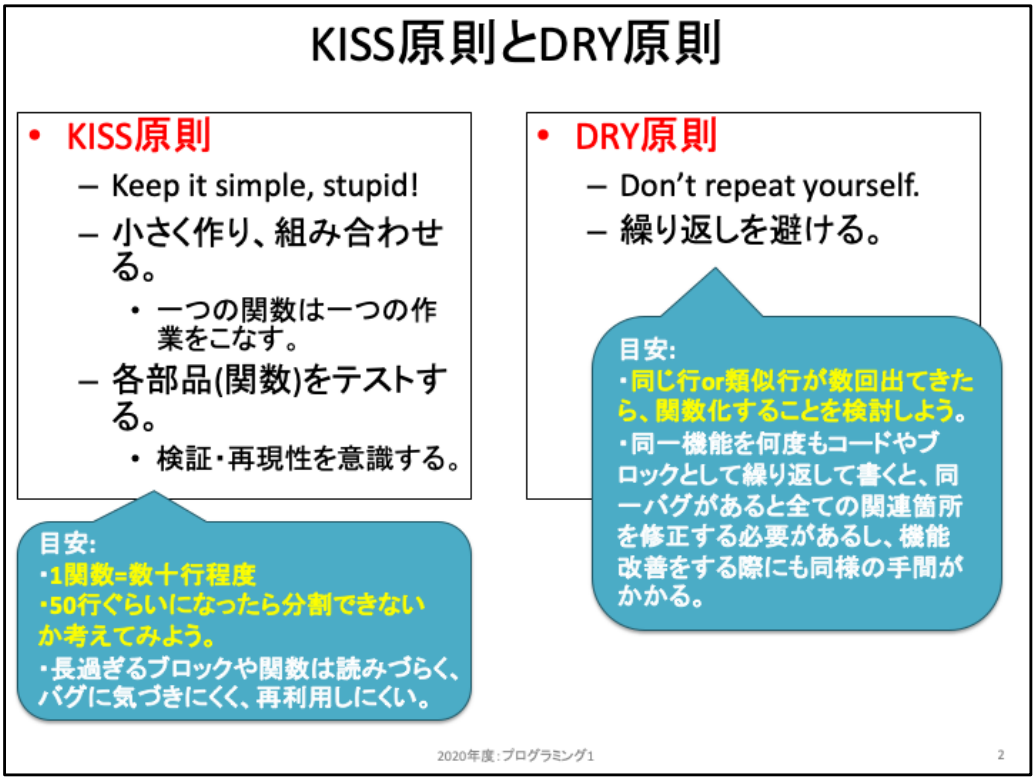
プログラミング1

(第9回) KISS原則＆DRY原則、再帰関数

1. プログラミング補足
   1. KISS原則、DRY原則
2. Chapter 4.3 Recursion
   1. 実際の動作とスタックフレームの対応
   2. 普通の反復処理 vs. 再帰呼び出し
   3. 迷路探索の例
3. Chapter 4.4 Global Variables
4. まとめ

講義ページ: http://ie.u-ryukyu.ac.jp/~tnal/2020/prog1/



まずは想定通りに動作するコードを書くことが最優先。その次のステップとして、コードの読みやすさを意識してみよう。コードは、授業課題のようなケースを除いて一般的には中長期的に使い続けることが多い。利用者は自分だけとは限らないし、自分だけであったとしても数週間前に書いたコードを思い出すのは困難であることも少なくない。これらを手助けするためにドキュメンテーションやコメントといった手段もあるが、これらに加えて「読みやすいコード（readable code）」がある。

読みやすいコードなら、どこかに潜在的なバグがありそうな場合に気づきやすく、またそうでなくともデバッグしやすいことが多い。つまり「使い続ける」ことを前提とした場合の可用性（availability）を高めることに繋がる。

読みやすいコードを厳密に定義することは困難だが、代表的な考え方としてKISS原則とDRY原則がある。KISS原則とは「とにかく一つの関数では一つの作業をこなすように小さく作れ」という指針だ。一つのことをやるだけならコード行も少なくなり、読みやすくなる。それを端的に示した指標の一つが「50行ぐらい」だ。これは多くの作業画面1つに収まる程度の行数という意味合いであり、これを超えると全体を視認することが難しくなる。50を超えそうなら、それを複数の関数に分解できないか検討してみよう。

もう一つのDRY原則とは、同じコードや類似したコードが複数回出てくるようなら、それらを整理できないか考えようという指針だ。「類似したコード」はかなり抽象的な考え方で難しいが、その一例を「再帰関数」のところで紹介する。

プログラミング1

(第9回) 命名規則、KISS原則＆DRY原則、抽象化

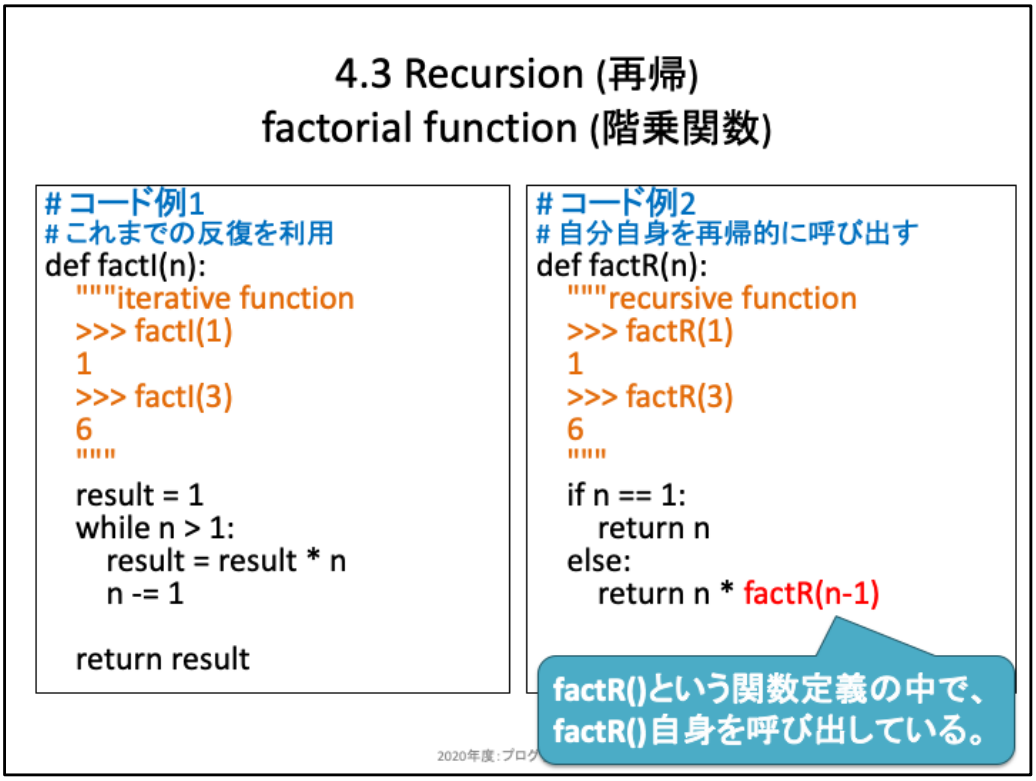
1. プログラミング補足
   1. KISS原則、DRY原則
2. Chapter 4.3 Recursion
   1. 実際の動作とスタックフレームの対応
   2. 普通の反復処理 vs. 再帰呼び出し
   3. 迷路探索の例
3. Chapter 4.4 Global Variables
4. まとめ

2大原則を意識して、**読みやすいコード**となるよう工夫してみよう。

対象を抽象的に捉えることで、実装しやすくなる。まずは**再帰関数の動作を理解できるようになろう**。

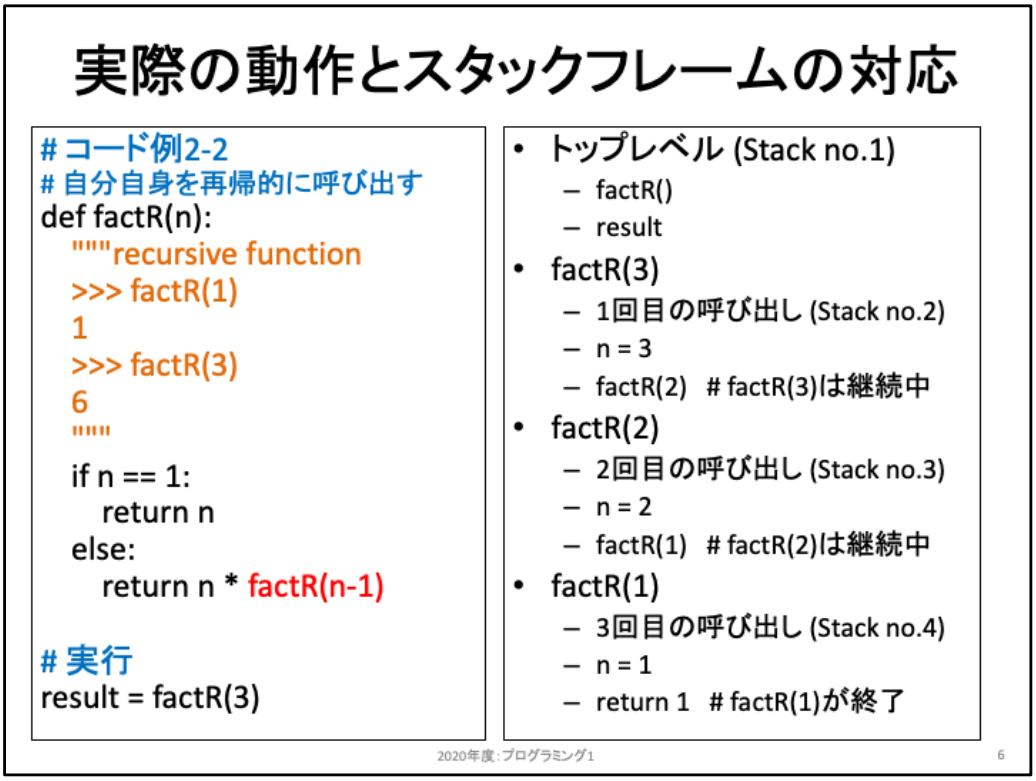
Chapter 4.3 の補足

4.3 Recursion (再帰)



階乗を計算する関数を実装してみよう。階乗関数とは、引数が1のときは1を返すだけ。引数が2のときは2*1で2を返す。引数が3のときは3*2*1で6を返す。といった動作をする。つまり、引数が4ならば 4*3*2*1 = 24 を返すような関数だ。これまでの書き方であれば、for文もしくはwhile文を用いて、引数に合わせて1より大きな整数までを全て掛け合わせるようなコードを書くだろう。その例が左側のコード factI() だ。

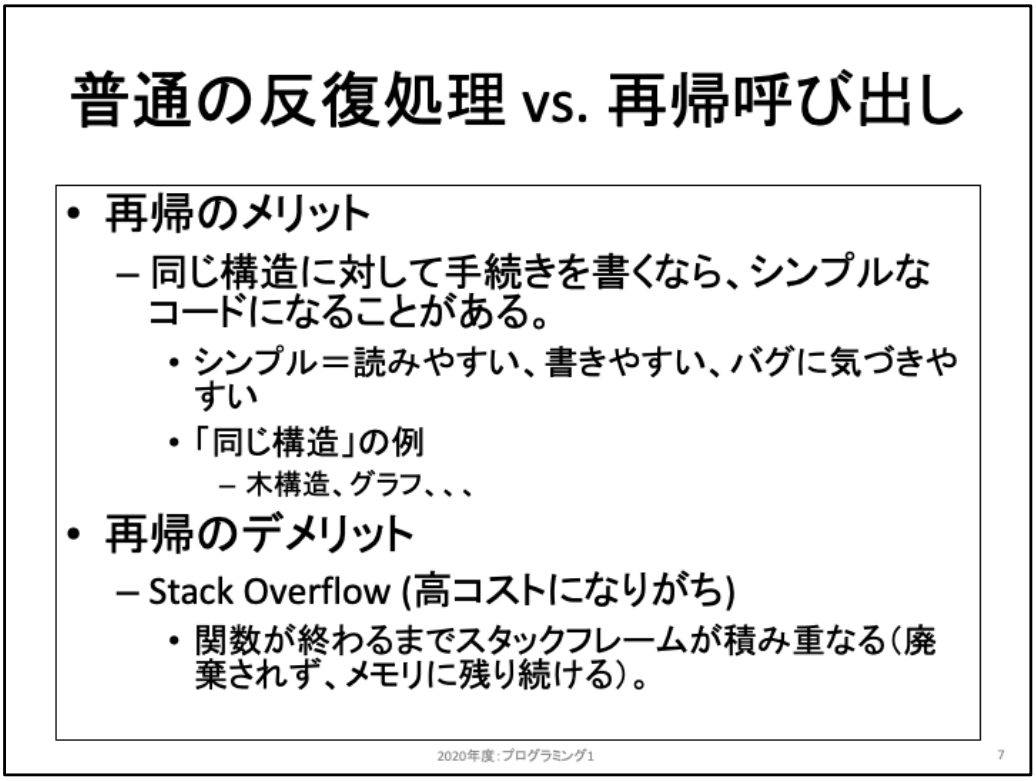
これに対し、関数の中で自分自身を再帰的に呼び出すコード例 factR() を右に示している。ここに記述しているのは条件分岐とそれに伴うreturn文のみであり、2つ目のreturn文で自分自身 factR() を呼び出している。Stack frame を考えながら解読してみよう。（次スライド）



factR 実行時のスタックフレームを書き出してみた。ここでは factR(3) を実行するものとし、この時点のスタックフレームを1番とする。つまり、
>>> factR(3)
を呼び出した時点のスタックフレームが1番である。

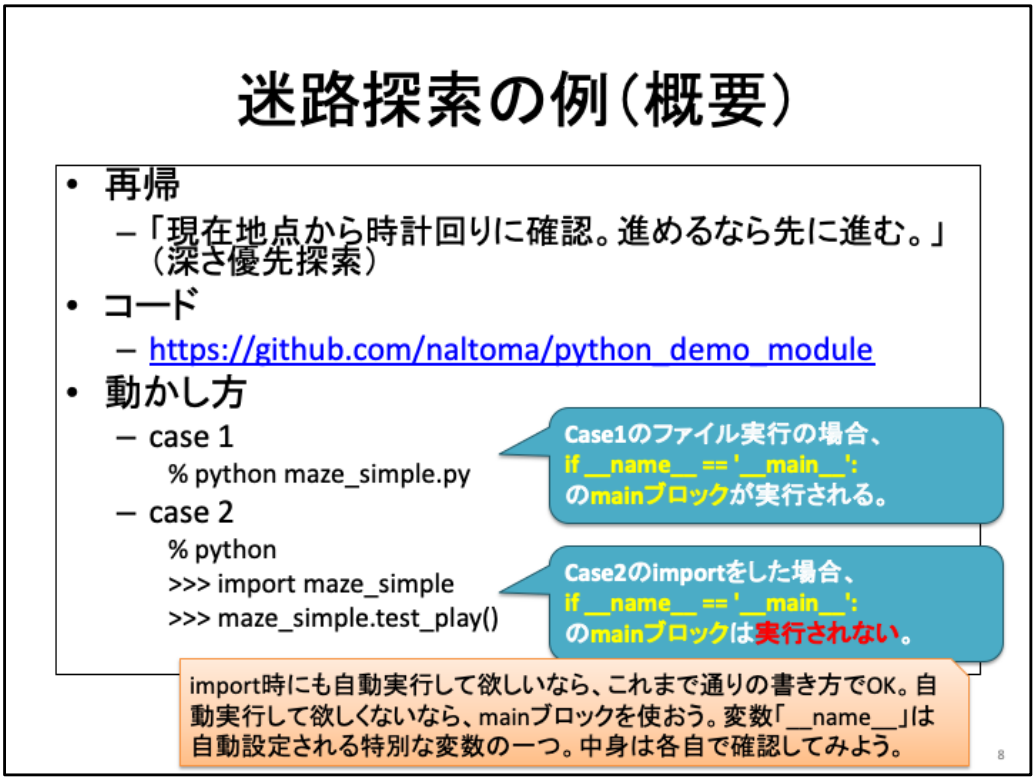
factRが呼び出されて内部に移動すると、新規にスタックフレーム2番「factR(3)」を生成する。このときの引数nは3であり、if文によりelseブロックが選択される。ここにはreturn文が記述されており「3 * factR(3-1)」、つまり「3 * factR(2)」を実行しようとする。factRは関数呼び出しのため新たにスタックフレームを呼び出してそこで処理を続ける。
factRが呼び出されて内部に移動すると、新規にスタックフレーム3番「factR(2)」を生成する。このときの引数nは2であり、if文によりelseブロックが選択される。ここにはreturn文が記述されており「2 * factR(2-1)」、つまり「2 * factR(1)」を実行しようとする。
factRが呼び出されて内部に移動すると、新規にスタックフレーム4番「factR(1)」を生成する。このときの引数nは1であり、if文によりTrueブロックが選択される。ここにはreturn文が記述されており「return 1」を実行する。ここで初めてスタックフレームの廃棄が始まり、スタックフレーム3番に戻る。
スタックフレーム3番「factR(2)」では「2 * factR(1)」の処理途中であった。factR(1)は1を返してきたため、2*1により2となり、これを返す。これによりスタックフレーム2番に戻る。
スタックフレーム2番「factR(3)」では「3 * factR(2)」の処理途中であった。factR(2)は2を返してきたため、3*2により6となり、これを返す。これによりスタックフレーム1番に戻る。
スタックフレーム1番は一番最初に「factR(3)」を呼び出した場所であり、先程の戻り値6が帰ってくる。

再帰関数の動作としては上記のとおりである。ではこれのどこが「類似したコード」なのだろうか。それは処理の手順にある。階乗関数ではnがどのぐらい大きくても「1まで遡って掛ける」という処理だ。これを「引数の値が1より大きいなら、1引いた値について同じ処理を実行しよう」という手順としてまとめたものが factR() となっている。

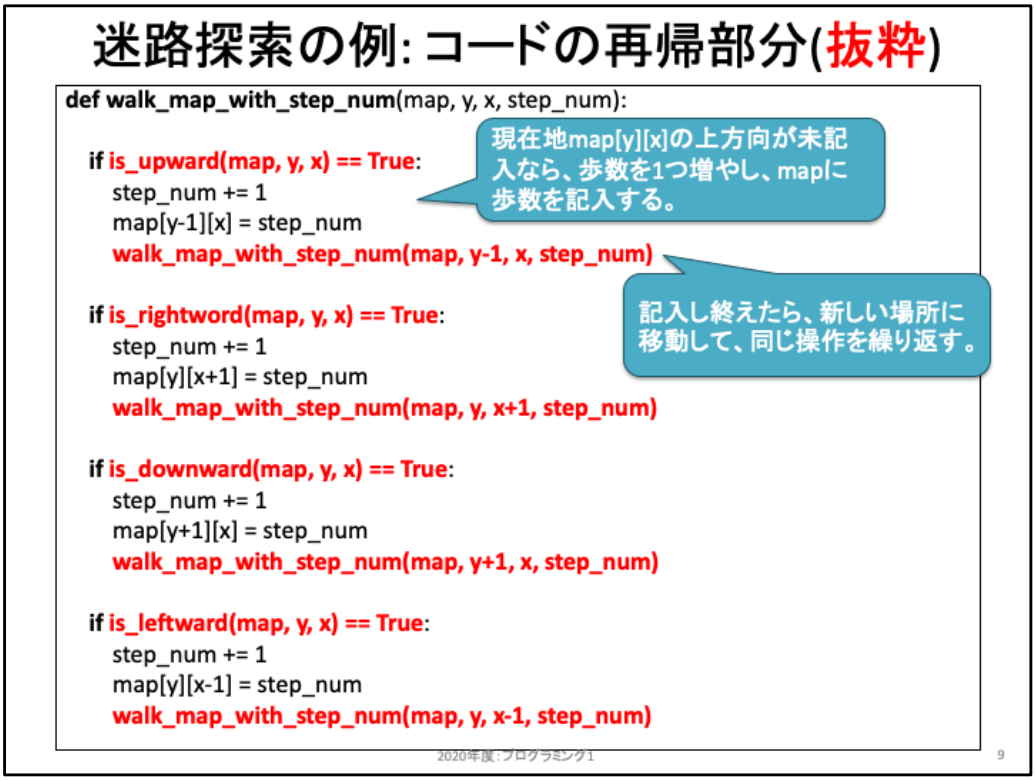


factRは、処理手順に共通点があることを利用して再帰関数として記述していた。このように「同じ構造」として処理手順を抽出できるなら、再帰関数としてシンプルに記述できることがある。シンプルに書けるということは読みやすいコードであり、潜在的なバグにも気づきやすくなる。一方で再帰関数にはデメリットもある。それはスタックフレームを積み上げている間は常にメモリを消費し続けているという点だ。あまりに膨大なスタックフレームを積み重ねると、StackOverFlowとなり、実行不可能となってしまうこともしばしばある。

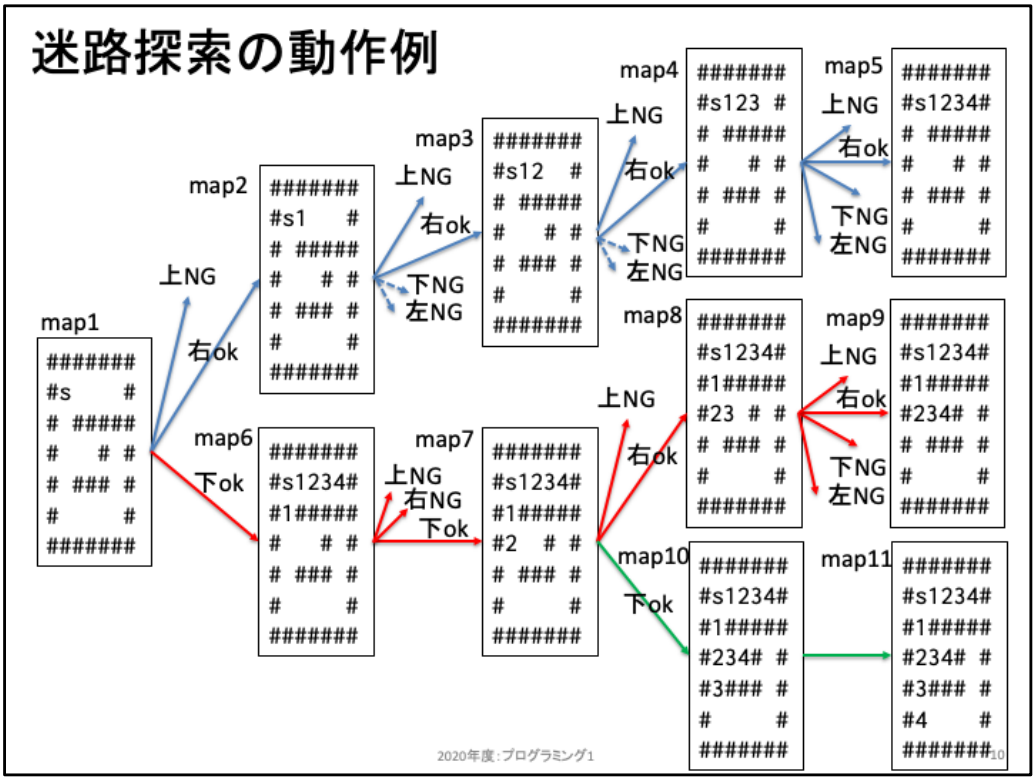
再帰関数の例をもう一つ紹介しよう。（次スライド）



maze_simple.py は、リストで用意された2次元格子空間を散策し、進むことができる空間を全て探索し尽くすためのプログラムである。探索方法にはいろいろあるが、例えば「現在地点から時計回りに確認。進めるなら先に進む。」ようにしてみよう。進める間は進み続けるため、後で戻ってきて未探索の場所を改めて探索し続ける必要がある。

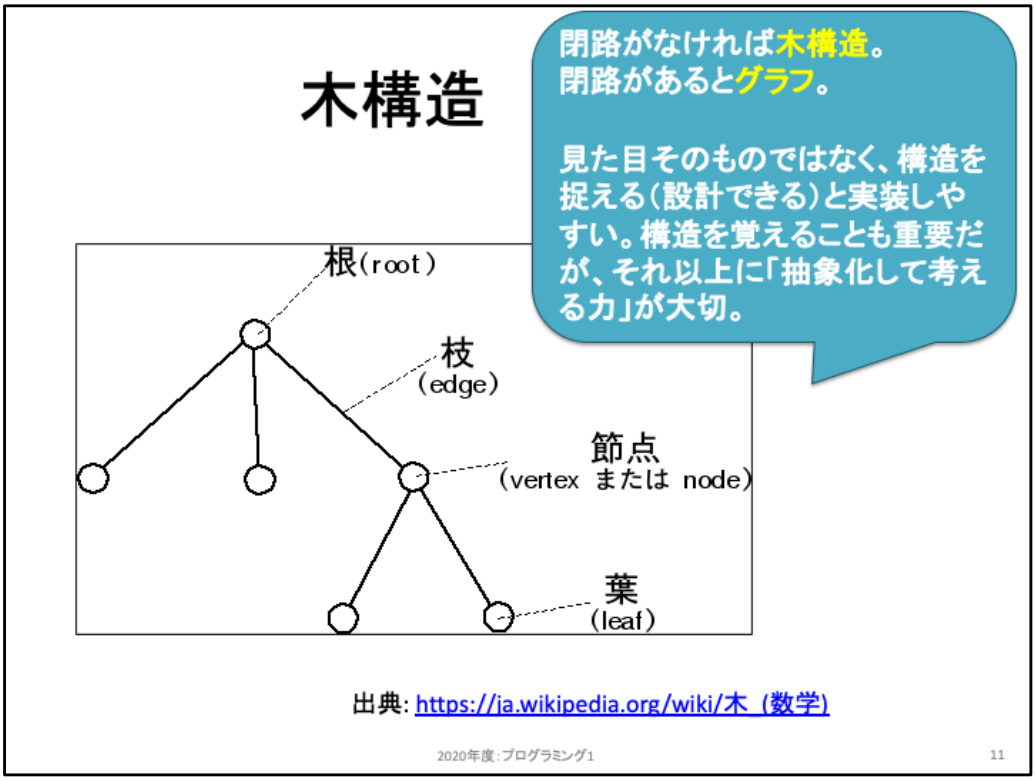


再帰関数として記述している主要部分だけを抜粋した。
walk_map_with_step_numは、現在地map[y][x]について上右下左の順に移動可能かどうかを確認している。移動可能であればそこに移動し、改めて自分自身を呼び出すことで上右下左の順に処理を行う。つまり、移動可能ならばその都度スタックフレームを新規作成することでそれまでの状態を保存しておき、その後の処理をし終えたら（スタックフレームがここまで戻ってきたら）、その後の処理を継続することになる。



Map1のsに最初いるものとする。#は移動できない場所だ。
Map1の上には行くことができない。次に右方向には移動可能なので、右に移動してから新規にスタックフレームを作成する。
Map2の上には行くことだけいない。次に右方向には移動可能なので、右に移動してから新規にスタックフレームを作成する。
中略。
Map5の状態までたどり着くと、上にも右にも下にも左にも移動することができない。（左は移動済みマーキングさされているため、移動できないと処理している）。これでmap5のスタックフレームを処理し終えたので、この状態で一つ手前（map4）に戻る。
Map4ではまだ上右の2箇所しか確認していなかったため、残り2箇所である下左を確認する。これらは移動できないため、この上阿智で一つ手前（map3）に戻る。
、、、

という具合に、全てのマスで「現在地点から時計回りに確認。進めるなら先に進む。」という処理を再帰処理として記述している。



先程の考え方は「木構造」と呼ばれる構造を踏まえた探索方法である。全てのノードを処理するためにどの順番で処理していくか。それを深さ優先（先にすすめるならそこから処理する。処理し終えたら戻ってくる）という考え方で手順を整理した。

このように「類似したコード」における抽象化はとても難しい。抽象化した考え方はプログラミング1以外の授業も含めて少しずつ学んでいくため、今はそこまでやれる必要はない。プログラミング1としては、再帰関数の考え方を理解しよう。具体的には、再帰関数が出てきたとしても動作を理解できるようになろう。より詳しく確認したい場合には、maze_simple.py等のコード例を元に、breakpointを設定してデバッグ実行するといいだろう。

プログラミング1

(第9回) 命名規則、KISS原則＆DRY原則、抽象化

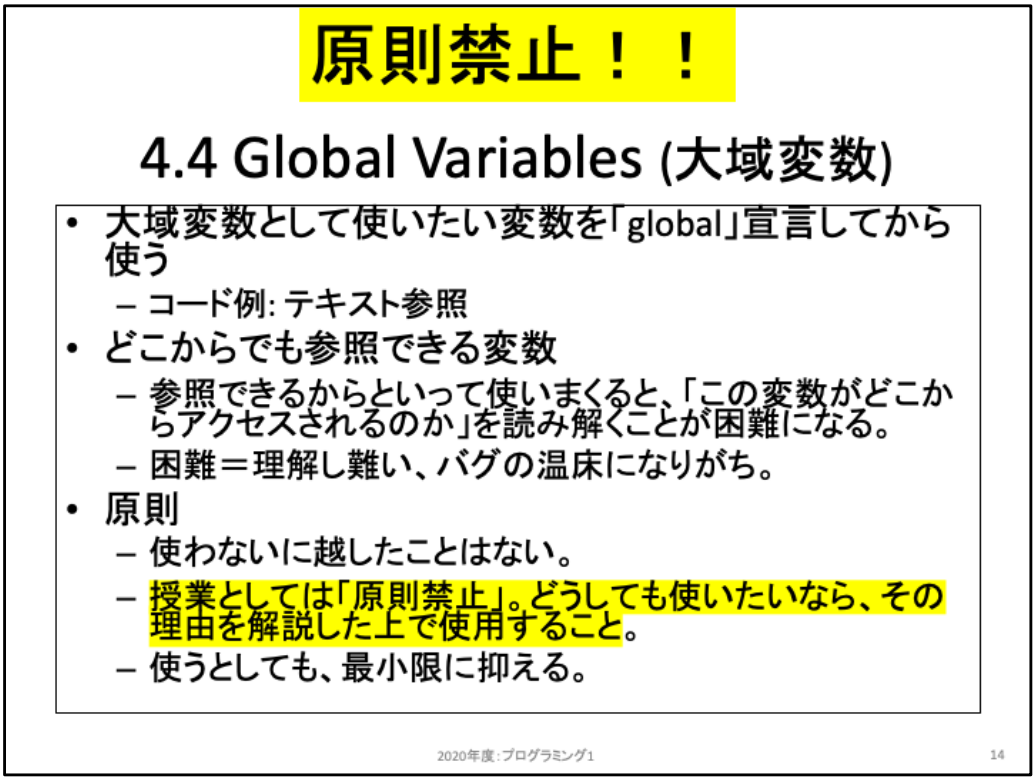
1. プログラミング補足
   1. KISS原則、DRY原則
2. Chapter 4.3 Recursion
   1. 実際の動作とスタックフレームの対応
   2. 普通の反復処理 vs. 再帰呼び出し
   3. 迷路探索の例
3. Chapter 4.4 Global Variables
4. まとめ

2大原則を意識して、**読みやすいコード**となるよう工夫してみよう。

対象を抽象的に捉えることで、実装しやすくなる。まずは**再帰関数の動作を理解できるようになろう**。

Chapter 4.4 の補足

4.4 Global Variables (大域変数)



特別な理由がなければ使ってはならない。以上。

プログラミング1

(第9回) 命名規則、KISS原則＆DRY原則、抽象化

1. プログラミング補足
   1. 変数名・ファイル名の命名規則
   2. KISS原則、DRY原則
2. Chapter 4.3 Recursion
   1. 実際の動作とスタックフレームの対応
   2. 普通の反復処理 vs. 再帰呼び出し
   3. 迷路探索の例
3. Chapter 4.4 Global Variables
4. まとめ
5. デモ？（時間あれば）

命名規則＆2大原則を意識して、**読みやすいコード**となるよう工夫してみよう。

対象を抽象的に捉えることで、実装しやすくなる。まずは**再帰関数の動作を理解できるようになろう。**

通常は大域変数を使う必要はありません。**使わないで下さい。**

まとめ

- KISS原則とDRY原則
  - 1関数はシンプル（単機能）にし、機能をテストする。orテスト駆動開発。
  - 繰り返してるコードがあれば、繰り返さずに書けないか考えてみる。
- 再帰
  - 関数自身を繰り返し呼ぶことで処理しやすいケースがある。
- 木構造・グラフ構造
  - 「基本構造」はツール。その構造に落とし込む形でモデル化できると、想定漏れを防ぎやすい（コードに落としやすい）。
- 大域変数
  - 原則禁止

モデル化って何だろう？
＊学習教育目標にあり。

デモ？（パースしてみる）

```
#name,HP
"slime":5
"goblin":10
```

- 読み込んだ行の冒頭が#ならば、処理対象外とする。
- 読み込んだ行の冒頭が"ならば、「"敵の名前":数字」という書式で、名前とHPが記述されているとする。
- 「"slime":5」を {'slime':5} としてdict型オブジェクトとして保存したい。