プログラミング1

(第7回) KISS原則とDRY原則、doctest補足、再帰と木構造

1. KISS原則、DRY原則
2. doctest補足
    1. doctestの書き方
    2. 何のためにユニットテストを書くのか？
3. 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. 木構造
    4. 迷路探索の例
4. Chapter 4.4 Global Variables
5. まとめ
6. 演習
    1. 演習7: doctest, while文
7. 宿題

講義ページ: http://ie.u-ryukyu.ac.jp/~tnal/2016/prog1/

関数や仕様はどう決定したら良いか？

授業計画:
第1回～第8回

- 代表的な原則
- KISS原則
  - Keep it simple, stupid!
  - 小さく作り、組み合わせる。
    - 一つの関数は一つの作業をこなす。
  - 各部品(関数)をテストする。
    - 検証・再現性を意識する。
- DRY原則
  - Don’t repeat yourself.
  - 繰り返しを避ける。

授業計画:
第2回～

- 経験を積む
  - 様々な問題にトライする。
  - ペア・プログラミング。
    - 互いに教え合う
    - 知識の共有化
- (マルチステークホルダーの存在を意識する)

目安:
・1関数=数10行程度
・50行ぐらいになったら分割できないか考えてみよう。
・長過ぎるブロックは読みづらく、バグに気づきにくく、再利用しにくい。

doctest補足

- ユニットテストの書き方
  - docstring形式ドキュメント中に、インタプリタの出力を含めて実行例と結果例を記入する。
  - インデントを揃えること。
  - スペースの有無に注意。
- テストしたい関数の例

```
def add(x, y):
    result = x + y
    return result
```

- テストを含めた例

```
def add(x, y):
    """
    >>> add(1, 2)
    3
    """
    result = x + y
    return result

if __name__ == "__main__":
    import doctest
    doctest.testmod()
```

テストの実装

-vオプション付けて実行した際にテストを実行するという宣言

何のためにユニットテストを書くのか？

- 関数が想定通りに動くことを「検証しやすくする」ため。
  - 毎回手動確認するのは辛い。
  - 「想定通り」の質が確認者に強く依存する。
  - 「動作例」を記述することで、想定している使い方を示すことにもなる。
- リファクタリング
  - 機能を保ったまま、コードを修正。

- テスト駆動開発
  - 背景: 開発現場でよくある状況
    - 「こういう入力が与えられた時に、こういう出力をする関数を書いて」
    - 入力と出力は分かっている。or 分からないなら、コードを書き始める前に明確にする。
  - テストを先に書くことで、その関数をどう動作させたいかを明確にする。
  - 書いたテストが通るようにコードを書く。

参考:
エンジニアのスキルを伸ばす「テスト駆動開発」を学んでみよう:
https://thinkit.co.jp/story/2014/07/30/5097

Chapter 4.3 の補足

4.3 Recursion

4.3 Recursion (再帰)
p.45, factorial function (階乗関数)

```
# コード例1
# これまでの反復を利用
def factI(n):
    """iterative function
    >>> factI(1)
    1
    >>> factI(3)
    6
    """
    result = 1
    while n > 1:
        result = result * n
        n -= 1

    return result
```

```
# コード例2
# 自分自身を再帰的に呼び出す
def factR(n):
    """recursive function
    >>> factR(1)
    1
    >>> factR(3)
    6
    """
    if n == 1:
        return n
    else:
        return n * factR(n-1)
```

factR()という関数定義の中で、factR()自身を呼び出している。

実際の動作とスタックフレームの対応

```
# コード例2-2
# 自分自身を再帰的に呼び出す
def factR(n):
    """recursive function
    >>> factR(1)
    1
    >>> factR(3)
    6
    """
    if n == 1:
        return n
    else:
        return n * factR(n-1)

# 実行
result = factR(3)
```

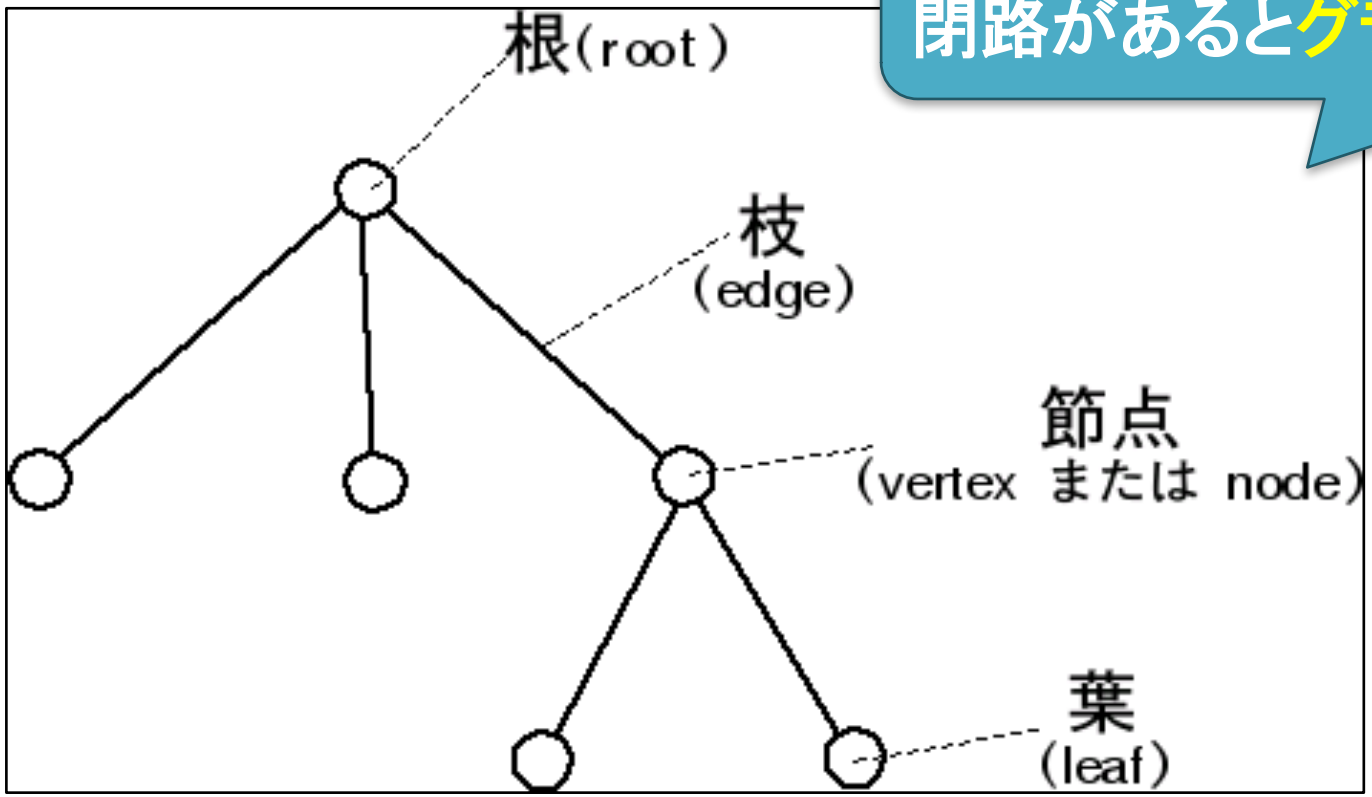
- トップレベル (Stack no.1)
  - factR()
  - result
- factR(3)
  - 1回目の呼び出し (Stack no.2)
  - n = 3
  - factR(2)　＊factR(3)は継続中
- factR(2)
  - 2回目の呼び出し (Stack no.3)
  - n = 2
  - factR(1)　＊factR(2)は継続中
- factR(1)
  - 3回目の呼び出し (Stack no.4)
  - n = 1
  - return 1　＊factR(1)が終了

普通の反復処理 vs. 再帰呼び出し

- 再帰のメリット
  - 同じ構造に対して手続きを書くなら、シンプルなコードになることがある。
    - シンプル＝読みやすい、書きやすい、バグに気づきやすい
    - 「同じ構造」の例
      - 木構造、グラフ、、、
- 再帰のデメリット
  - Stack Overflow (高コストになりがち)
    - 関数が終わるまでスタックフレームが積み重なる（廃棄されず、メモリに残り続ける）。

木構造

閉路がなければ**木構造**。
閉路があると**グラフ**。



出典: https://ja.wikipedia.org/wiki/木_(数学)

迷路探索の例（概要）

- 再帰
  - 「現在地点から時計回りに確認。進めるなら先に進む。」（深さ優先探索）
- コード
  - http://ie.u-ryukyu.ac.jp/~tnal/2016/prog1/maze_simple.py
- 動かし方
  - case 1

    ```
    % python3 maze_simple.py
    ```
  - case 2

    ```
    % python3
    >>> import maze_simple
    >>> maze_simple.test_play()
    ```

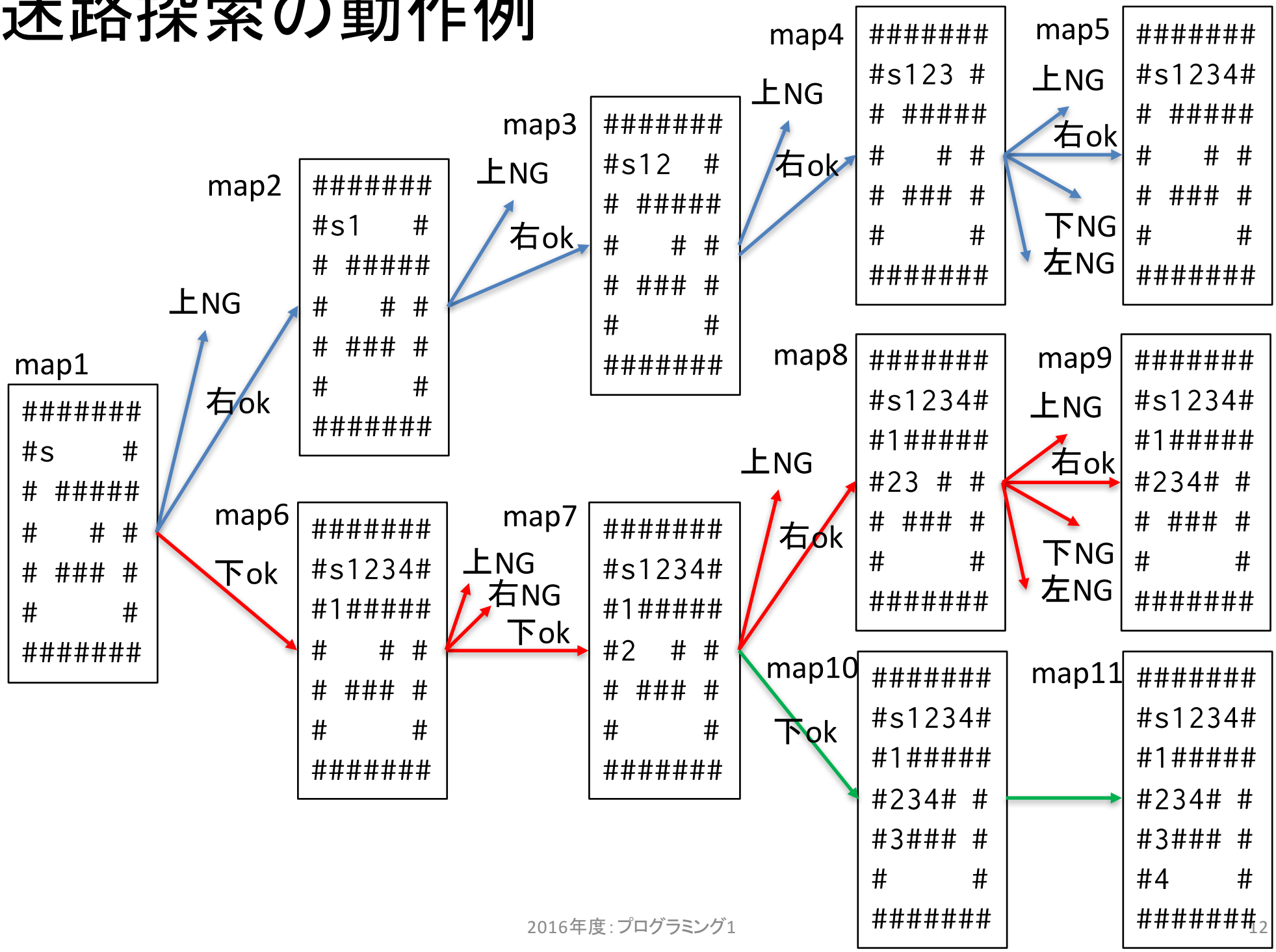
迷路探索の例:コードの再帰部分(抜粋)

```
def walk_map_with_step_num(map,  y, x, step_num):

    if is_uppper(map,  y, x) == True:
        step_num  += 1
        map[y-1][x] = step_num
        walk_map_with_step_num(map,  y-1, x, step_num)

    if is_forword(map,  y, x) == True:
        step_num  += 1
        map[y][x+1] = step_num
        walk_map_with_step_num(map,  y, x+1, step_num)

    if is_lower(map, y, x) == True:
        step_num  += 1
        map[y+1][x] = step_num
        walk_map_with_step_num(map,  y+1, x, step_num)

    if is_backword(map,  y, x) == True:
        step_num  += 1
        map[y][x-1] = step_num
        walk_map_with_step_num(map,  y, x-1, step_num)
```

現在地map[y][x]の上方向が未記入なら、歩数を1つ増やし、mapに歩数を記入する。

記入し終えたら、新しい場所に移動して、同じ操作を繰り返す。

迷路探索の動作例


map1
#######
#s   #
#####
#   # #
#
#   #
#######
上NG
右ok
map2
#######
#s1   #
#####
#   # #
#
#   #
#######
上NG
右ok
map3
#######
#s12 #
#####
#   # #
#
#   #
#######
上NG
右ok
map4
#######
#s123 #
#####
#   # #
#
#   #
#######
上NG
右ok
下NG
左NG
map5
#######
#s1234#
#####
#   # #
#
#   #
#######
下ok
map6
#######
#s1234#
#1#####
#   # #
#
#   #
#######
上NG
右NG
下ok
map7
#######
#s1234#
#1#####
#2 # #
#
#   #
#######
上NG
右ok
map8
#######
#s1234#
#1#####
#23 # #
#
#   #
#######
上NG
右ok
下NG
左NG
map9
#######
#s1234#
#1#####
#234# #
#
#   #
#######
下ok
map10
#######
#s1234#
#1#####
#234# #
#3### #
#   #
#######
map11
#######
#s1234#
#1#####
#234# #
#3### #
#4   #
#######

Chapter 4.4 の補足

4.4 Global Variables

4.4 Global Variables (大域変数)

- 大域変数として使いたい変数を「global」宣言してから使う
  - コード例: テキスト参照
- どこからでも参照できる変数
  - 参照できるからといって使いまくると、「この変数がどこからアクセスされるのか」を読み解くことが困難になる。
  - 困難＝理解し難い、バグの温床になりがち。
- 原則
  - 使わないに越したことはない。
  - 使うとしても、最小限に抑える。
  - 授業としては「原則禁止」。どうしても使いたいなら、その理由を解説した上で使用すること。

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。
- ユニットテスト
  - 動作検証の自動化、テスト駆動開発、リファクタリング。
- 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。

演習

演習6をベースに修正しました。

演習7: doctest, while文の利用

補足1

- ペアプログラミングを始める前に
  - 記入漏れ
    - 「実施日」と「報告者」
  - 意思疎通しながら取り組む（独立して作業しない）
    - driver: メイン作業者。
    - observer: 観察者。
  - 前回の復習確認
    - 「何をやったっけ？」
    - 「これはこうやれば良いんだっけ？」

補足2

- ペアプログラミングのやり方
  - 7ステップ
    - 作業を決める
    - 最初の目標を決める
    - パートナーを頼りにし、支えてやる
      - driver: 仕事を終わらせることに専念
      - observer: 横から観察し、疑問・改善・簡潔化など大局的な問題について考える
    - 喋る
      - 「一人で悩む」のは十秒程度に留める
      - 一緒に相談しながら考える練習
    - お互い何をやっているか把握する
      - 頻繁に同期をとる
    - 喜ぶ
    - 交代する

分業ではない（observer=観察しながら気づいたことをコメント、分からないことを質問）

二人で2,3分考えても分からない場合には、手を上げて質問しよう

演習

- 演習7: while文
  - https://ie.u-ryukyu.ac.jp/~tnal/2016/prog1/week7-ex.html
- ペア・プログラミング
  - https://ie.u-ryukyu.ac.jp/~tnal/2016/prog1/week2-pair-programming.html
  - driver, observer (navigator)

宿題

- 復習: 適宜(これまでの内容)
  - レポート課題2「コード読解」　＊講義ページ参照。
    - レポート課題1の良レポートの「良さ」を真似よう。
- 予習: 教科書読み
  - 5章
    - 5.1 Tuples
    - (5.2 Lists and Mutabilities)　＊余裕があれば。
- 復習・予習(オススメ): paiza
  - プログラミングスキルチェック　＊レベル設定のある課題集
    - https://paiza.jp/challenges/info

参考文献

- 教科書: Introduction to Computation and Programming Using Python, Revised And Expanded Edition
- Python 3.5.1 documentation, https://docs.python.org/3.5/index.html
- doctest — Test interactive Python examples, https://docs.python.org/3/library/doctest.html
- エンジニアのスキルを伸ばす「テスト駆動開発」を学んでみよう, https://thinkit.co.jp/story/2014/07/30/5097