



**pythondrops.com**

Curso Básico de Python

*(c) 2019 Cleuton Sampaio*

## **Lição 2: Introdução quântica**



Este trabalho e todos os seus exemplos, mesmo que não explicitamente citado, estão distribuídos de acordo com a licença Creative Commons Atribuição-CompartilhaIgual 4.0 Internacional. Este é o link para os termos desta licença:

[https://creativecommons.org/licenses/by-sa/4.0/deed.pt\\_BR](https://creativecommons.org/licenses/by-sa/4.0/deed.pt_BR)

Você tem o direito de:

- **Compartilhar** — copiar e redistribuir o material em qualquer suporte ou formato
- **Adaptar** — remixar, transformar, e criar a partir do material para qualquer fim, mesmo que comercial.

Esta licença é aceitável para Trabalhos Culturais Livres.

O licenciante não pode revogar estes direitos desde que você respeite os termos da licença.

De acordo com os termos seguintes:

- **Atribuição** — Você deve dar o crédito apropriado, prover um link para a licença e indicar se mudanças foram feitas. Você deve fazê-lo em qualquer circunstância razoável, mas de nenhuma maneira que sugira que o licenciante apoia você ou o seu uso.
- **CompartilhaIgual** — Se você remixar, transformar, ou criar a partir do material, tem de distribuir as suas contribuições sob a mesma licença que o original.
- **Sem restrições adicionais** — Você não pode aplicar termos jurídicos ou medidas de caráter tecnológico que restrinjam legalmente outros de fazerem algo que a licença permita.

Avisos:

- Você não tem de cumprir com os termos da licença relativamente a elementos do material que estejam no domínio público ou cuja utilização seja permitida por uma exceção ou limitação que seja aplicável.
- Não são dadas quaisquer garantias. A licença pode não lhe dar todas as autorizações necessárias para o uso pretendido. Por exemplo, outros direitos, tais como direitos de imagem, de privacidade ou direitos morais, podem limitar o uso do material.

## Sumário

Fala sério.....	4
Wheels.....	6
Rodando o maze.....	7
Histórico do maze.....	8
Analisando o código.....	11
Módulos.....	11
Expressões condicionais.....	12
Classes e objetos.....	12
Criação de classes.....	13
Expressões lógicas e de controle de fluxo.....	18
Variáveis multivaloradas.....	20
Como funciona o algoritmo.....	20
Exercício.....	21

## Fala sério

Você já aprendeu pelo menos uma linguagem de programação, certo? Então, vamos pular aquela bobagem comum em todos os cursos e vamos direto para o nosso **Hello World bombado com esteróides!**

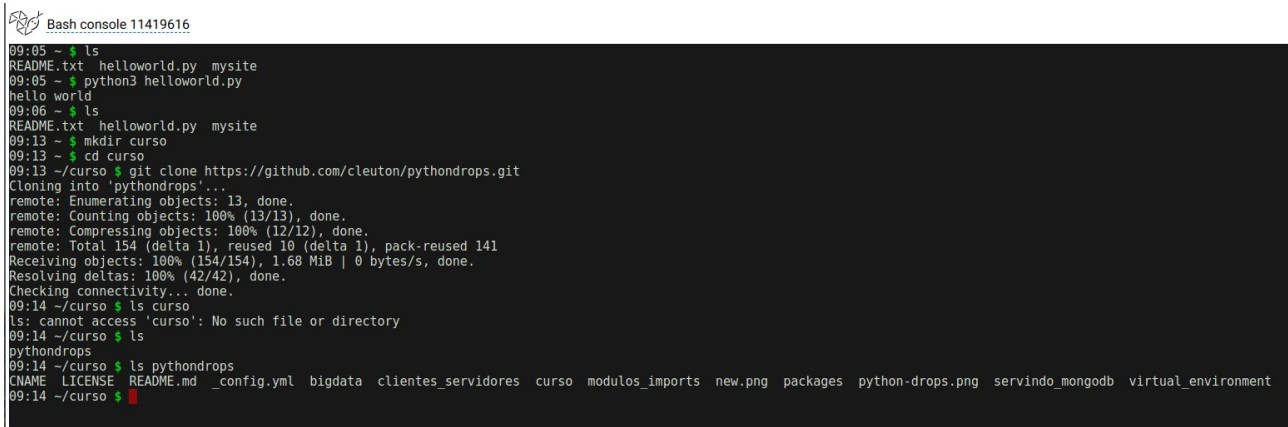
Acesse o site do repositório do curso e baixe um ZIP completo:

<https://github.com/cleuton/pythondrops>

Tem um botão “Clone or download”. Ao clicá-lo, você pode baixar um zip ou utilizar o git para clonar o repositório:

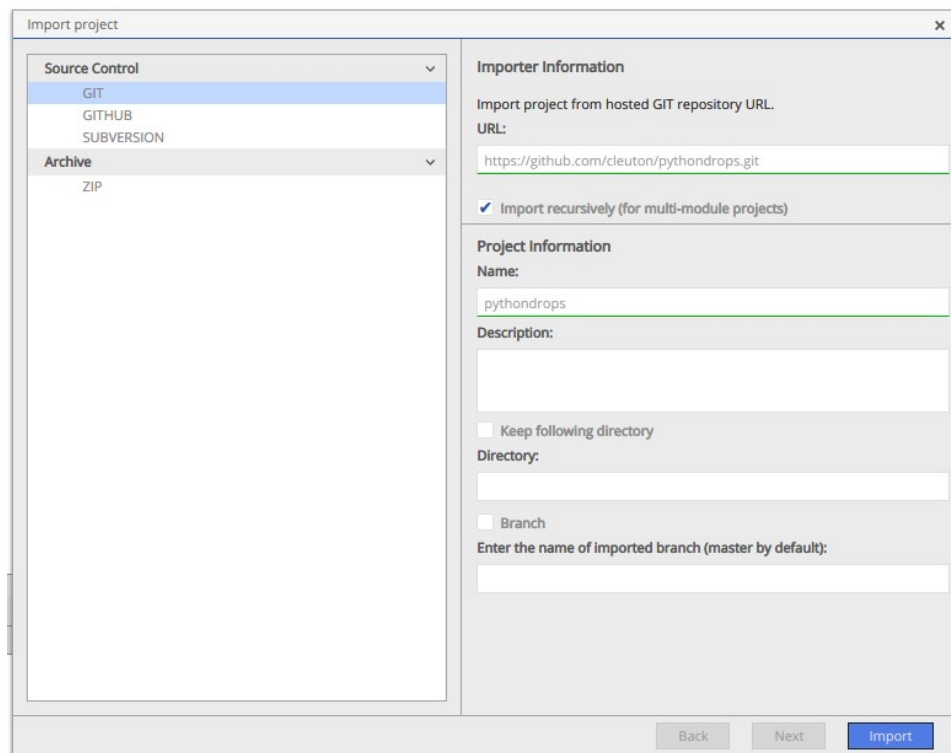
```
mkdir curso
cd curso
git clone https://github.com/cleuton/pythondrops.git
```

Se você estiver utilizando o PythonAnywhere, como eu recomendei, então abra uma console Bash (como eu ensinei) e execute os comandos que descrevi:

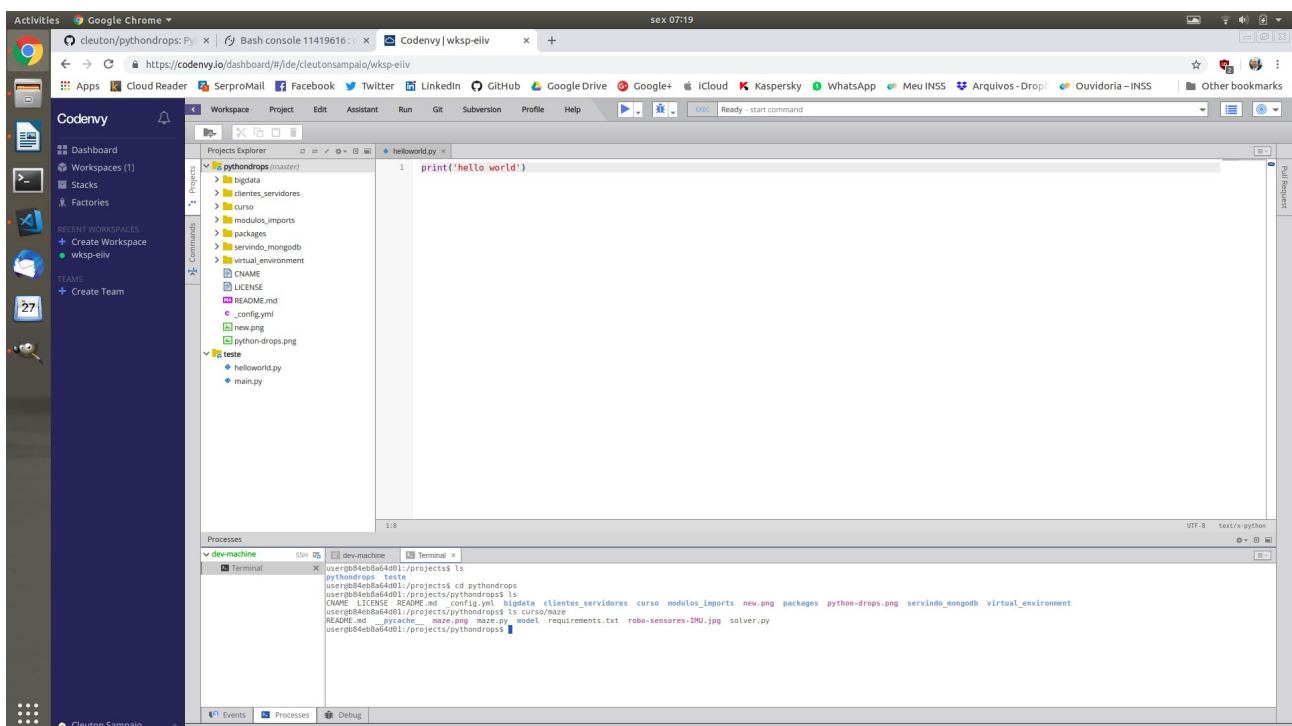


```
Bash console 11419616
09:05 ~ $ ls
README.txt helloworld.py mysite
09:05 ~ $ python3 helloworld.py
hello world
09:06 ~ $ ls
README.txt helloworld.py mysite
09:13 ~ $ mkdir curso
09:13 ~ $ cd curso
09:13 ~/curso $ git clone https://github.com/cleuton/pythondrops.git
Cloning into 'pythondrops'...
remote: Enumerating objects: 13, done.
remote: Counting objects: 100% (13/13), done.
remote: Compressing objects: 100% (12/12), done.
remote: Total 154 (delta 1), reused 10 (delta 1), pack-reused 141
Receiving objects: 100% (154/154), 1.68 MiB | 0 bytes/s, done.
Resolving deltas: 100% (42/42), done.
Checking connectivity... done.
09:14 ~/curso $ ls curso
ls: cannot access 'curso': No such file or directory
09:14 ~/curso $ ls
pythondrops
09:14 ~/curso $ ls pythondrops
CNAME LICENSE README.md _config.yml bigdata clientes_servidores curso modulos_imports new.png packages python-drops.png servindo_mongodb virtual_environment
09:14 ~/curso $
```

Se você estiver utilizando o Codenvy, é ainda mais fácil! Em sua workspace, selecione o menu “workspace” e “import project”:



Selecione “git” e digite a URL do projeto no Github. Pronto! Você importou o projeto inteiro:



Note, na janela “Terminal”, que a pasta foi criada e listamos o conteúdo da subpasta “curso”, onde está o projeto “maze”!

Agora, se você é uma pessoa teimosa, então terá que baixar um zip ou clonar o projeto para a sua própria máquina, onde deve ter instalado o Python. Se quiser clonar, terá que instalar o “git”:

- MS Windows: <https://git-scm.com/download/win>
- Linux (debian, ubuntu, mint etc): `sudo apt install git-core`
- Linux Redhat: `dnf -y install git`

Bom, dentro da pasta principal (“pythondrops”) tem uma subpasta “curso” e, dentro dela, uma subpasta “maze”.

## Wheels

Se você, apressadamente, tentou executar o programa “maze.py”, pode ter tomado o seguinte erro:

```
ImportError: No module named 'termcolor'
```

(Isso não acontece no PythonAnywhere)

Python, assim como a maioria das linguagens modernas, permite que você a estenda utilizando software desenvolvido pela Comunidade, os chamados “pacotes” (não confunda com os módulos importados dentro do código). O padrão atual de distribuição de pacotes é o formato “Wheel”, que permite instalar software pré-compilado ou construir a partir do código-fonte.

O “termcolor” é um destes “pacotes” e precisa ser instalado. Em Python, instalamos pacotes com o comando “pip”:

```
pip install termcolor
```

Se você tomou esse erro, então instale o pacote “termcolor” como eu mostrei. Aliás, guarde esse conceito para o futuro!

Se você acabou de instalar o Python, pode ser necessário atualizar as ferramentas: “pip”, “setuptools” e “wheels”:

```
python -m pip install --upgrade pip setuptools wheel
```

(pode ser necessário o “sudo” antes do comando)

Só uma observação: Em ambientes como o Codenvy, você tem que executar o “pip” com “sudo”:

```
sudo install termcolor
```

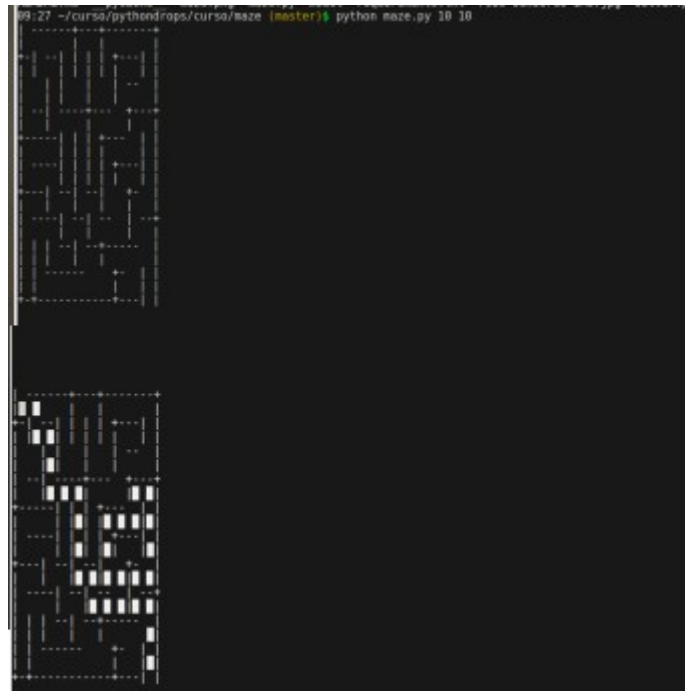
Sempre que você tomar um erro do tipo: “ImportError: No module...” já sabe que tem que usar o “pip” para instalar o pacote necessário.

## Rodando o maze

Agora, finalmente, vamos executar nosso “Hello world” bombado! Dentro da pasta “pythondrops”:

```
cd curso/maze  
python maze.py 10 10
```

Você deve ter algo assim na sua console (seja ela qual for):



Mandamos criar um labirinto com 10 linhas e 10 colunas. Ele mostrou o labirinto original e depois encontrou um caminho para a saída.

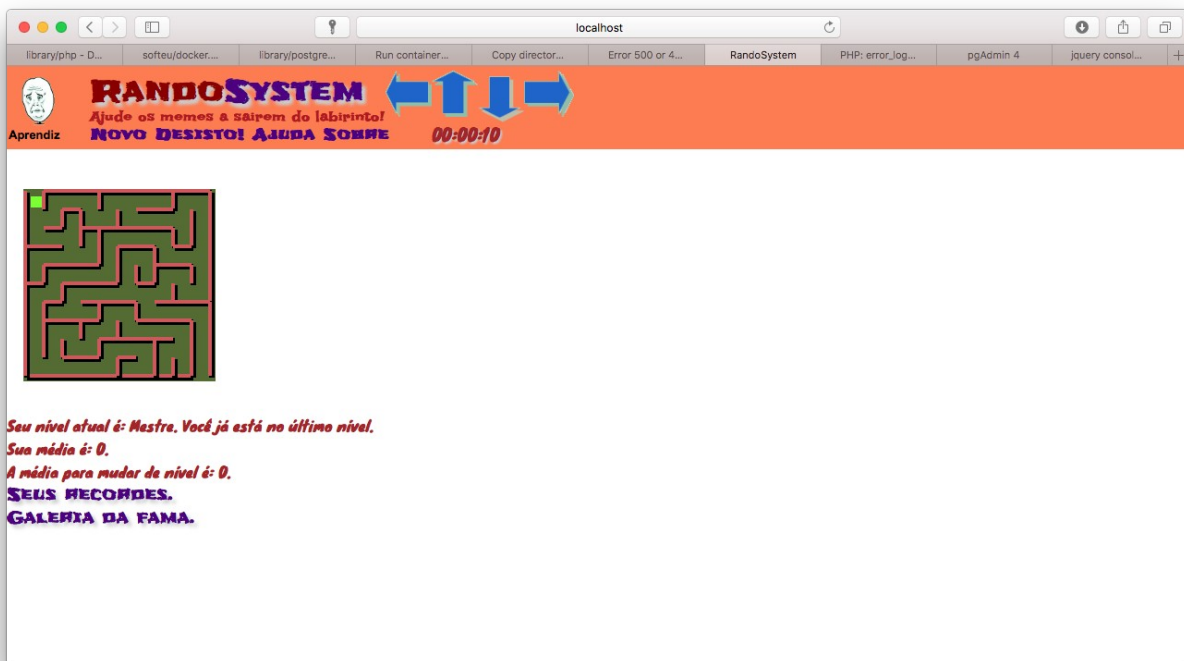
Se quiser criar um labirinto maior ou menor, é só informar a quantidade de linhas e a quantidade de colunas como argumento.

## Histórico do maze

Este programa já está na sua quarta “encarnação”! Eu o desenvolvi originalmente para ensinar Java na Faculdade, e depois, fui adaptando para vários usos. Fiz várias versões: Java, Javascript, PHP e Python. Uma das versões pode ser vista no meu projeto “fakestartup”, que é um site com um jogo online (certa vez eu o disponibilizei como um Jogo do Facebook):

<https://github.com/cleuton/fakestartup-sample>

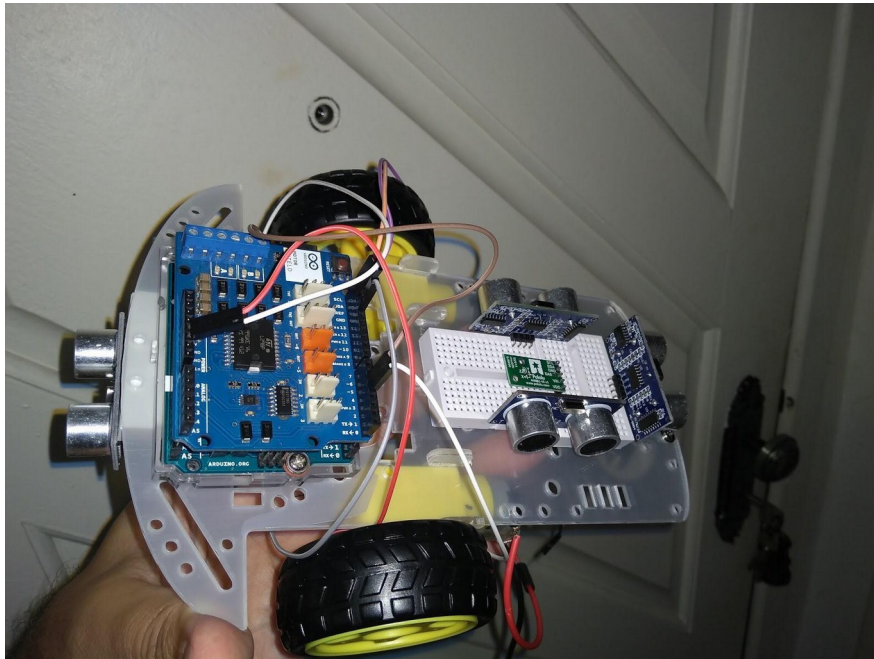




Neste projeto, o labirinto é gerado em PHP, no servidor, e renderizado em Javascript, no browser. E você pode navegar clicando nas setas!

Eu também fiz uma versão em Unity (<https://unity3d.com/pt>) e um dia espero criar um game completo 3D, mas fiz o labirinto 3D básico!

A última “encarnação” desse algoritmo foi para um projeto de robô que eu criei:



Esse robô era composto por dois processadores: Um Arduino embarcado e um Raspberry remoto. O Arduino era responsável pela operação básica do robô: andar (avante e ré), parar, virar (direita e esquerda), além dos sensores (4 sensores ultrassônicos de distância) e um sensor de movimento inercial. O Raspberry era a parte “inteligente” e recebia dados do Arduino, tomava decisões e enviava comandos a ele.

Eu usei a parte do “solver.py” (está na pasta do curso) para fazer o robô sair de um labirinto (criado com obstáculos de papelão. Utilizando a técnica de “backtracking” (<https://pt.wikipedia.org/wiki/Backtracking>) ele andava pelo labirinto até encontrar a saída. Era difícil, pois ao movimentar, o carrinho as vezes derrubava os obstáculos. Tive que fazer muitos ajustes finos, porém, funcionou!

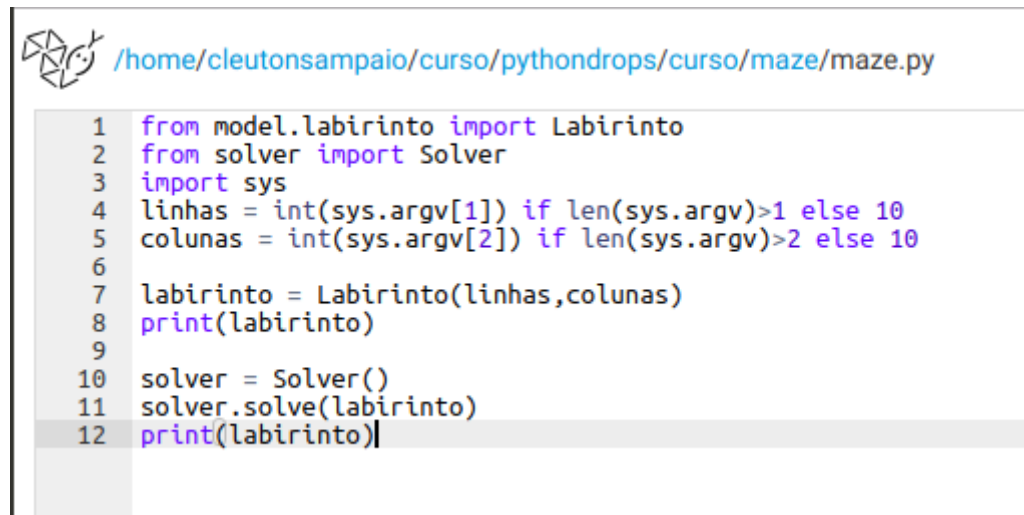
Infelizmente, devido a um “acidente canino”, o robô foi destruído. Estou montando outro, desta vez utilizando o SoC Omega2 (<https://onion.io/omega2/>), mas o código-fonte também será em Python.

Ele não está otimizado e, como foi migrado de outras linguagens, não está totalmente “pythônico”, mas serve para aprendermos o básico da linguagem Python.

## Analizando o código

Vamos analisar os arquivos deste exemplo, a começar pelo principal

Edite o arquivo “maze.py”. Se estiver no PythonAnywhere, já sabe como fazer (files).

A screenshot of a code editor window. The title bar shows a file icon and the path `/home/cleutonsampaio/curso/pythondrops/curso/maze/maze.py`. The code is as follows:

```
1 from model.labirinto import Labirinto
2 from solver import Solver
3 import sys
4 linhas = int(sys.argv[1]) if len(sys.argv)>1 else 10
5 columnas = int(sys.argv[2]) if len(sys.argv)>2 else 10
6
7 labirinto = Labirinto(linhas,columnas)
8 print(labirinto)
9
10 solver = Solver()
11 solver.solve(labirinto)
12 print(labirinto)
```

## Módulos

Este é o programa principal, ou módulo principal, da aplicação “maze”. A primeira coisa que notamos é a quantidade de coisas importadas, ou seja, que não estão definidas neste módulo. Os comandos “from ... import” e “import” importam módulos externos para este módulo.

Um módulo é um arquivo que contém código python, com uma determinada função. Podemos criar módulos para compartilhar código-fonte entre diversos programas de nossa aplicação.

Os módulos “model.labirinto” e “solver” foram criados por mim, mas o módulo “sys” é do próprio Python. Alguns dos módulos mais frequentemente usados do Python:

sys	os	numpy	requests	datetime
matplotlib	PIL	path	dateutil	optparse

Alguns fazem parte da “Python standard library” e podem ser vistos em:

<https://docs.python.org/3/library/>

A forma “from” permite importar apenas alguns elementos do módulo.

## Expressões condicionais

Continuando com o código “maze.py”, notamos as seguintes linhas:

```
linhas = int(sys.argv[1]) if len(sys.argv)>1 else 10
colunas = int(sys.argv[2]) if len(sys.argv)>2 else 10
```

Em Python isso seria equivalente ao operador ternário do Java. Utilizamos muito expressões condicionais, em vez de comandos “if”, pois isto reduz o tamanho do código. Nos exemplos, eu inicializo duas variáveis numéricas com os números informados na linha de comando.

A lista “sys.argv” contém vários elementos, começando do zero, que é o nome do script (“maze.py”) indo até o último argumento informado. Todos são strings, portanto, se eu quero números inteiros, devo convertê-los com a função “int()”. Só que eu só posso fazer isso se o usuário informou os argumentos, daí o teste “len(sys.argv[nn])”. Se o tamanho da lista for maior que 1, então ele informou a quantidade de linhas e, se for maior que 2, então informou também a quantidade de colunas. Caso contrário, eu atribuo o valor 10 para ambas.

Em Python, não informamos o tipo de dados (domínio) das variáveis no momento da declaração. Ele é inferido pelo valor atribuído.

## Classes e objetos

Continuando com o código, temos:

```
labirinto = Labirinto(linhas,colunas)
print(labirinto)

solver = Solver()
solver.solve(labirinto)
print(labirinto)
```

Lembre-se que importamos dois elementos:

```
from model.labirinto import Labirinto
from solver import Solver
```

Ambos são classes! Se olharmos os arquivos “model/labirinto.py” e “solver.py”, veremos isto:

```
--- maze/labirinto.py:  
class Labirinto:  
--- solver.py:  
class Solver:
```

Para instanciar a classe “Labirinto” nós...

***(Peraí! Você sabe o que significa “instanciar” uma classe? Não? Conheça Orientação a Objetos? Caso contrário, leia isso: [https://pt.wikipedia.org/wiki/Inst%C3%A2ncia\\_\(classe\)](https://pt.wikipedia.org/wiki/Inst%C3%A2ncia_(classe)))***

Como eu dizia, em Python, para instanciar a classe “Labirinto”, basta atribuí-la desta forma:

```
labirinto = Labirinto(linhas,colunas)
```

A variável “labirinto” agora aponta para um objeto do tipo “Labirinto()”, inicializado com os valores informados de linhas e colunas.

Depois, eu passei a variável para a função “print()”, mas não invoquei nenhum método. A função “print()” espera um string, portanto a classe “Labirinto()” tem que saber converter alguma coisa para string. Veremos isto depois.

Depois, instanciei a classe “Solver()” e invoquei o método “solve()”, passando a instância do labirinto para ele:

```
solver = Solver()  
solver.solve(labirinto)
```

O método “solve()” certamente alterou alguma coisa dentro do objeto “labirinto”. Então, eu invoco o “print()” novamente, passando o mesmo labirinto, e notamos que ele foi preenchido com um caminho, da entrada até a saída. Foi isso que o método “solve()” fez.

## Criação de classes

Agora, edite o arquivo “model/labirinto.py”:

Vou mostrar só um pedacinho:

```
from model.celula import Celula
from model.stack import Stack
from model import constantes
from random import randint
from termcolor import colored

class Labirinto:

    def __init__(self, linhas=10, colunas=10):
        self.linhas = linhas
        self.colunas = colunas
        self.celulas = []
        self.celulaInicial = None
        self.celunaFinal = None
        self.valido = False
        self._corrente = None
        self._proxima = None
        self._qtdTotal = 0
        self._qtdVisitadas = 0
        self._pilha = Stack()
        self.inicializar()
        self.caminho = None

    def inicializar(self):
        contador = 0
        while contador < 4:
            self.celulas=[]
            for i in range(self.linhas):
                linha = []
                for j in range(self.colunas):
                    celula = Celula()
                    celula.y=i
                    celula.x=j
                    celula.inicio=False
            ...
```

Notou algo interessante? Cadê as chaves (“{ }”)? Não tem! Em Python, como em qualquer outra linguagem, blocos de comandos precisam ser identificados. Em **Java**, por exemplo, é assim:

```
class Labirinto {...}
if (x > 5) {...}
for (int a=0; a<7; a++) {...}
while x<10 {...}
```

Em Python é diferente! Usamos a indentação para delimitar blocos de comando. Vejamos alguns exemplos:

```
class Labirinto:
    <resto da classe>
```

```
if x > 5:
    <comandos se a condição for verdadeira>

for a in range(7):
    <resto dos comandos a serem repetidos>

while x<10:
    <resto dos comandos a serem repetidos>
```

Sempre utilizamos 4 espaços, de acordo com o guia de estilo Python

(<https://www.python.org/dev/peps/pep-0008/#indentation>) e não misturamos tab com espaço. Cada nível de bloco de comando é indentado em 4 espaços. Vejamos um exemplo:

```
def procurar(self):
    buffer = None
    while(self._pilha.top):
        self._visitadas[self._corrente.y][self._corrente.x]=True
        if self._corrente.fim:
            return
        proxima = None
        for parede in range(4):
            if not self._corrente.paredes[parede]:
                if parede == constantes.NORTE and self._corrente.y == 0:
                    continue
...

```

Cada faixa colorida marca o início de um novo bloco de comandos. Temos o bloco principal, onde está a função “procurar”, o bloco da própria função “procurar”, o bloco do “while”, o bloco do primeiro “if”, o bloco do “for”, o bloco do segundo “if” e, finalmente, o bloco do terceiro “if”.

Um erro na indentação pode passar despercebido e causar erros de lógica!

Uma classe é declarada assim:

```
class Labirinto:
```

E um novo bloco se inicia, contendo os membros da classe. Declaramos métodos assim:

```
    def inicializar(self):
```

Note a indentação do comando “def”, pois ele está dentro do bloco da classe. E note que passamos uma variável entre parêntesis, que é um argumento para o método utilizar. No final, temos os dois pontos (“:”) que marcam o início de outro bloco de comandos (o do método).

Outro detalhe interessante é como declaramos atributos de uma instância da classe. Em java, faríamos algo assim:

```
class Labirinto {
    int linhas;
    int colunas;
    ...
}
```

“linhas” e “colunas” são atributos das instâncias da classe Labirinto, e, para acessá-las, faríamos assim:

```
Labirinto lab = new Labirinto();
lab.linhas = 10;
System.out.println(lab.linhas );
```

Em Python, declaramos atributos de instância de classe simplesmente atribuindo algo a uma variável de instância. Usando a console Python, eu fiz um teste:

```
>>> class Teste:
...     pass
...
>>> x = Teste()
>>> x.nome = "Meu nome"
>>> y = Teste()
>>> print(x.nome)
Meu nome
>>> print(y.nome)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'Teste' object has no attribute 'nome'
```

A classe “Teste” não possui membro algum. Eu coloquei um comando “pass” (que significa algo como “nenhuma operação”) para não dar erro. Depois, eu instanciei uma variável da classe “Teste”:

```
>>> x = Teste()
```

Depois, eu atribui um string a uma variável “nome”, associada à variável “x”, que é uma instância da classe:

```
>>> x.nome = "Meu nome"
```

Como a classe “Teste” não possui um atributo “nome”, então, esse atributo existe apenas para a instância apontada por “x”. Se eu criar outra variável e tentar mostrar o conteúdo de “nome”, tomarei erro:

```
>>> y = Teste()
...
>>> print(y.nome)
```



AttributeError: 'Teste' object has no attribute 'nome'

Em Java, podemos declarar um método “construtor”, invocado quando a classe é instanciada:

```
class Labirinto {
    int linhas;
    int colunas;
    public Labirinto(int linhas, int colunas) {
        this.linhas = linhas;
        this.colunas = colunas;
    }
    ...
}
```

Então, eu posso instanciar a classe passando o valor que eu quero para as linhas e colunas:

```
Labirinto a = new Labirinto(5,10);
```

Em Python, utilizamos “**dunders**” para declarar métodos especiais. Um “dunder” é um nome especial, que é cercado por dois caracteres sublinha (“\_\_”). O método “\_\_init\_\_” é invocado quando uma instância da classe é criada.

Quando um método de uma classe Python é invocado, ele recebe automaticamente um parâmetro posicional, que representa o objeto que estamos trabalhando. Tradicionalmente, damos a ele o nome de “self”. E ele não é informado ao invocar o método. Usamos o parâmetro “self” para criar ou acessar atributos de instância:

```
def __init__(self, linhas=10, colunas=10):
    self.linhas = linhas
    self.colunas = colunas
    self.celulas = []
```

O primeiro comando do método “\_\_init\_\_” cria uma variável de instância chamada “linhas”, dentro do objeto representado por “self”. Os outros, criam outras variáveis. Note que os parâmetros “linhas” e “colunas” do método “\_\_init\_\_” foram declarados com valor inicial (10), o que os torna opcionais. Se não forem informados, o valor 10 será assumido. Podemos instanciar labirintos desta forma:

```
l = Labirinto() # assume valores 10 e 10
m = Labirinto(5,5)
```

Ah, o caracter “#” representa início de comentário!

Antes de terminar de falar de classes, deixe me dizer só uma coisa: O controle de acesso aos métodos é diferente do java! Aquela história de “private”, “public” e “protected” é diferente. Para não confundir muito, digamos que membros cujo nome inicia por “\_” (sublinha) são considerados “protected”, ou seja, só o próprio código da classe (ou de suas classes derivadas) pode acessá-los.

## Expressões lógicas e de controle de fluxo

Observando o código de “model/labirinto.py”, notamos várias expressões lógicas, tanto em comandos “if” como em “while”. Veja alguns exemplos pinçados do código:

Linha	Comando	Explicação
26	while contador < 4:	Repete o bloco enquanto a variável “contador” for menor que o valor 4
75	while True:	Repete o bloco para sempre. <b>True</b> é o valor lógico “verdadeiro” em Python
76	if self.pilha.top:	Estamos testado se o valor da propriedade “top”, do objeto “pilha” da nossa instância (“self”) é diferente de zero. Qualquer coisa diferente de zero e <b>None</b> é <b>True</b> .
77	if self.isDeadEnd(self._corrente) or \ self._corrente.fim or \ self._corrente.inicio:	Note o uso da disjunção na condição. Trata-se de uma condição composta e o <b>or</b> é a disjunção (“ou”). Também podemos ter a conjunção <b>and</b> (“e”) e a negação <b>not</b> (“não”). O caractere “\ ” indica que o comando continua na próxima linha
132	if vizinha==constantes.NORTE:	O operador “==” testa se os dois lados contém o mesmo valor (igualdade). Temos também “<=” (menor ou igual) e “>=” maior ou igual. A negação seria: <b>not</b> vizinha==constantes.NORTE

Vamos ver um exemplo de bloco composto de “if” (linha 77):

```
if self.isDeadEnd(self._corrente) or \  
    self._corrente.fim or \  
    self._corrente.inicio:  
    self._proxima = self.pilha.pop()  
    self._corrente = self._proxima  
else:  
    self._proxima = self.pegarVizinha(self._corrente)  
    self.quebrarParedes(self._corrente, self._proxima)  
    self.pilha.push(self._corrente)  
    self._proxima.visitada = True  
    self._qtdVisitadas = self._qtdVisitadas + 1  
    self._corrente = self._proxima
```

Vemos um bloco composto por “if” e “else”. Se a condição for verdadeira, o bloco “if” é executado, caso contrário, o bloco “else” é executado.

E, na linha 132, temos um bloco com “if” / “elif” / “else”:

```
if vizinha==constantes.NORTE:
    if celula.y>0:
        cel = self.celulas[(celula.y - 1)][celula.x]
elif vizinha==constantes.SUL:
    if celula.y < (self.linhas - 1):
        cel = self.celulas[(celula.y + 1)][celula.x]
elif vizinha==constantes.LESTE:
    if celula.x < (self.colunas - 1):
        cel = self.celulas[celula.y][(celula.x + 1)]
else:
    if celula.x > 0:
        cel = self.celulas[celula.y][(celula.x - 1)]
```

O comando “elif” significa “else if”. Temos o famoso “ninho de if”. Se a condição for verdadeira, o bloco “if” é executado, caso contrário, testa a condição do “elif”. O último “else” é a negação do último “elif”. Se não entrar em nada, o bloco “else” é executado.

Finalmente, gostaria de chamar a atenção para os blocos “for”. O comando “for” é bem diferente em Python. Por exemplo, na linha 152 temos este comando:

```
for i in range(self.linhas):
```

Ele vai inicializar a variável “i” com zero e vai repetir o bloco de comandos, incrementando “i” a cada passagem, até que o valor de “i” seja igual ao valor de “self.linhas”, quando a execução será interrompida. Por exemplo, se “self.linhas” for igual a 3, ele repetirá o bloco de comandos para os valores de “i”: 0,1,2. Quando “i” for igual a 3, o bloco não será mais repetido. O equivalente em Java seria:

```
for (var i=0; x < self.linhas; i++) {
```

Outra forma do “for”, que não utiliza a função “range()”, é vista na linha 188:

```
for celula in self.caminho:
```

Neste caso, estamos iterando em uma lista de objetos, representada pela variável “self.caminho”. Para cada elemento contido nela, a variável local “caminho” assume seu valor. Um exemplo mais simples:

```
>>> numeros = [10, -1, 40, 0, 7]
>>> for n in numeros:
...     print(n)
...
10
-1
40
0
7
```

## Variáveis multivaloradas

Em Python, temos vários tipos multivalorados (array, list, dictionary e tuple), mas vamos focar no mais simples: List.

Uma lista (List) pode conter elementos heterogêneos, e é sempre dinâmica, ou seja, é capaz de aumentar seu tamanho:

```
>>> lista = [10, 'a', True, 100.50]
>>> print(lista[0]) # primeiro elemento
10
>>> print(lista[-1]) # último elemento
100.5
>>> print(type(lista[2])) # tipo de dados do terceiro elemento da lista
<class 'bool'>
>>> lista.append('chiclete')
>>> lista
[10, 'a', True, 100.5, 'chiclete']
```

Uma lista pode conter objetos também.

## Como funciona o algoritmo

A aplicação é composta por dois módulos básicos: Labirinto e Solver. O módulo principal, “maze.py”, cria uma instância da classe Labirinto, utilizando um algoritmo para encontrar um caminho do início ao fim, abrindo algumas “paredes”. É isso que o construtor da classe faz.

O labirinto possui um atributo chamado “caminho”, que, no início está vazio. Aliás, vazio em Python é **None**. Ao usarmos a classe Solver, o método “solve()” utiliza a técnica de “backtracking” para encontrar um caminho do início (célula 0,0) até o fim (célula última linha, última coluna). Ao resolver o labirinto, ele modifica a propriedade “caminho” do mesmo.

Como eu fiz para imprimir o labirinto? Há um método **dunder** chamado “\_\_str\_\_”, que é invocado sempre que é necessário retornar um string a partir de um objeto. Ele é invocado no momento em que eu uso o “print()” dentro do “maze.py”:

```
print(labirinto)
```

Este método constrói uma matriz (lista de listas), ou seja um quadrado com um bloco 3 x 3 para cada célula. E eu vou colocando as paredes conforme as células. Se houver um caminho, eu inverte a cor de fundo do miolo da célula, usando a função “colored()” do pacote “termcolor”.

## Exercício

Estude o código todo, todas as classes, em especial a classe “Stack()” (por que eu tive que criar uma classe para isso?)

Veja o que pode ser otimizado.