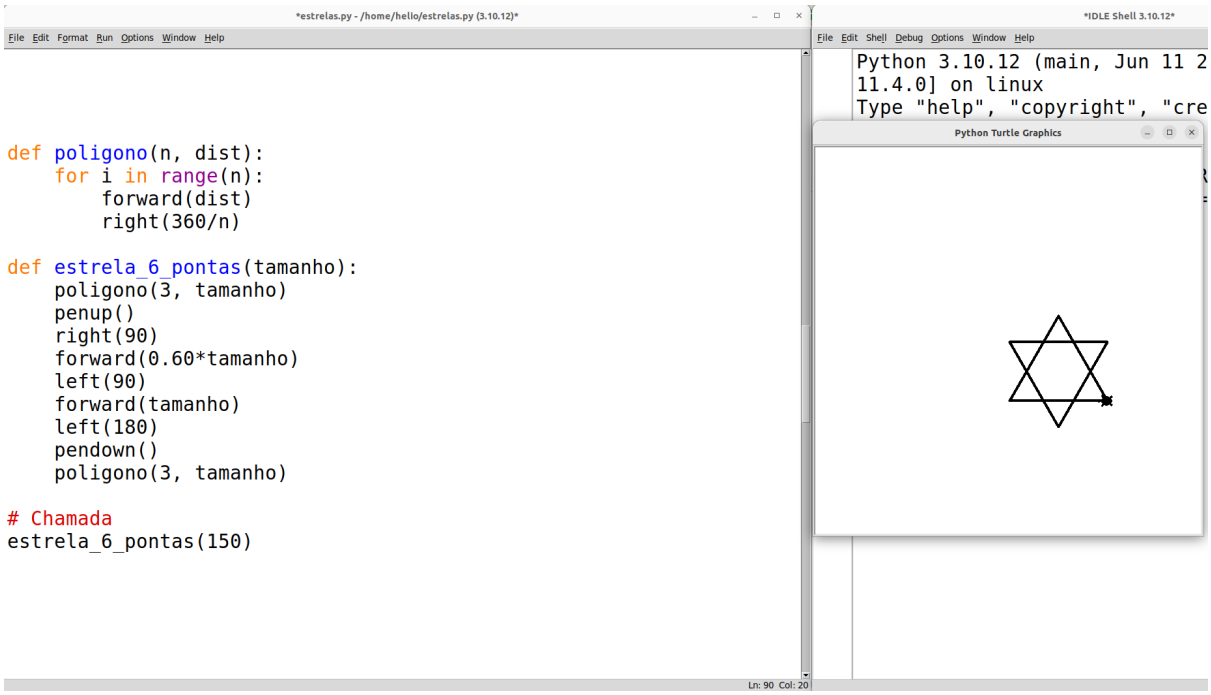


Note também que o uso do `return` dentro do `if` é uma alternativa ao uso do `else` após o `if`, pois, se a execução entrar no `if`, terminará. Senão (*"else"*), ela continua a execução após o `if`. Muitos programadores acabam preferindo esse tipo de uso do `return` no lugar de se utilizar o `else` pois deixa o código com menos níveis de aninhamento (perceba que evitamos de ter que colocar um nível de indentação a mais no `for`). No entanto, há situações em que pode ser mais interessante o uso do `else`, ficando a cargo do programador decidir o que é mais adequado.

Como já vimos, temos uma exceção conhecida para a qual nosso algoritmo não funciona: quando $n=6$. Esse é um exemplo de caso específico que temos que tratar separadamente. Uma forma de se desenhar uma estrela de 6 pontas é desenhando dois triângulos equiláteros sobrepostos (por exemplo, usando a função `poligono` que definimos anteriormente) e virados em ângulos diferentes. Um exemplo de como isso pode ser feito:

```
def poligono(n, dist):
    for i in range(n):
        forward(dist)
        right(360/n)

def estrela_6_pontas(tamanho):
    poligono(3, tamanho)
    penup()
    right(90)
    forward(0.60*tamanho)
    left(90)
    forward(tamanho)
    left(180)
```

```
    pendown()
    poligono(3, tamanho)

# Chamada
estrela_6_pontas(150)
```



Note que, dentro da função `estrela_6_pontas`, chamamos duas vezes a função `poligono` para desenhar dois triângulos. A dificuldade maior é posicionar a tartaruga de modo que os dois triângulos fiquem sobrepostos de modo a formar a estrela. A ideia é que a tartaruga, antes de desenhar o segundo triângulo, deve se mover por uma determinada proporção ao tamanho das linhas da estrela. De modo experimental — i.e. testando várias proporções — cheguei à ideia de fazer a tartaruga se mover por 60% do tamanho das linhas.

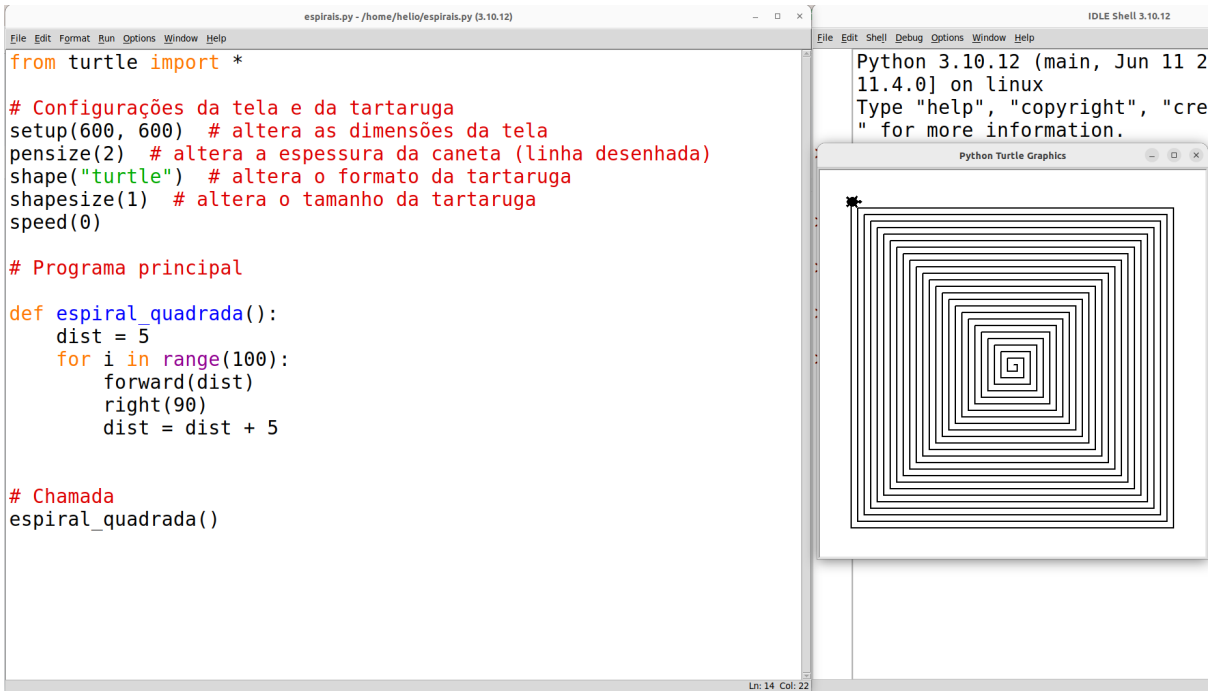
Espirais

Para construir espirais, é necessário fazer um laço de repetição similar ao que já fizemos para polígonos. No entanto, será necessária a criação de uma **variável de laço** que altere, a cada repetição, a distância a percorrer e, em alguns casos, o ângulo de giro da tartaruga.

Veja a seguir o exemplo de uma espiral em formato quadriculado:

```
def espiral_quadrada():
    dist = 5
    for i in range(100):
        forward(dist)
        right(90)
```

```
        dist = dist + 5

# Chamada
espiral_quadrada()
```

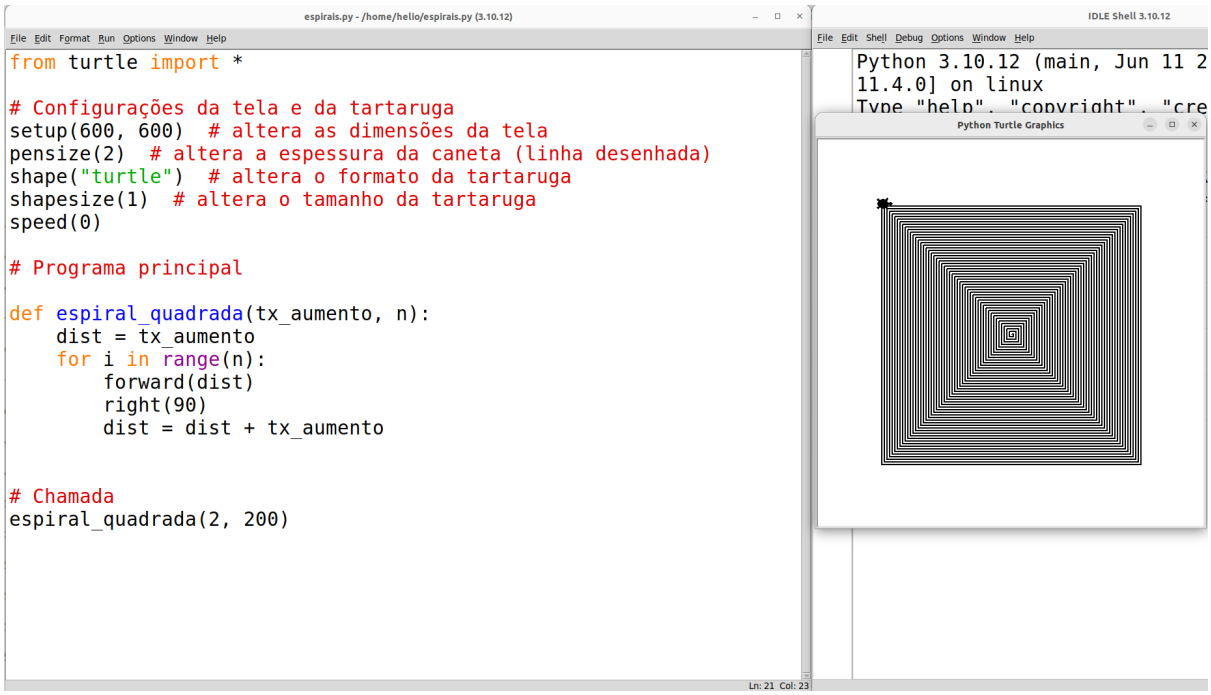


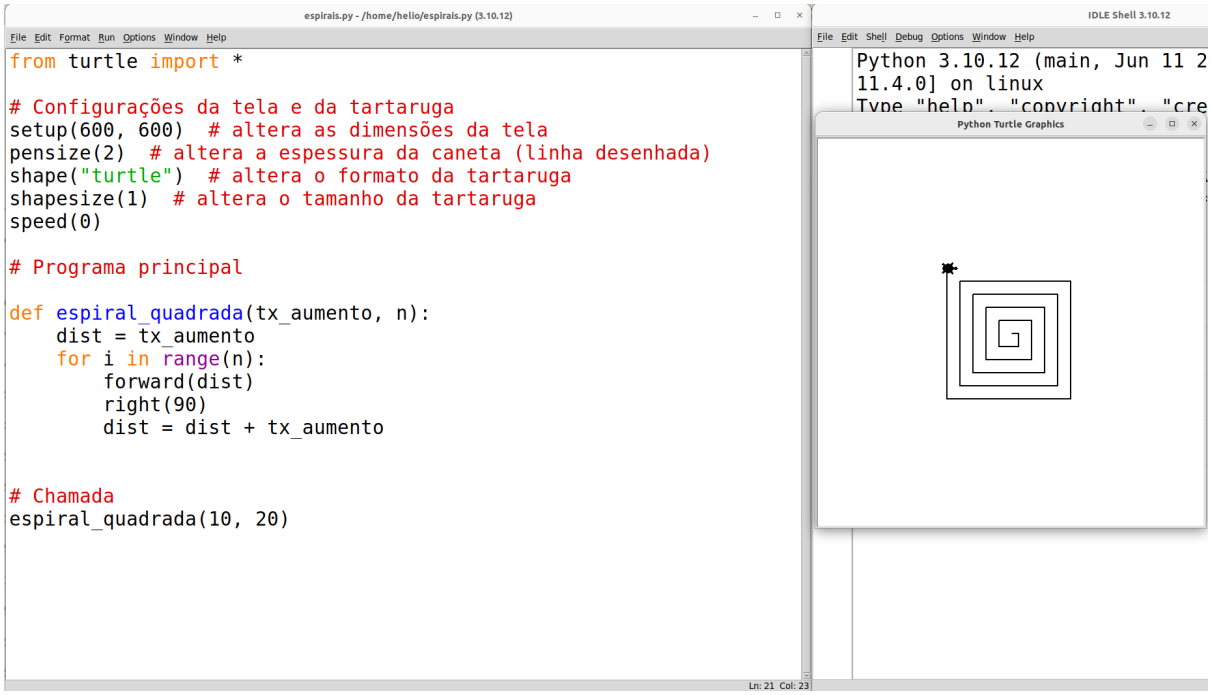
Note que, no início da função, definiu-se uma variável chamada `dist`. Essa variável é inicializada com o valor 5, e, a cada iteração do **`for`**, utiliza-se essa variável no comando `forward` e, ao final, incrementa-se em 5 o valor da variável. Isto é, o valor da variável sempre é atualizado a cada iteração somando-se 5 ao seu valor atual. Dessa forma, a cada iteração a tartaruga avança uma distância cada vez maior. Por exemplo, na primeira iteração, ela avançará 5. Na segunda, ela avançará 10. Na terceira, ela avançará 15. E assim por diante, até que a quantidade máxima de iterações definida no **`for`** (100, no caso acima), seja atingida.

É possível também parametrizar essa função para que possamos escolher a "taxa de aumento" e a quantidade de iterações. Essa "taxa de aumento" (variável `tx_aumento`) substituirá o valor 5 no código acima, de modo que, se definirmos um valor menor, a espiral será mais "densa", com espaços menores, e se definirmos um valor maior, será mais "esparsa", com espaços maiores. Veja abaixo essa nova versão e diferentes espirais geradas:

```
def espiral_quadrada(tx_aumento, n):
    dist = tx_aumento
    for i in range(n):
        forward(dist)
        right(90)
        dist = dist + tx_aumento
```

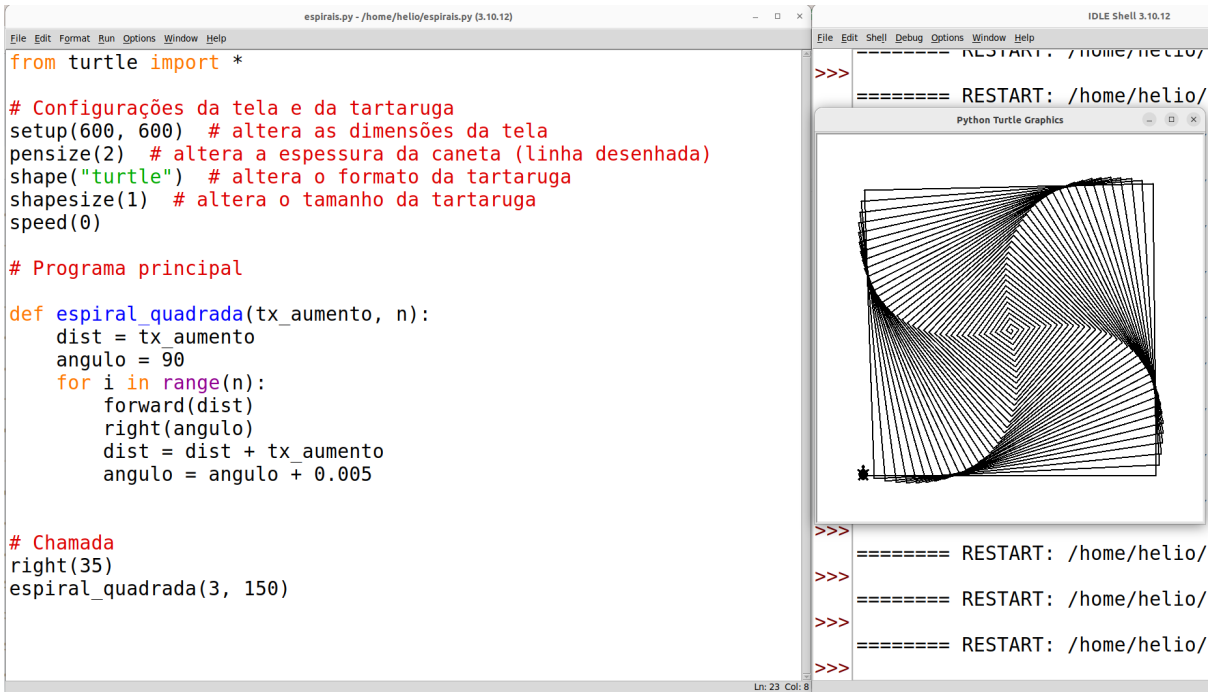
```
# Chamada
espiral_quadrada(2, 100)
espiral_quadrada(10, 20)
```





É possível também definir uma variante dessa espiral que faz um efeito de "torção" na espiral. Basta definir uma variável `angulo`, que inicia com o valor 90 e é incrementada a cada repetição por um pequeno valor (por exemplo, 0.005). Veja:

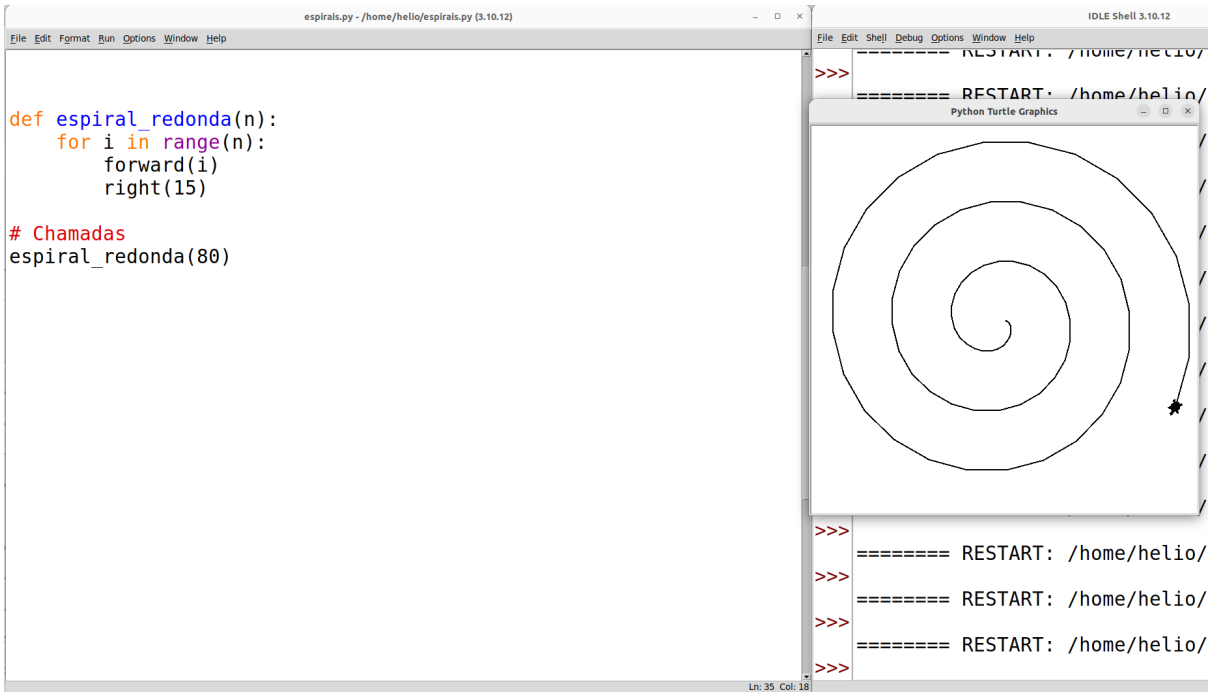
```
def espiral_quadrada(tx_aumento, n):
    dist = tx_aumento
    angulo = 90
    for i in range(n):
        forward(dist)
        right(angulo)
        dist = dist + tx_aumento
        angulo = angulo + 0.005

# Chamada
right(35)
espiral_quadrada(3, 150)
```



É possível também desenhar espirais circulares. A lógica para isso é muito similar à espiral quadriculada. No entanto, fazemos com que a distância percorrida e o ângulo sejam pequenos, e alteramos aos poucos a distância e/ou o ângulo. Nesse caso, podemos utilizar a própria variável de laço `i` do **`for`** para calcular uma taxa de crescimento para a distância e o ângulo. Veja abaixo uma primeira tentativa, utilizando `i` para a distância a se percorrer e 15º fixos de ângulo. Note que, dessa forma, inicialmente a tartaruga percorre a distância de 0 (pois i=0), e

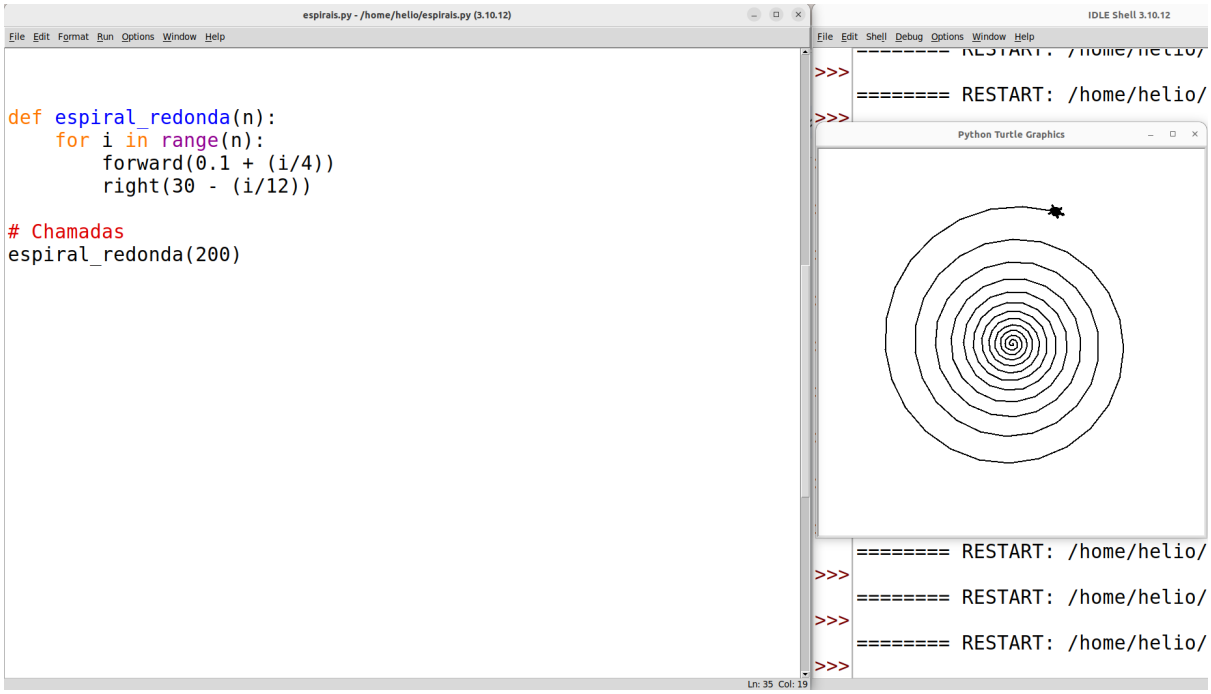
vira 15º. Em seguida, percorre a distância de 1 (pois *i*=1), e vira 15º. Em seguida, distância de 2 (pois *i=2*), e vira 15º, e assim por diante.

```
def espiral_redonda(n):
    for i in range(n):
        forward(i)
        right(15)

# Chamada
espiral_redonda(80)
```



A partir dessa ideia, podemos testar diferentes cálculos de distância e ângulo envolvendo `i`. Veja um exemplo, calculando-se a distância usando a fórmula `0.1 + (i/4)` para a distância e `30 - (i/12)` para o ângulo.

```
def espiral_redonda(n):
    for i in range(n):
        forward(0.1 + (i/4))
        right(30 - (i/12))

# Chamada
espiral_redonda(200)
```

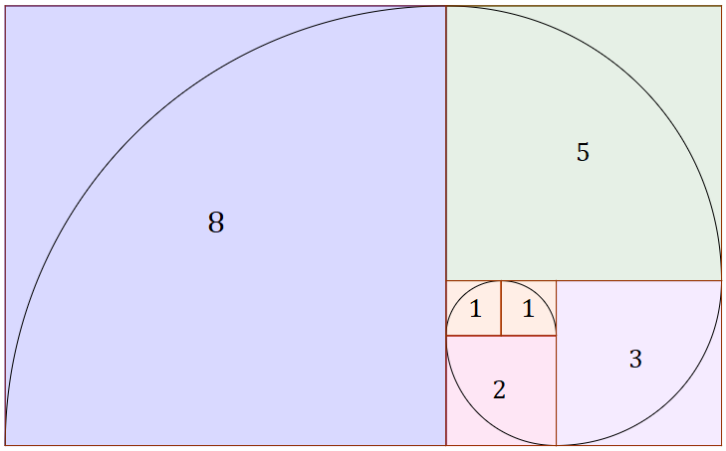


Fique à vontade para testar diferentes fórmulas e veja como ficam as diferentes espirais.

Um tipo muito interessante de espiral que é possível desenhar com o Turtle é a **espiral de Fibonacci**, que é baseada na **sequência de Fibonacci**. Essa conhecida sequência (ou série) consiste em gerar o próximo valor da sequência somando-se os dois valores anteriores. Por exemplo, os 10 primeiros números da sequência de Fibonacci são: 1, 1, 2, 3, 5, 8, 13, 21, 34, 55[9]. Note que a sequência sempre se inicia com "1, 1" e, então, a partir do terceiro elemento, soma-se os dois anteriores. Assim, o terceiro elemento é 2 = 1 + 1, o quarto é 3 = 2 + 1, o quinto é 5 = 3 + 2, e assim por diante. A partir da sequência de Fibonacci, é possível, definindo-se e calculando-se sua função de recorrência, calcular a assim chamada **proporção áurea**. Ou seja, à medida que a sequência de Fibonacci avança em valores cada vez maiores, a proporção entre o último e o penúltimo elemento calculados será de aproximadamente 1,61803 (trata-se de um número irracional, como o *pi*). Note, por exemplo, que se dividirmos 34 (o 9º valor da sequência) por 21 (o 8º), obtermos 1,61904, e se dividirmos 55 (o 10º) por 34 (o 9º), obtemos 1,61764. Quanto mais adiante na sequência, mais essa proporção se aproxima de 1,61803.

A espiral de Fibonacci consiste em desenhar quadrados cujos tamanhos dos lados são os números da sequência, e, dentro de cada quadrado, desenhar um arco de 90° cujo raio é o tamanho do lado do quadrado. Deste modo, concatenando-se esses quadrados de maneira a formar uma espiral, obtemos a espiral de Fibonacci. A figura abaixo mostra um exemplo.

[9] Alguns autores iniciam a sequência com o número 0, seguido por 1. Aqui optou-se por iniciar a sequência com 1, seguido por 1, para facilitar o entendimento da espiral de Fibonacci mais adiante.



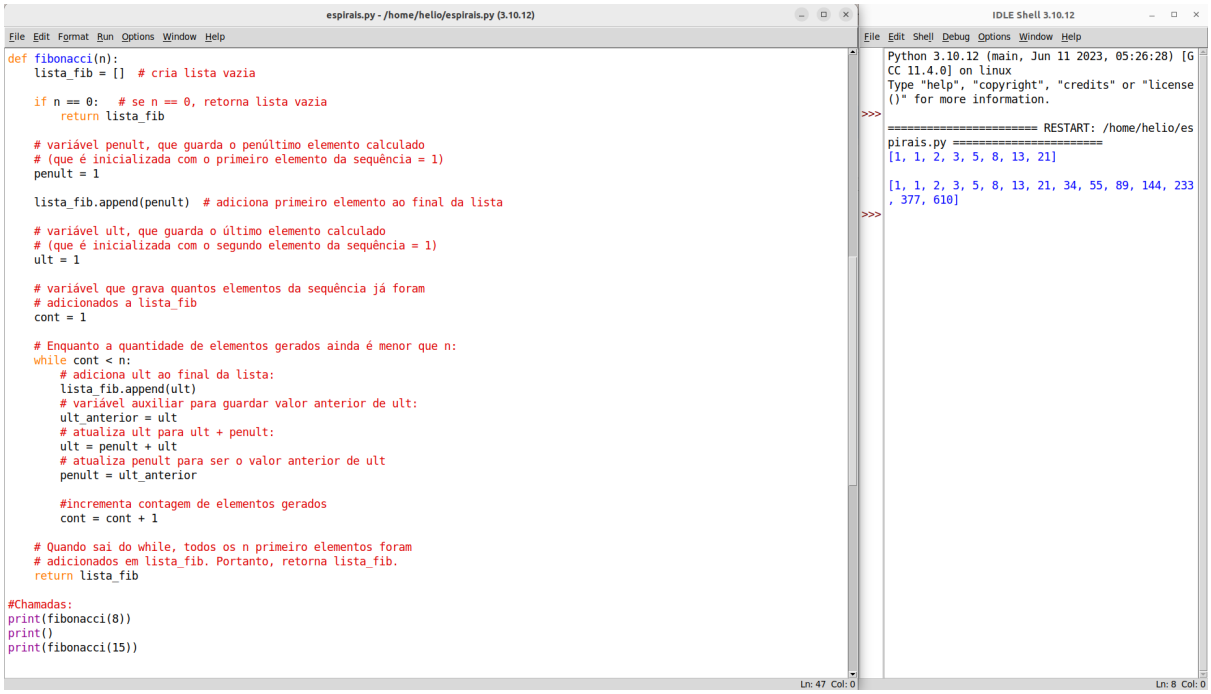
Fonte: https://www.infoescola.com/matematica/sequencia-de-fibonacci/

A sequência de Fibonacci, assim como sua proporção áurea e sua espiral, são encontradas em abundância na natureza, como na concha de caracóis, na cauda do cavalo-marinho, no formato de galáxias espirais, em diferentes proporções do corpo humano, no formato da molécula de DNA, etc. Por esse motivo, alguns consideram a proporção áurea como uma assinatura de Deus na Criação, e está frequentemente associada ao que é belo aos olhos humanos. Por exemplo, diversas obras de arte, obras arquitetônicas, e até composições de música clássica, utilizam essa proporção.

Para desenharmos a espiral, primeiramente é necessário definir uma função que calcula a sequência de Fibonacci. O resultado (retorno) dessa função será uma **lista** contendo os primeiros `n` números da sequência. Uma lista nada mais é que uma sequência de valores. O funcionamento da função consiste em definir, além da lista resultante (`lista_fib`), duas variáveis `ult` e `penult`, que, respectivamente, armazenam os valores dos dois últimos valores da sequência calculados até o momento. Ambas as variáveis são inicializadas com o valor 1, e então inicia-se um laço **`while`**, que realiza a atualização dessas duas variáveis com base na soma de seus valores atuais, e uma variável de laço `cont` que faz a contagem de quantos números da sequência já foram gerados. O laço repete enquanto `cont < n`. Logo, ele para de repetir quando `cont == n`, o que significa que a quantidade de números especificada na variável `n` terminou de ser gerada. Dentro do laço, a variável `penult` recebe o valor de `ult`, e `ult` recebe o valor de `penult + ult`. Além disso, `ult` (o último elemento calculado), é inserido ao final da lista. Dessa forma, ao final da execução desse laço, teremos a lista completa com os primeiros `n` valores da sequência de Fibonacci. Veja como fica:

```
def fibonacci(n):
    lista_fib = []  # cria lista vazia

    if n == 0:   # se n == 0, retorna lista vazia
        return lista_fib

    # variável penult, que guarda o penúltimo elemento calculado
```

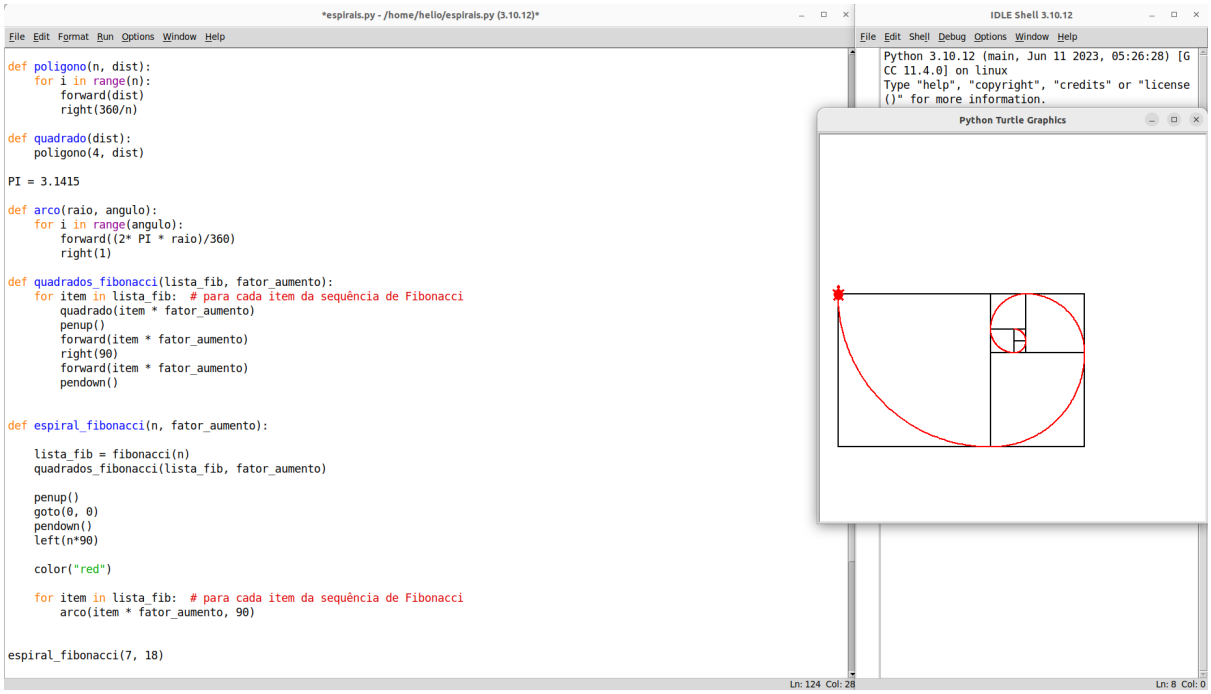
```
    # (inicializada com o primeiro elemento da sequência = 1)
    penult = 1

    # adiciona primeiro elemento ao final da lista:
    lista_fib.append(penult)

    # variável ult, que guarda o último elemento calculado
    # (inicializada com o segundo elemento da sequência = 1)
    ult = 1

    # variável que grava quantos elementos da sequência já foram
    # adicionados a lista_fib
    cont = 1

    # Enquanto a quantidade de elementos gerados é menor que n:
    while cont < n:
        # adiciona ult ao final da lista:
        lista_fib.append(ult)
        # variável auxiliar para guardar valor anterior de ult:
        ult_anterior = ult
        # atualiza ult para ult + penult:
        ult = penult + ult
        # atualiza penult para ser o valor anterior de ult
        penult = ult_anterior

        #incrementa contagem de elementos gerados
        cont = cont + 1

    # Quando sair do while, todos os n primeiro elementos
    # terão sido adicionados em lista_fib.
    # Portanto, retorna lista_fib.
    return lista_fib

#Chamadas:
print(fibonacci(8))
print()
print(fibonacci(15))
```



Uma vez gerada a sequência, podemos utilizá-la para o desenho dos quadrados. Para isso, podemos utilizar a mesma função `poligono` definida anteriormente. Criaremos também uma função chamada de `arco`, que recebe um ângulo e um raio, e desenha um arco. Essa função é muito similar à função `circulo` definida anteriormente (ver seção **De Polígonos para Círculos**), com a diferença que um círculo é um arco de 360°, enquanto a função `arco` permite desenhar arcos com diferentes ângulos (por exemplo, 90°, que é o ângulo que precisamos para desenhar a espiral de Fibonacci). A partir disso, duas funções são definidas: uma que desenha os quadrados (`quadrados_fibonacci`) e outra que desenha a espiral (`espiral_fibonacci`).

```
def poligono(n, dist):
    for i in range(n):
        forward(dist)
        right(360/n)

def quadrado(dist):
    poligono(4, dist)

PI = 3.1415


def arco(raio, angulo):
    for i in range(angulo):
        forward((2* PI * raio)/360)
        right(1)
```

```
def quadrados_fibonacci(lista_fib, fator_aumento):
    # Para cada item da sequência de Fibonacci
    for item in lista_fib:
        quadrado(item * fator_aumento)
        penup()
        forward(item * fator_aumento)
        right(90)
        forward(item * fator_aumento)
        pendown()

def espiral_fibonacci(n, fator_aumento):

    lista_fib = fibonacci(n)
    quadrados_fibonacci(lista_fib, fator_aumento)

    penup()
    goto(0, 0)
    pendown()
    left(n*90)

    color("red")

    # Para cada item da sequência de Fibonacci
    for item in lista_fib:
        arco(item * fator_aumento, 90)

# Chamada:
espiral_fibonacci(7, 18)
```



Note que, antes de desenharmos a espiral, mudamos a cor da caneta. Isso pode ser feito por meio do comando `color`. No caso, definimos a cor como vermelha usando `color("red")`. Mais adiante, no último exemplo deste livro, veremos como definir uma maior variedade de cores.

Note que dividimos a solução do problema em diferentes funções para tornar mais fácil seu entendimento.

Fractais

Uma última classe de figuras que abordaremos neste livro são os fractais. Um fractal (do latim *fractu*: fração, quebrado) é uma figura da geometria não clássica que consiste em um padrão que se repete, tal que suas partes menores repetem os traços (a aparência) do todo completo. De fato, espirais, como a de Fibonacci, também são um tipo de fractal, pois possuem a propriedade de poderem ser desenhadas por um período de tempo indeterminado, seguindo uma regra que define um padrão que se repete. Nesta seção, veremos alguns outros exemplos de fractais e duas formas de desenhá-los: da forma **recursiva** e utilizando **Sistemas-L**, ou Sistemas de Lindenmayer.

A primeira forma introduz um conceito importante na Ciência da Computação e na Matemática: a **recursividade**. Uma função é dita recursiva quando sua definição depende da própria definição da função. Tomemos o problema do cálculo do **fatorial** de um número. Por exemplo:

$$4! = 4 \times 3 \times 2 \times 1$$

$$3! = 3 \times 2 \times 1$$

$$2! = 2 \times 1$$

$$1! = 1$$

Note que a definição de cada um desses exemplos pode ser feita da seguinte forma:

$$4! = 4 \times 3!$$

$$3! = 3 \times 2!$$

$$2! = 2 \times 1!$$

$$1! = 1$$

Assim, percebemos que a fórmula que define a função matemática do fatorial é dada por:

$$n! = n \times (n-1)!$$
$$1! = 1$$

Podemos definir isso como uma função da forma convencional, chamando a função de *fat*:

$$fat(n) = n \times fat(n-1)$$
$$fat(1) = 1$$

Note que a definição da função deve ser feita considerando mais de um caso. O primeiro é chamado de **passo recursivo**, que envolve a chamada (aplicação) da própria função. Perceba que, para calcularmos o fatorial de 4 (4!), precisamos calcular o fatorial de 3 (3!); para calcular o fatorial de 3 (3!), precisamos calcular o fatorial de 2 (2!); e para calcular o fatorial de 2 (2!), precisamos calcular o fatorial de 1 (1!). Assim, em algum momento, a função será chamada para um caso que é conhecido como **caso base**, que é aquele caso que não envolve a necessidade de se aplicar a função recursivamente, geralmente possuindo uma solução (resposta) imediata. É o caso do 1!, que sempre será igual a 1. No momento em que a cadeia de chamadas recursivas à função alcança o caso base, cada chamada é retornada para a anterior que a chamou. Veja um exemplo de um teste de mesa, ou rastreio, da chamada recursiva *fat(4)*:

$$fat(4)$$

$\rightarrow 4 \times fat(3)$

$\rightarrow 3 \times fat(2)$

$\rightarrow 2 \times fat(1)$

$2 \times 1 = 2$

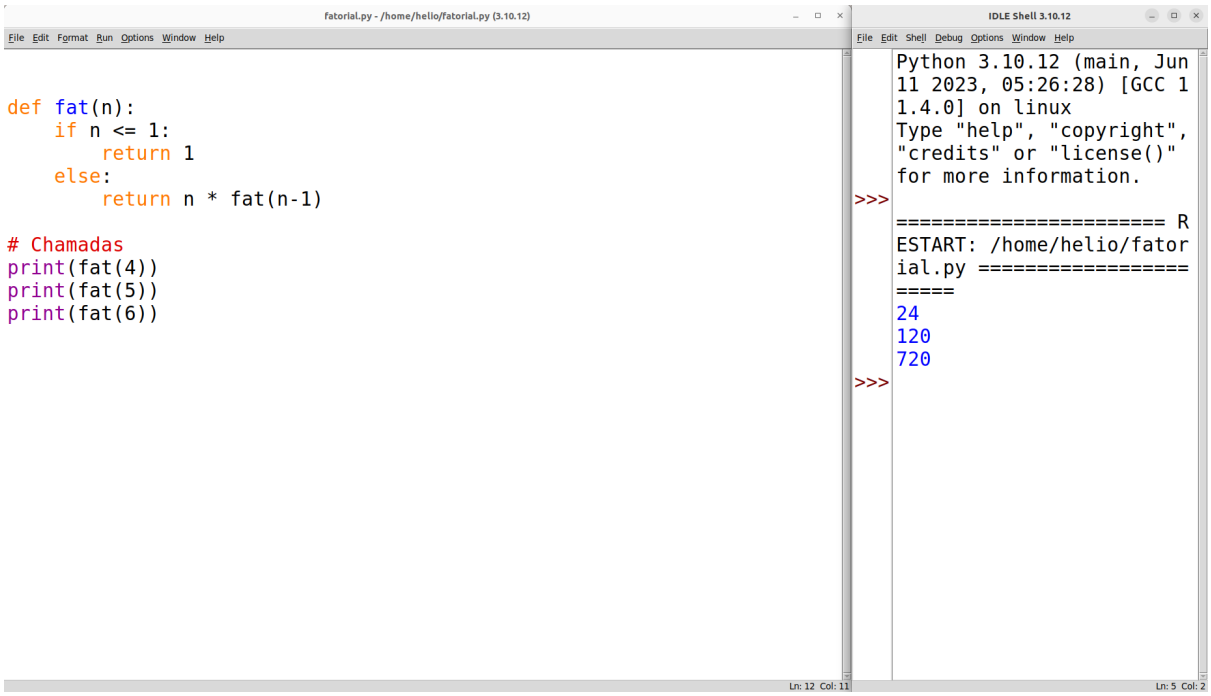
$3 \times 2 = 6 \leftarrow$

$4 \times 6 = 24 \leftarrow$

$24 \leftarrow$

Veja abaixo um exemplo de implementação da função `fat` em Python.

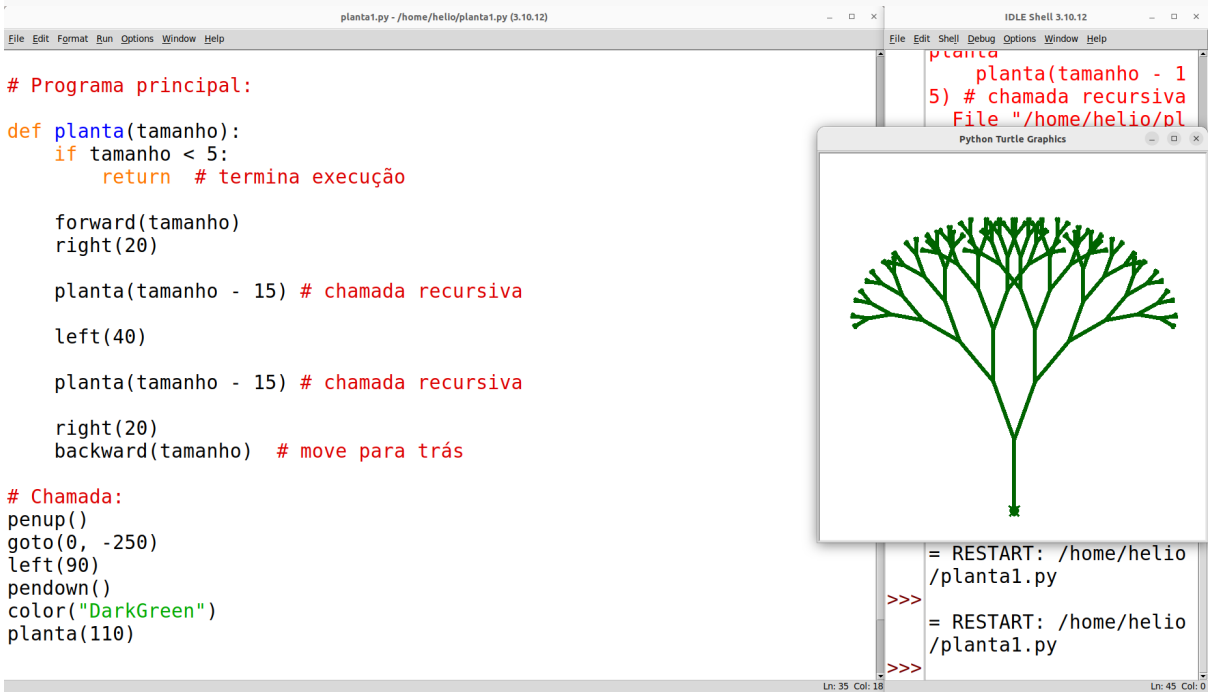
```python
def fat(n):
    if n <= 1:
        return 1
    else:
        return n * fat(n-1)

# Chamadas
print(fat(4))
print(fat(5))
print(fat(6))
```



Tendo compreendido a ideia de funções recursivas, tomemos um exemplo de fractal em forma de planta. A ideia é definir uma função chamada `planta`, que recebe por parâmetro o `tamanho` do ramo de uma planta, e recursivamente chama a própria função, porém com ramos cada vez menores. Quando o tamanho do ramo alcançar um valor pequeno (e.g. 5), a função para de ser chamada recursivamente. Ou seja, o **caso base** dessa função será quando o `tamanho` do ramo for menor ou igual a 5. Antes e depois de cada chamada recursiva, são desenhadas duas linhas em forma de V, representando um raminho. Veja como fica no Python:

```python
def planta(tamanho):
    if tamanho < 5:
        return  # termina execução

    forward(tamanho)
    right(20)
```

```
    planta(tamanho - 15) # chamada recursiva

    left(40)

    planta(tamanho - 15) # chamada recursiva

    right(20)
    backward(tamanho)  # move para trás

# Chamada:
penup()
goto(0, -250)
left(90)
pendown()
color("DarkGreen")
planta(110)
```



Perceba que, dentro da função `planta`, a própria função `planta` é chamada, no entanto, passando `tamanho - 15` por parâmetro. Assim, as chamadas recursivas vão se repetindo até que se alcança um tamanho menor que 5 (o caso base), e então a função termina sua execução.

Embora o uso de recursão para desenhar fractais seja muito interessante, abordaremos agora uma forma alternativa, que é o uso de Sistemas-L. O interessante dessa abordagem é que é possível facilmente adaptar nosso código para desenhar diferentes fractais, como veremos adiante.

Um **Sistema-L**, ou Sistema de Lindenmayer, é um sistema de reescrita paralela por meio de um tipo de gramática formal. Define-se um alfabeto de símbolos a partir dos quais são criadas *strings* (sequências de caracteres). Caracteres são formas tipográficas como as que encontramos no teclado do computador, tais como letras, números, pontos e sinais, e as *strings* são simplesmente sequências de caracteres, tais como as que usamos para formar palavras e textos. Já usamos *strings* várias vezes no decorrer deste material, inclusive em nosso primeiro programa: `print("Olá mundo!!")`. O que essa instrução faz é escrever na tela a *string* `"Olá mundo!!"`, que é uma sequência de caracteres. *Strings* são sempre criadas colocando-se uma sequência de caracteres entre aspas (“”), como fizemos nesse caso.

Nos Sistemas-L, as *strings* são geradas automaticamente por meio de uma **gramática** (também chamada de **conjunto de regras de produção***)*. Dessa forma, inicia-se com uma *string* inicial, também chamada de **axioma**, e a partir de repetidas aplicações das regras da gramática, gera-se uma *string* maior.

Para começar, vamos ver um exemplo simples de conjunto de regras:

A	Axioma
A → B	Regra 1: Mude A para B
B → AB	Regra 2: Mude B para AB

A execução do Sistema-L consiste em iniciar com o axioma e iterativamente aplicar as regras de produção. Assim, sempre que um caractere “A” é encontrado, ele é substituído por “B”, e sempre que um caractere “B” é encontrado, é substituído por “AB”. Supondo que executemos 7 iterações, teremos a seguinte sequência de *strings* geradas:

A (inicial)
B
AB
BAB
ABBAB
BABABBAB
ABBABBABABBAB
BABABBABABBABBABABBAB (final)

Esse Sistema-L de exemplo não serve para muita coisa. Portanto, vejamos um Sistema-L que pode ser utilizado para construir o famoso fractal conhecido como **Triângulo de Sierpinski**:

F−F−F	Axioma
F → F−G+F+G−F	Regra 1: Mude F para F−G+F+G−F
G → GG	Regra 2: Mude G para GG

Um exemplo de geração com 2 iterações seria:

F−F−F (inicial)

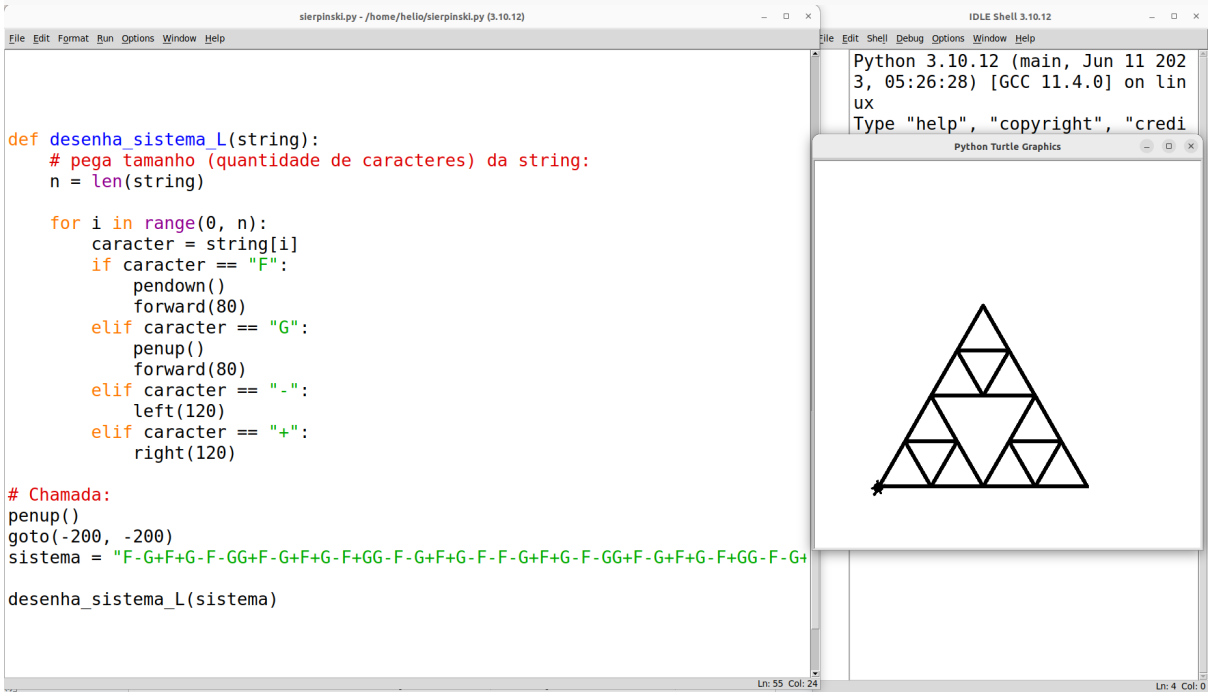
F−G+F+G−F−F−G+F+G−F−F−G+F+G−F

F−G+F+G−F−GG+F−G+F+G−F+GG−F−G+F+G−F−F−G+F+G−F−GG+F−G+F+G−F+GG−F−G+F+G−F−F−G+F+G−F−GG+F−G+F+G−F+GG−F−G+F+G−F (final)

Ok, mas você deve estar pensando: "O que fazemos com esse monte de letrinhas?" Imagine que cada uma dessas letrinhas/caracteres são instruções que podemos dar à nossa tartaruga. Por exemplo, quando encontramos um "F", então queremos nos mover 40 passos à frente com a caneta abaixada. Quando encontramos um "G", queremos nos mover 40 passos à frente com a caneta levantada. Quando encontramos um "−" queremos virar à esquerda em um ângulo de 120°. E, por fim, quando encontramos um "+" queremos virar à direita em um ângulo de 120º. Se lermos cada caractere da *string* final, na ordem em que aparecem, e realizarmos os comandos conforme mencionamos, teremos o seguinte desenho:

```
def desenha_sistema_L(string):
    # pega tamanho (quantidade de caracteres) da string:
    n = len(string)

    for i in range(0, n):
        caracter = string[i]
        if caracter == "F":
            pendown()
            forward(80)
        elif caracter == "G":
            penup()
            forward(80)
        elif caracter == "-":
            left(120)
```

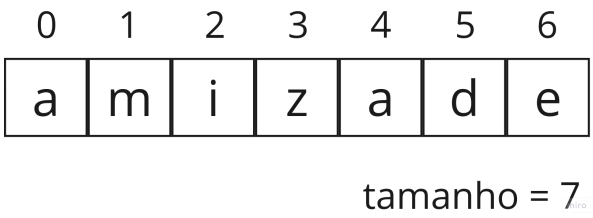
```
        elif caracter == "+":
            right(120)

# Chamada:
penup()
goto(-200, -200)
string = "F-G+F+G-F-GG+F-G+F+G-F+GG-F-G+F+G-F-F-G+F+G-F-GG+F-
G+F+G-F+GG-F-G+F+G-F-F-G+F+G-F-GG+F-G+F+G-F+GG-F-G+F+G-F"

desenha_sistema_L(string)
```



No exemplo acima, foi criada uma função chamada `desenha_sistema_L`, que recebe por parâmetro uma *string*. Em seguida, chamamos a função `desenha_sistema_L` passando a *string* que escrevemos como resultado da aplicação das regras de produção. Dentro da função, é feito o seguinte: cria-se uma variável `n` que recebe o valor do tamanho da *string* por meio da função `len`. Em seguida, cria-se um laço do tipo **`for`**, começando do `0` até `n - 1` (o segundo parâmetro do `range` não é incluído, portanto indo somente até o anterior). Isto serve para pegar os índices (ou posições) de cada caractere dentro da *string*.

Nas linguagens de programação em geral, o índice 0 representa a 1ª posição de uma sequência. Portanto, se quisermos pegar o primeiro caractere da *string* (no caso em questão, seria a letra "F"), então pegamos o caractere que está no índice 0. De modo similar, se quisermos pegar o segundo caractere (no nosso caso, o tracinho "-"), devemos pegar o caractere no índice 1, e assim por diante. Supondo que nossa *string* tenha 29 caracteres (como é o caso da *string* gerada após a 1º iteração), então o último caractere será o que se encontra no índice 28. Ora, mas por quê? Porque começamos a contar do índice 0. Veja, por exemplo, a palavra "amor". O índice

0 pegaria o caractere “a”, o índice 1 o caractere “m”, o índice 2 o caractere “o” e o índice 3 o caractere “r”. Note que o **tamanho** (isto é, a quantidade de caracteres) da palavra “amor” é 4. Porém, o último caractere da palavra é acessado pelo índice 3, que é exatamente o tamanho da palavra subtraído por um. Veja abaixo uma ilustração de uma *string* e seus índices:



É assim que acessamos o caractere em determinado índice: coloca-se o nome da variável que guarda a *string*, em seguida abre-se o colchete, coloca-se o índice, e fecha-se o colchete. Por exemplo, se quisermos pegar o 4º caractere da variável chamada `string`, faríamos `string[3]`.

Portanto, dentro do **`for`**, cada caractere é acessado pelo seu índice `i`, por meio da instrução `string[i]`, e então inicia-se um bloco de condicionais do tipo **`if... elif... elif...`** Já vimos anteriormente como funciona o **`if`** e o **`else`**. Então o que seriam esses “**`elif`**”? O *“elif”* nada mais é do que uma abreviação para *“else if”* (“senão, se”). Ele serve para simular um **`else`** seguido por um **`if`** em Python. Usamos **`elif`** quando temos várias alternativas e só queremos escolher uma, como é o caso aqui: queremos verificar se o caracter é igual a `"F"` (em cujo caso andamos desenhando), ou `"G"` (em cujo caso andamos sem desenhar), ou `"+"` (em cujo caso viramos à direita), ou `"-"` (em cujo caso viramos à esquerda). Note que, diferente do **`else`**, devemos colocar uma nova condição na frente de um **`elif`**. Resumindo, em Python:

```
if <condição>:
    #faça alguma coisa...
elif <condição>:
    #faça outra coisa...
elif <condição>:
    #faça ainda outra coisa...
else:
    #faça algo não previsto nas condições anteriores
```

quer dizer, em língua portuguesa, o seguinte:

```
se <condição>
    faça alguma coisa
senão, se <outra condição>
    faça outra coisa
senão, se <condição>
```

```
    faça ainda outra coisa
senão
    faça algo não previsto nas condições anteriores
```

Note que o `else`, após todos os `if/elif`, é opcional, servindo como o caso que ocorre somente se a execução não entrou em nenhum `if/elif`.

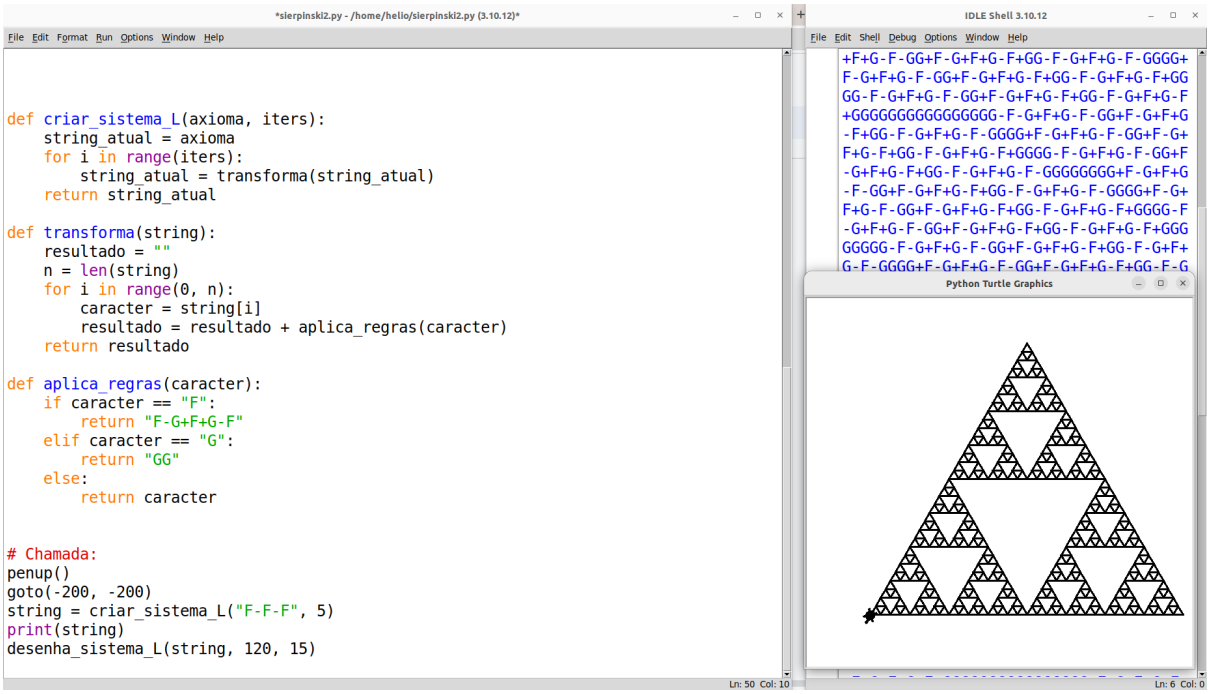
Assim, cada caractere na `string` é lido e interpretado, e uma ação é realizada como resultado. Pense na `string` como uma fita contendo uma sequência de letras, sendo que cada letra é uma instrução. Se for "F", você anda para frente riscando o chão. Se for "G", você anda para frente sem riscar o chão. Se for "–", você vira à esquerda por 120°. E se for "+", você vira à direita por 120°.

Apenas a título de curiosidade, note que o desenho do triângulo de Sierpinski consiste na repetição de um padrão de triângulos que lembra muito o desenho da *Triforce* dos jogos da franquia *The Legend of Zelda*. Você já jogou algum jogo dessa franquia? Se não, recomendo!

Enfim, vimos como um Sistema-L do tipo "Triângulo de Sierpinski" é capaz de desenhar tal triângulo, mas como podemos fazer para gerar a *string* de forma automática? Devemos elaborar um algoritmo que faça *n* iterações (repita *n* vezes) a substituição de cada caractere da *string* conforme as regras de produção definidas na gramática. Veja abaixo uma forma organizada de fazer isso:

```
def desenha_sistema_L(string, angulo, dist):
    # pega tamanho (quantidade de caracteres) da string:
    n = len(string)

    for i in range(0, n):
        caracter = string[i]
        if caracter == "F":
            pendown()
            forward(dist)
        elif caracter == "G":
            penup()
            forward(dist)
        elif caracter == "-":
            left(angulo)
        elif caracter == "+":
            right(angulo)

def criar_sistema_L(axioma, iters):
    string_atual = axioma
    for i in range(iters):
```

```
        string_atual = transforma(string_atual)
    return string_atual
def transforma(string):
    resultado = ""
    n = len(string)
    for i in range(0, n):
        caracter = string[i]
        resultado = resultado + aplica_regras(caracter)
    return resultado

def aplica_regras(caracter):
    if caracter == "F":
        return "F-G+F+G-F"
    elif caracter == "G":
        return "GG"
    else:
        return caracter

# Chamada:
penup()
goto(-200, -200)
string = criar_sistema_L("F-F-F", 5)
print(string)
desenha_sistema_L(string, 120, 15)
```



Note que criamos 3 novas funções: `criar_sistema_L`, `transforma` e `aplica_regras`. A primeira função é a que é chamada para iniciar o processo de construção da *string*. Ela recebe dois parâmetros: o `axioma`, que é a *string* inicial, e `iters`, que é a quantidade de iterações desejadas. Dentro da função, cria-se uma variável chamada `string_atual`, que é inicializado com o valor do `axioma`. Em seguida, inicia-se um laço `for` que repete `iters` vezes o seguinte: chama-se a função `transforma`, passando a `string_atual`, e atualiza-se a `string_atual` com o resultado retornado por essa função.

A função `transforma` é responsável por percorrer a *string*, isto é, passar por cada caractere, aplicando as regras. Portanto, cria-se uma variável `resultado` que é inicializada com uma *string* vazia (um texto sem nenhum caractere). Em seguida, define-se um laço `for` que cria uma variável de laço `i`, que começa no valor 0 (primeira posição da *string*) e termina na posição correspondente a `n-1` (última posição da *string*). Dentro do laço, captura-se o caractere na posição `i`, guardando-o em uma variável chamada `caracter` e, em seguida, chama-se a função `aplica_regras` passando o `caracter`.

Por fim chegamos à função `aplica_regras`. Ela recebe por parâmetro um caractere (variável `caracter`) e, por meio de condicionais, define as regras de produção.

Resumindo, a função `criar_sistema_L` repete várias vezes a função `transforma`, que, por sua vez, chama `aplica_regras` para cada caractere da *string*. Ao final, a primeira função retorna como resultado a *string* resultante de se aplicar regras nas várias iterações. Basta então passar a *string* resultante para a função `desenha_sistema_L`.

Note também que a última versão da função `desenha_sistema_L` (veja código antes da última imagem) está um pouco diferente da que definimos anteriormente. Para tornar a função mais flexível, definimos nela dois parâmetros adicionais: `angulo` e `dist`. Isso porque, para cada diferente padrão de fractal, o ângulo de giro muda. No caso do Triângulo de Sierpinski, o ângulo deve sempre ser 120°, mas outros exemplos, como veremos adiante, requerem outros ângulos. Também passamos a distância percorrida em cada `forward`. Com isso, podemos alterar o tamanho do fractal simplesmente mudando o valor desse parâmetro.

Ok, o código ficou razoavelmente grande, com várias funções. Mas existe uma grande vantagem em se fazer dessa forma (i.e., usando Sistemas-L). Diferentes fractais podem ser desenhados simplesmente alterando-se as regras de produção (função `aplica_regras`), as instruções que cada caractere representa (função `desenha_sistema_L`) e o ângulo de giro. Portanto, podemos **reutilizar** a maior parte do código, alterando somente esses pontos, e assim podemos desenhar qualquer fractal. Aqui temos mais um conceito importante em Ciência da Computação, especialmente na área de Engenharia de Software: a **reutilização**, ou **reuso**.

Vamos pegar outro exemplo famoso de fractal, a chamada “Curva de Dragão”. Ela é definida como segue:

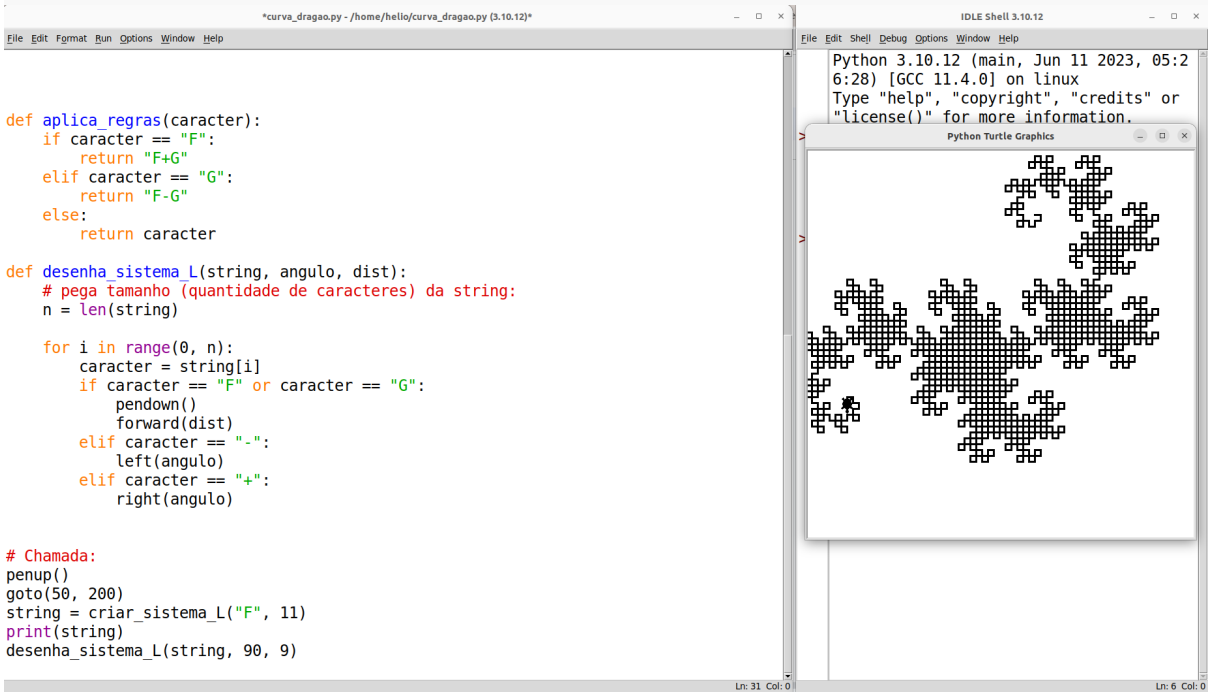
F	Axioma
F → F+G	Regra 1: Mude F para F+G
G → F-G	Regra 2: Mude G to F-G

O ângulo de giro deve ser de 90º. Outra diferença aqui é que tanto F quanto G significam que se deve avançar desenhando (isto é, com a caneta abaixada). Pronto! Temos todas as informações que precisamos para alterar nosso exemplo anterior para que agora desenhe a Curva de Dragão.

Primeiro, copie (Ctrl+C) todo o código do arquivo atual, crie um novo arquivo e cole (Ctrl+V). Uma vez feito isso, vamos alterar a função `aplica_regras` e `desenha_sistemas_L`, assim como o axioma e a quantidade de iterações, para podermos desenhar a Curva de Dragão. Note que não é necessário alterar nem a função `criar_sistema_L` nem `transforma`, e as alterações em `aplica_regras` e `desenha_sistemas_L` são fáceis de fazer pois seguem a mesma estrutura lógica.

```
# (...) criar_sistema_L e transforma continuam igual antes

def aplica_regras(caracter):
    if caracter == "F":
        return "F+G"
    elif caracter == "G":
        return "F-G"
    else:
        return caracter

def desenha_sistema_L(string, angulo, dist):
    # pega tamanho (quantidade de caracteres) da string:
    n = len(string)

    for i in range(0, n):
        caracter = string[i]
        if caracter == "F" or caracter == "G":
            pendown()
            forward(dist)
        elif caracter == "-":
            left(angulo)
```

```
        elif caracter == "+":
            right(angulo)

# Chamada:
penup()
goto(50, 200)
string = criar_sistema_L("F", 11)
print(string)
desenha_sistema_L(string, 90, 9)
```



> **Obs:** é aconselhável, definir um único *"*`if...`*"* para os dois casos, "F" e "G", pois ambos resultam no mesmo código. Para isso, basta utilizar o operador lógico `or` ("ou"). Aqui é possível pegar mais um gancho para um assunto que não tratamos com muito detalhe no conteúdo deste livro, que diz respeito aos operadores lógicos, também conhecidos como *operadores booleanos*.

Tente também por conta própria os seguintes Sistemas-L:

- Curva de Koch

F	Axioma
F → F+F−F−F+F	Regra 1: Mude F para F+F−F−F+F
Ângulo: 90º	

- Curva de Sierpinski “Ponta de Flecha”:

F	Axioma
F → G−F−G	Regra 1: Mude F para G−F−G
G → F+G+F	Regra 2: Mude G to F+G+F

Ângulo: 60°. Tanto “F” quanto “G” implicam em avançar desenhando (caneta abaixada).

Um último exemplo de fractal que colocaremos aqui, usando Sistema-L, será novamente um que se assemelha a uma planta. Anteriormente fizemos um fractal de geração de planta usando recursão. Veremos agora como poderíamos fazer uma planta, um tanto mais complexa e detalhada, usando Sistemas-L. Seja a seguinte configuração:

FX	Axioma
F → C0FF-[C1-F+F]+[C2+F-F]	Regra 1: Mude F para C0FF-[C1-F+F]+[C2+F-F]
X→C0FF+[C1+F]+[C3-F]	Regra 2: Mude X para C0FF+[C1+F]+[C3-F]

Ângulo: 25

Perceba que temos alguns caracteres novos aqui: “X”, “[”,“]”, “C0”, “C1”, “C2” e “C3”.

O “X” não faz nada, servindo apenas como elemento de geração da *string*. Pense nele como um “miolo” que serve apenas para permitir a geração de um tipo de ramo secundário da planta.

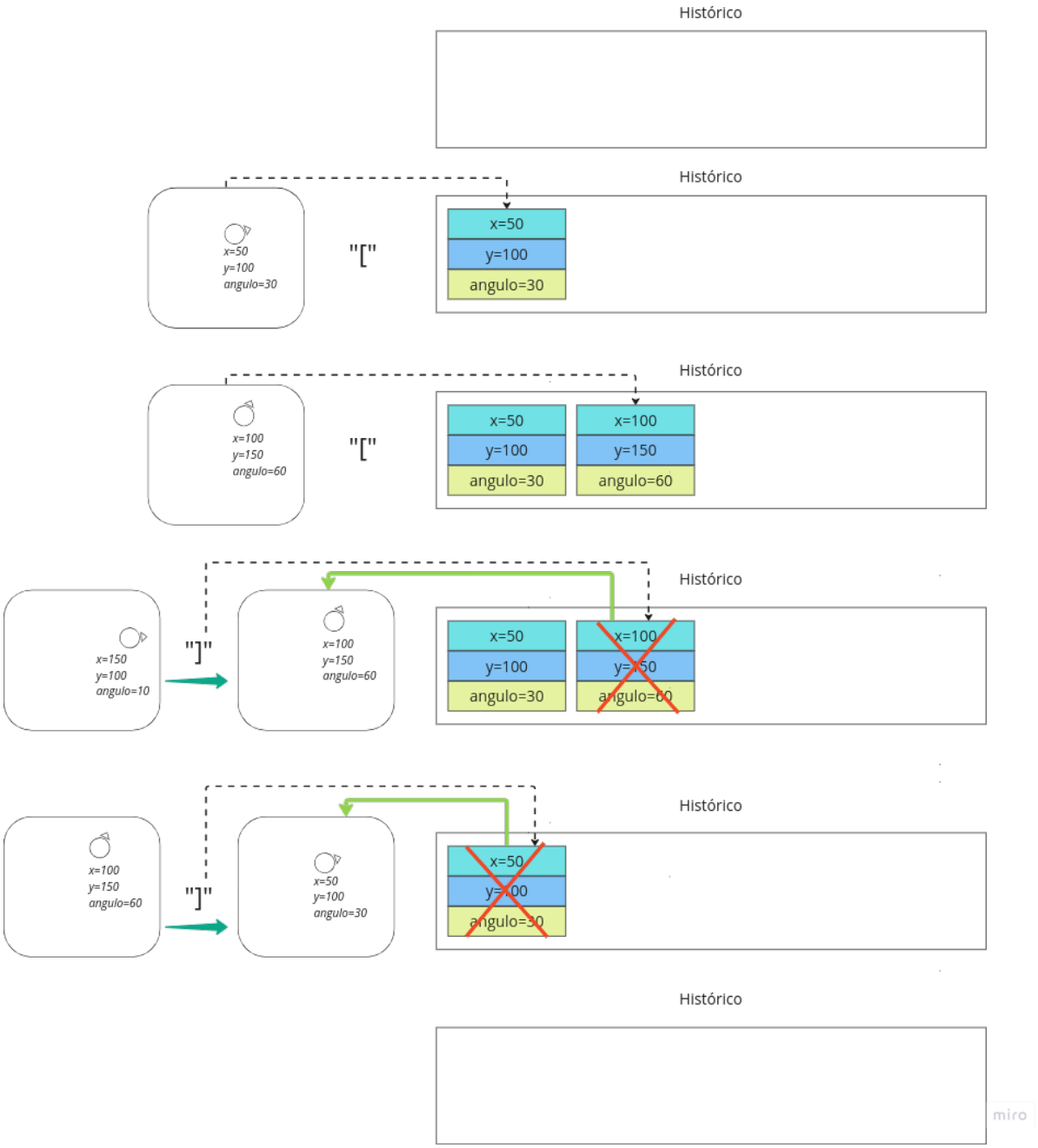
“C0”, “C1”, “C2” e “C3” representam mudanças de cores da linha desenhada pela tartaruga. Isso permitirá desenhar uma planta que possua o caule/galho de cor marrom (“C0”), e folhas em 3 tons de verde diferentes (“C1”, “C2” e “C3”).

O “[” e o “]” são os comandos novos mais importantes. Um “[” indica que as coordenadas *x* e *y,* e o ângulo de *direção* atual da tartaruga, devem ser armazenados em um histórico. Esse histórico é uma lista, e, sempre que encontramos um “[”, inserimos essas informações ao final da lista. Como as informações consistem em um conjunto de valores (*x, y, direção*), elas próprias são agrupadas também por meio de uma “mini-lista” (que podemos também chamar de **tupla**, ou **registro**), e essa “mini-lista” é então inserida ao final do histórico. Ao

encontrarmos um “]”, devemos remover os valores do final do histórico (ou seja, a última tupla de valores inserida no histórico), e fazer a tartaruga voltar para a posição *x* e *y* e apontar para o ângulo de *direção* contidos na tupla de valores removida. Dessa forma, será possível desenhar os ramos da planta, que devem retornar ao último ponto de ramificação para completar o desenho.

Esse histórico, em Ciência da Computação, é também conhecido como uma estrutura chamada de **pilha**. Pense em uma pilha de livros. Cada novo livro que você adiciona a essa pilha sempre é inserida no topo da pilha. Quando você remove um livro, deve-se também remover o livro que está no topo, nunca os que estão embaixo. Assim é com nosso histórico: o comando “[” adiciona (*x*, *y*, *direção*) ao final (“topo”) da lista, e o comando “]” remove (*x, y, direção*) do final (“topo”) da lista.

Abaixo é dada uma ilustração da estrutura e funcionamento desse histórico. O histórico começa vazio, e quando um “[” é lido, grava-se as informações atuais da tartaruga no final. Quando “]” é lido, remove-se as informações do final, que são utilizadas para restaurar o estado da tartaruga.

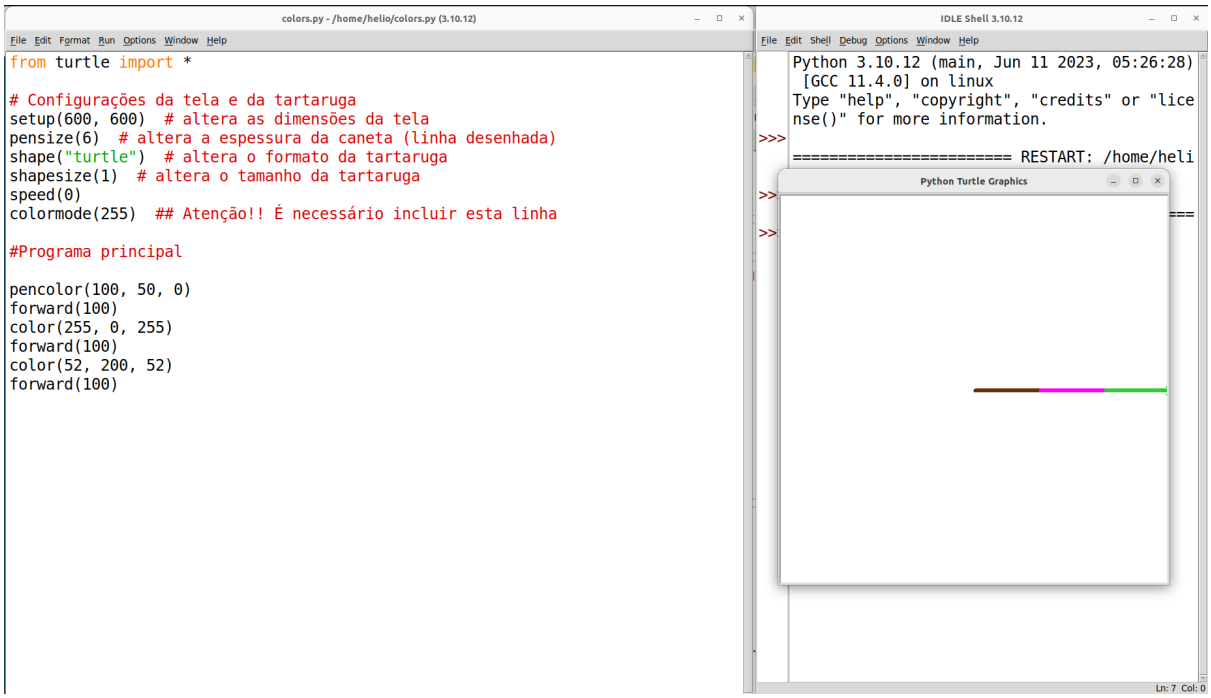


Vamos então ao nosso código. Como já vimos, basta alterar a função `aplica_regras`, inserindo as regras desse Sistema-L, e a função `desenha_sistema_L`, que agora fica um pouquinho mais complexa devido à necessidade de se alterar cores e, principalmente, de se tratar com o histórico.

Na função `desenha_sistema_L`, devemos primeiramente definir uma lista de cores, o que facilitará a troca de cores. Já vimos o comando color anteriormente, e vimos que existem cores predefinidas que podemos escolher por meio de uma *string* (por exemplo, *"red"*, *"DarkGreen"*, etc.). No entanto, é possível definir qualquer cor para a caneta usando o padrão de cores conhecido como RGB (*red*, *green, blue*). A ideia é combinar diferentes intensidades das cores vermelho, verde e azul para criarmos qualquer cor. Essas intensidades são configuradas por meio de números de 0 a 255, sendo o 0 a menor intensidade da cor, e 255 a

maior intensidade. Por exemplo, se configurarmos, se incluirmos no código o comando `color(255, 0, 0)`, teremos uma cor totalmente vermelha vibrante, pois estaremos atribuindo 255 à cor vermelha, 0 à cor verde e 0 à cor azul. Se incluirmos `color(0, 255, 0)`, teremos um verde vibrante. Se usarmos `color(255, 0, 255)`, teremos um tom rosa/violeta, conhecido como *fuchsia*. Se quisermos a cor preta, basta usar `color(0, 0, 0)`, isto é, a total ausência de cores. E se quisermos a cor branca, usamos `color(255, 255, 255)`, isto é, a junção de todas as cores, que resulta na cor branca. A brincadeira é muito parecida com a mistura de tintas de cores diferentes. Se você quiser testar mais cores usando o padrão RGB, faça uma busca na Internet por "RGB color picker". Um site recomendado é o https://www.w3schools.com/colors/colors_picker.asp.

Veja um exemplo abaixo do uso de diferentes cores usando o padrão RGB. Note que precisamos inserir o comando de configuração `colormode(255)` no início.



Segue abaixo, então, o trecho dentro da função `desenha_sistema_L` que cria uma lista de cores, sendo cada cor definida como uma tupla contendo os três valores para vermelho, verde e azul. Uma tupla é criada colocando-se itens separados por vírgulas enclausurados por parênteses.

```
def desenha_sistema_L(string, angulo, dist):
    # (...)
    # cria lista de cores:
     lista_de_cores = [
         (153, 51, 0),  # marrom
         (21, 81, 21),  # verde escuro
         (31, 122, 31)   # verde médio
         (41, 162, 42)   # verde claro
```

```
    ]
    for i in range(0, n):
        # (...)
```

É importante lembrar que os elementos em uma lista podem ser acessados por meio de seus índices, que funcionam da mesma forma que com *strings*, isto é, começando do índice 0. Ou seja, se quisermos pegar a cor marrom (primeira cor), acessamos o índice 0, se quisermos a segunda cor, acessamos o índice 1, a terceira, o índice 2, e a quarta, o índice 3. Lembra que as cores nas regras de produção são representadas como "C0", "C1", "C2" e "C3"? Pois esta é a ideia: vamos utilizar o número que vem na frente da letra "C" para pegarmos o índice da cor na lista de cores. Então, se encontrarmos "C" seguido por "0", trocamos a cor da caneta pela cor que está no índice 0 da lista de cores. Se encontrarmos "C" seguido por "1", trocamos pela cor que está no índice 1, e assim por diante. Formidável, não é mesmo? Pois é esse tipo de solução esperta que encanta muitos programadores!

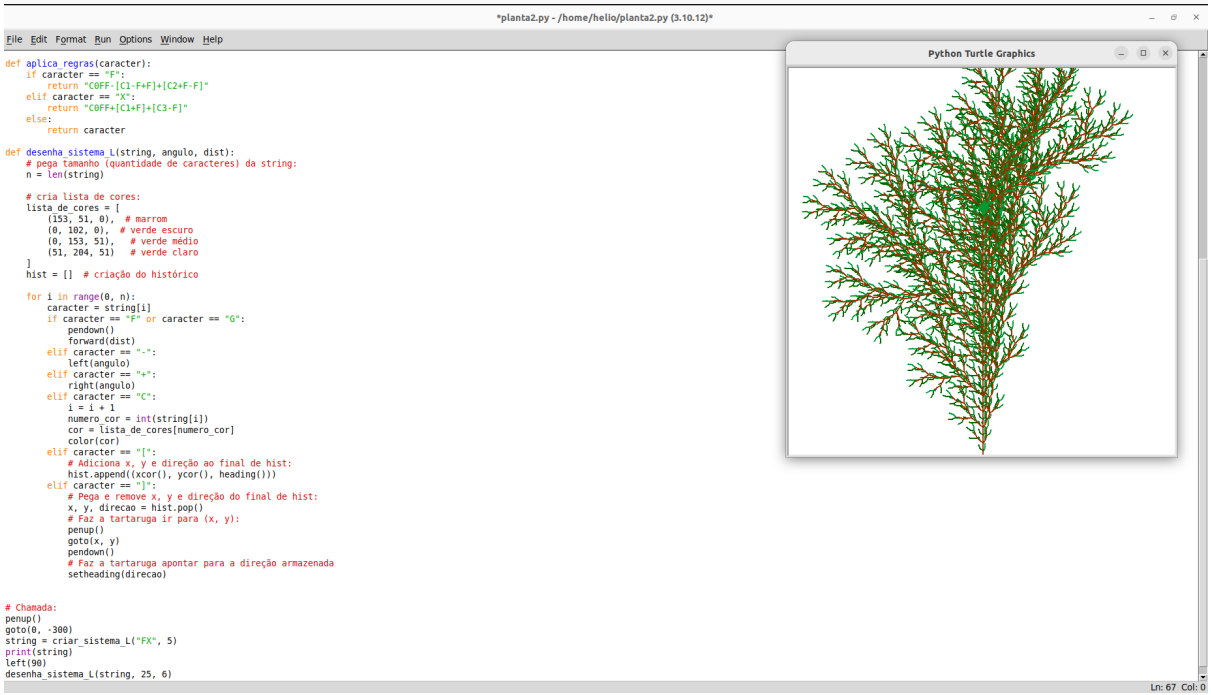
Veja abaixo como fica o trecho que verifica se o caractere atual é o "C". Aumenta-se em 1 o valor de *i* (que é o índice da *string* que está sendo processada) para pegar o próximo caractere, que será um dos números (0, 1, 2 ou 3); grava-se o número na variável `numero_cor`; usa-se a variável `numero_cor` como índice para pegar a cor na `lista_de_cores`, e então usa-se a cor no comando `color` para trocar a cor da caneta da tartaruga.

```
def desenha_sistema_L(string, angulo, dist):
    # (...)
    for i in range(0, n):
        caracter = string[i]
        # (...)
        elif caracter == "C":
            i = i + 1
            numero_cor = int(string[i])
            cor = lista_de_cores[numero_cor]
            color(cor)
        # (...)
```

Vejamos agora a parte relacionada ao tratamento do histórico, direcionada pelos caracteres "[" e "]". Antes de tudo precisamos criar uma lista vazia, que será guardada em uma variável chamada `hist`, representando o histórico. Isso deve ser feito também dentro da função `desenha_sistema_L`, antes do laço, assim como foi feito na criação da lista de cores. Em seguida, criamos dois novos condicionais `elif` dentro do laço: um para o "[" (guardar no histórico) e o outro para o "]" (recuperar do histórico e remover). Segue abaixo o código completo, incluindo o tratamento desses dois novos caracteres:

```
# (...) criar_sistema_L e transforma continuam igual antes

def aplica_regras(caracter):
    if caracter == "F":
        return "C0FF-[C1-F+F]+[C2+F-F]"
    elif caracter == "X":
        return "C0FF+[C1+F]+[C3-F]"
    else:
        return caracter


def desenha_sistema_L(string, angulo, dist):
    # pega tamanho (quantidade de caracteres) da string:
    n = len(string)

    # cria lista de cores:
    lista_de_cores = [
        (153, 51, 0),  # marrom
        (0, 102, 0),  # verde escuro
        (0, 153, 51),   # verde médio
        (51, 204, 51)   # verde claro
    ]
    hist = []  # criação do histórico

    for i in range(0, n):
        caracter = string[i]
        if caracter == "F" or caracter == "G":
            pendown()
            forward(dist)
        elif caracter == "-":
            left(angulo)
        elif caracter == "+":
            right(angulo)
        elif caracter == "C":
            i = i + 1
            numero_cor = int(string[i])
            cor = lista_de_cores[numero_cor]
            color(cor)
        elif caracter == “[”:
            # Adiciona x, y e direção ao final de hist:
            hist.append((xcor(), ycor(), heading()))
        elif caracter == "]":
```

```
            # Pega e remove x, y e direção do final de hist:
            x, y, direcao = hist.pop()
            # Faz a tartaruga ir para (x, y):
            penup()
            goto(x, y)
            pendown()
            # Faz a tartaruga apontar para a direção armazenada:
            setheading(direcao)

# Chamadas:
penup()
goto(0, -300)
string = criar_sistema_L("FX", 5)
print(string)
left(90)
desenha_sistema_L(string, 25, 6)
```



Note que, quando o caractere atual é "[", insere-se, como último elemento da lista `hist` (o histórico), uma nova tupla contendo `xcor()` (comando que pega a coordenada *x* atual da tartaruga), `ycor()` (comando que pega a coordenada *y* atual da tartaruga) e `heading()` (comando que pega a direção/ângulo da tartaruga), nessa ordem.

Quando o caractere atual é "]", usa-se o comando `hist.pop()` para remover a última tupla e guardar os valores de *x, y* e *direção* contidos nessa tupla nas variáveis `x`, `y` e `direcao`. Com isso, levantamos a caneta e fazemos a tartaruga ir para a posição `x` e `y`, e por fim pedimos que a tartaruga aponte para a direção por meio do comando `setheading()`. Com isso,

restauramos a tartaruga à posição em que ela estava na última vez em que foi dado o comando "[".

Além disso, modificamos também ligeiramente alguns dos parâmetros nas chamadas, tais como a quantidade de iterações, o ângulo, o axioma e a distância, e *voilá*… temos uma árvore/planta com vários ramos e de aspecto bastante natural.

Com isso terminamos essa saga. Se quiser brincar com mais exemplos de Sistemas-L, esta página possui alguns muito interessantes: https://www.kevs3d.co.uk/dev/lsystems.

Apêndice I

Dicas diversas de uso do Python Turtle

Comandos “Desfazer” e “Refazer”

Clique com o botão direito sobre a tela em branco e escolha a opção “Desfazer”. Isso desfará a última modificação realizada no código (blocos). A opção pode ser usada várias vezes para desfazer várias modificações. Pode-se também utilizar o atalho do teclado “Ctrl+Z”.

Clique com o botão direito sobre a tela em branco e escolha a opção “Refazer”. Isso refará a última modificação desfeita por meio do comando “Desfazer”. Pode-se também utilizar o atalho do teclado “Ctrl+Y”.

Cores RGB

Como explicado na seção sobre Fractais, é possível definir diferentes cores para a caneta da tartaruga, e o Blockly Turtle permite a definição de cores usando o padrão RGB por comando `color(red, green, blue)`. É possível definir as cores experimentalmente combinando-se diferentes intensidades dessas três cores. Cada intensidade pode receber um valor entre 0 e 255 (é necessário, no entanto, em algum local anterior no código, usar o comando `colormode(255)`). No entanto, muitas vezes é mais conveniente testar cores por meio de uma ferramenta do tipo *Color Picker*, que permite escolher cores por meio de uma interface gráfica intuitiva, retornando ao usuário os valores para as cores RGB. Há várias dessas ferramentas disponíveis na Web, mas sugerimos a seguinte: https://www.w3schools.com/colors/colors_picker.asp.