**第二回 変数のスコープと再帰**

* 実験年月日 2018/05/28
* 提出年月日 2018/05/28
* 班番号 6
* 報告者 3年19番6班 末田 貴一
* 共同実験者
  + 7番 川上 求
  + 42番 山崎 敦史
  + 47番 ロンサン

**問題2.1**

**ソースコード**

サンプルのエラーが出るコード

x=2

a=5

def f(x):

x=x+a

y=x+y

return y

y=a

z=f(4)

print('x=',x)

print('y=',y)

print('z=',z)

修正後のコード

x=2

a=5

def f(x):

x=x+a

b=x+y

return b

y=a

z=f(4)

print('x=',x)

print('y=',y)

print('z=',z)

**実行結果**

修正前の実行結果

Traceback (most recent call last):

File "2\_1.py", line 9, in <module>

z=f(4)

File "2\_1.py", line 5, in f

y=x+y

UnboundLocalError: local variable 'y' referenced before assignment

修正後の実行結果

x= 2

y= 5

z= 14

**なぜエラーが起きたのか，どのように修正したのか**

修正前の関数f内にて使用されているyはローカル変数として使っているつもりのサンプルコードでした。  
しかしグローバル変数としてyは使っているので別名のbというローカル変数を用意することで簡単にエラーを解消できる。

**問題 2.2**

**ソースコードと実行結果**

図 2.4

def f():

print(x)

x=1

x=100

f()

print(x)

実行結果

Traceback (most recent call last):

File "2\_2.py", line 5, in <module>

f()

File "2\_2.py", line 2, in f

print(x)

UnboundLocalError: local variable 'x' referenced before assignment

先程と同様の理由でエラーが出た。

図 2.5

def f():

global x

print(x)

x=1

x=100

f()

print(x)

実行結果

100

1

fという関数の中で用いる変数xをglobalというキーワードをつけることでグローバル変数として認識させた。  
なので最初は100と表示され，つぎは関数内で書き換えられた1と表示された。

**問題 2.3**

**ソースコード**

def factorial\_rec(n):

#nが1以上のint型と家庭してnの階乗を返す

if n==1:

return 1

else:

return n \* factorial\_rec(n-1)

for i in range(10):

key\_input=input("int型のn(1<=n<=10)を指定して下さい：")

print(factorial\_rec(int(key\_input)))

**実行結果**

>python 2\_3.py

int型のn(1<=n<=10)を指定して下さい：1

1

int型のn(1<=n<=10)を指定して下さい：2

2

int型のn(1<=n<=10)を指定して下さい：3

6

int型のn(1<=n<=10)を指定して下さい：4

24

int型のn(1<=n<=10)を指定して下さい：5

120

int型のn(1<=n<=10)を指定して下さい：6

720

int型のn(1<=n<=10)を指定して下さい：7

5040

int型のn(1<=n<=10)を指定して下さい：8

40320

int型のn(1<=n<=10)を指定して下さい：9

362880

int型のn(1<=n<=10)を指定して下さい：10

3628800

**表**

| **n** | **階乗** |
| --- | --- |
| 1 | 1 |
| 2 | 2 |
| 3 | 6 |
| 4 | 24 |
| 5 | 120 |
| 6 | 720 |
| 7 | 5040 |
| 8 | 40320 |
| 9 | 362880 |
| 10 | 3628800 |

**問題 2.4**

**ソースコード**

def fibonacci\_rec(n):

#nが0位上のint型と過程してnのフィボナッチ数を返す

if n<=1:

return 1

else:

return fibonacci\_rec(n-1)+fibonacci\_rec(n-2)

for i in range(31):

key\_input=input("int型のn(0<=n<=30)を指定して下さい：")

print(fibonacci\_rec(int(key\_input)))

**実行結果**

>python 2\_4.py

int型のn(0<=n<=30)を指定して下さい：0

1

int型のn(0<=n<=30)を指定して下さい：1

1

int型のn(0<=n<=30)を指定して下さい：2

2

int型のn(0<=n<=30)を指定して下さい：3

3

int型のn(0<=n<=30)を指定して下さい：4

5

int型のn(0<=n<=30)を指定して下さい：5

8

int型のn(0<=n<=30)を指定して下さい：6

13

int型のn(0<=n<=30)を指定して下さい：7

21

int型のn(0<=n<=30)を指定して下さい：8

34

int型のn(0<=n<=30)を指定して下さい：9

55

int型のn(0<=n<=30)を指定して下さい：10

89

int型のn(0<=n<=30)を指定して下さい：11

144

int型のn(0<=n<=30)を指定して下さい：12

233

int型のn(0<=n<=30)を指定して下さい：13

377

int型のn(0<=n<=30)を指定して下さい：14

610

int型のn(0<=n<=30)を指定して下さい：15

987

int型のn(0<=n<=30)を指定して下さい：16

1597

int型のn(0<=n<=30)を指定して下さい：17

2584

int型のn(0<=n<=30)を指定して下さい：18

4181

int型のn(0<=n<=30)を指定して下さい：19

6765

int型のn(0<=n<=30)を指定して下さい：20

10946

int型のn(0<=n<=30)を指定して下さい：21

17711

int型のn(0<=n<=30)を指定して下さい：22

28657

int型のn(0<=n<=30)を指定して下さい：23

46368

int型のn(0<=n<=30)を指定して下さい：24

75025

int型のn(0<=n<=30)を指定して下さい：25

121393

int型のn(0<=n<=30)を指定して下さい：26

196418

int型のn(0<=n<=30)を指定して下さい：27

317811

int型のn(0<=n<=30)を指定して下さい：28

514229

int型のn(0<=n<=30)を指定して下さい：29

832040

int型のn(0<=n<=30)を指定して下さい：30

1346269

**表**

| **n** | **フィボナッチ数** |
| --- | --- |
| 0 | 1 |
| 1 | 1 |
| 2 | 2 |
| 3 | 3 |
| 4 | 5 |
| 5 | 8 |
| 6 | 13 |
| 7 | 21 |
| 8 | 34 |
| 9 | 55 |
| 10 | 89 |
| 11 | 144 |
| 12 | 233 |
| 13 | 377 |
| 14 | 610 |
| 15 | 987 |
| 16 | 1597 |
| 17 | 2684 |
| 18 | 4181 |
| 19 | 6765 |
| 20 | 10946 |
| 21 | 17711 |
| 22 | 28657 |
| 23 | 46368 |
| 24 | 75025 |
| 25 | 121393 |
| 26 | 196418 |
| 27 | 317811 |
| 28 | 514299 |
| 29 | 832040 |
| 30 | 1346269 |

**問題2.5**

ソースコード

QUEEN = 9

EMPTY = 0

def print\_grids(grids):

#盤面の2次元配列gridsを表示する

for row in grids:

print(row)

print()

def is\_free\_anti\_diagonal(grids,y,x):

#盤面gridsの座標(y,x)を基準として反対書く方向にクイーンが配置されていないならTrueを返す

for i in range(8+1):

if(y-i>=0 and x+i<8+1):

if(grids[y-i][x+i]!=0):

return False

if(y+i<8+1 and x-i>=0):

if(grids[y+1][x-i]!=0):

return False

return True

def is\_free\_diagonal(grids,y,x):

#盤面gridsの座標(x,y)を基準として対角方向にクイーンが配置されていないならTrueを返す

for i in range(8+1):

if(y-i>=0 and x-i>=0):

if(grids[y-i][x-i]!=0):

return False

if(y+i<8+1 and x+i<8+1):

if(grids[y+i][x+i]!=0):

return False

return True

def is\_free\_column(grids,x):

#盤面gridsのx座標を基準として列方向にクイーンが配置されていないならTrueを返す

for j in range(8+1):

if(grids[j][x]!=EMPTY):

return False

return True

def is\_free(grids,y,x):

#盤面gridsの座標(x,y)にクイーンが配置できるならTrueを返す

if(is\_free\_column(grids,x) and is\_free\_diagonal(grids,y,x) and is\_free\_anti\_diagonal(grids,y,x)):

return True

else:

return False

i=0

def find\_eight\_queen(grids,y):

global i

#盤面gridsのy座標を基準として行方向にクイーンが配置できる場所を探す

for x in range (8):

if(is\_free(grids,y,x)):

grids[y][x]=QUEEN

if(y==(8-1)):

i=i+1

# print('{}枚目'.format(i))

# print\_grids(grids)

else:

find\_eight\_queen(grids,y+1)

grids[y][x]=EMPTY

return

def create\_grids(n):

#n\*nサイズの初期盤面を返す

grids=[]

for \_ in range(n):

column=[]

for \_ in range(n):

column.append(0)

grids.append(column)

return grids

if \_\_name\_\_ == '\_\_main\_\_':

grids = create\_grids(8+1)

find\_eight\_queen(grids,0)

print('正答盤面の総数：{}'.format(i))

**実行結果**

>python 2\_5.py

正答盤面の総数：92

**問題2.6**

**ソースコード**

QUEEN = 9

EMPTY = 0

def print\_grids(grids):

#盤面の2次元配列gridsを表示する

for row in grids:

print(row)

print()

def is\_free\_anti\_diagonal(grids,y,x):

#盤面gridsの座標(y,x)を基準として反対書く方向にクイーンが配置されていないならTrueを返す

for i in range(8):

if(y-i>=0 and x+i<8):

if(grids[y-i][x+i]!=0):

return False

if(y+i<8 and x-i>=0):

if(grids[y+1][x-i]!=0):

return False

return True

def is\_free\_diagonal(grids,y,x):

#盤面gridsの座標(x,y)を基準として対角方向にクイーンが配置されていないならTrueを返す

for i in range(8):

if(y-i>=0 and x-i>=0):

if(grids[y-i][x-i]!=0):

return False

if(y+i<8 and x+i<8):

if(grids[y+i][x+i]!=0):

return False

return True

def is\_free\_column(grids,x):

#盤面gridsのx座標を基準として列方向にクイーンが配置されていないならTrueを返す

for j in range(8):

if(grids[j][x]!=EMPTY):

return False

return True

def is\_free(grids,y,x):

#盤面gridsの座標(x,y)にクイーンが配置できるならTrueを返す

if(is\_free\_column(grids,x) and is\_free\_diagonal(grids,y,x) and is\_free\_anti\_diagonal(grids,y,x)):

return True

else:

return False

i=0

def find\_eight\_queen(grids,y):

global i

global key\_input

#盤面gridsのy座標を基準として行方向にクイーンが配置できる場所を探す

for x in range (key\_input-1):

if(is\_free(grids,y,x)):

grids[y][x]=QUEEN

if(y==(key\_input-2)):

i=i+1

#print('{}枚目'.format(i))

#print\_grids(grids)

else:

find\_eight\_queen(grids,y+1)

grids[y][x]=EMPTY

return

def create\_grids(n):

#n\*nサイズの初期盤面を返す

grids=[]

for \_ in range(n):

column=[]

for \_ in range(n):

column.append(0)

grids.append(column)

return grids

key\_input=int(input('N(4<=N<=13):'))+1

if \_\_name\_\_ == '\_\_main\_\_':

grids = create\_grids(key\_input)

find\_eight\_queen(grids,0)

print('正答盤面の総数：{}'.format(i))

**実行結果**

>python 2\_6.py

N(4<=N<=13):4

正答盤面の総数：2

>python 2\_6.py

N(4<=N<=13):5

正答盤面の総数：10

>python 2\_6.py

N(4<=N<=13):6

正答盤面の総数：4

>python 2\_6.py

N(4<=N<=13):7

正答盤面の総数：40

>python 2\_6.py

N(4<=N<=13):8

正答盤面の総数：92

>python 2\_6.py

N(4<=N<=13):9

正答盤面の総数：352

>python 2\_6.py

N(4<=N<=13):10

正答盤面の総数：724

>python 2\_6.py

N(4<=N<=13):11

正答盤面の総数：2680

>python 2\_6.py

N(4<=N<=13):12

正答盤面の総数：14200

>python 2\_6.py

N(4<=N<=13):12

正答盤面の総数：14200

>python 2\_6.py

N(4<=N<=13):13

正答盤面の総数：73712

**表**

| **N** | **正答盤面の総数** |
| --- | --- |
| 4 | 2 |
| 5 | 10 |
| 6 | 4 |
| 7 | 40 |
| 8 | 92 |
| 9 | 352 |
| 10 | 724 |
| 11 | 2680 |
| 12 | 14200 |
| 13 | 73712 |

**レポート課題 2.1**

females(5)で計算すると  
females(5)=females(4)+females(3)  
となる

females(4)=females(3)+females(2)  
であり，  
females(3)=females(2)+females(1)  
であり，  
female(2)=females(1)+females(0)  
であり，  
females(1)=females(0)=1  
と言える。

また，  
females(3)==females(2)+females(1)  
であり，  
female(2)=females(1)+females(0)  
であり，  
females(1)=females(0)=1  
と言える。

female(2)は5回計算されている。

**レポート課題 2.2**

フィボナッチ数列の各数を一辺とする正方形を利用した対数らせんや黄金比を見かけるように，自然界にも同様の原理を応用した様子がみられる。  
有名なものでいうとひまわりの種の配置はフィボナッチ数を利用して次の種の位置を137.5度と決まっている。  
またオウムガイも有名で體が大きくなってもバランスが崩れないように同様の設計がされている。

**レポート課題 2.3**

フローチャート

**レポート課題 2.4**

**メリット**

コードがシンプルになることがあるのでリーダブルなコードになる可能性がある。  
ネストが深くなりにくいためリーダブルなコードが書きやすい

**デメリット**

メモリの使用量が多いのでオーバーフローする可能性がある。