復習

プログラミング1

(第2回) Pythonインタプリタとスクリプトの体験1，ペア・プログラミングの導入

1. Chapter 1 の補足1
   1. Calculations and Remembers
   2. Computational thinking
2. Chapter 2 -- 2.1.2までの補足
   1. Glossaries, 用語集1, 2, 3
   2. Reserved words, 予約語
3. 文字列結合の例
4. スクリプトの利用
   1. スクリプトとは？
   2. スクリプトを書いて動かしてみよう
   3. スクリプト vs.インタプリタ
5. 変数名・ファイル名の命名規則
6. マニュアルの参照
7. 演習
8. 宿題

欲しい出力を得るためのレシピを考える必要がある。レシピ≒アルゴリズム。

レシピを記述するための道具（基本的な型・算術演算子・比較演算子・論理演算子）を使えるようになろう。

基本演算とstr.format書式を読めるようになろう。

インタプリタ実行とファイル実行を使い分けよう。

慣習を守ることで「他人が読みやすいコード（readable code）」になる。

help()やオンラインマニュアルを活用しよう。

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

プログラミング1

(第3回) インタプリタとスクリプトの体験2: 文字列とif文, 関数の利用

1. Chapter 2.2, 3.1, 4.1.1の補足
   - 2.2 Branching Programs (条件分岐)
   - 3.1 Looping (繰り返し処理)
   - 4.1.1 Function Definitions (関数定義)
   - Reserved words, 予約語
2. ペアプロ演習
3. 宿題

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

Branching Programs (条件分岐)とは？

- これまでのプログラム
  - インタプリタやファイルに書かれた命令文を、上から下に向かって順序よく実行する。
  - これだけだと、以下のような処理を書けない。
    - ゲームアプリで、レビュー書いたユーザに対して特別報酬を与えたい。
    - Webサービスで、アカウントを作成してログインしたユーザ向けに専用ページを表示したい。

- 条件分岐の考え方
  1. ある条件を満足しているか否かを確認する。(条件判定し、True/Falseいずれなのかを確認する。
  2. 判定結果に基づき、True時の処理(True block)、False時の処理(False block)を分けて記載する。True blockだけでもok。

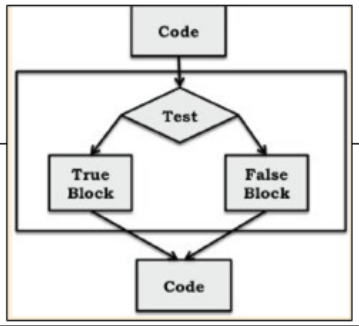
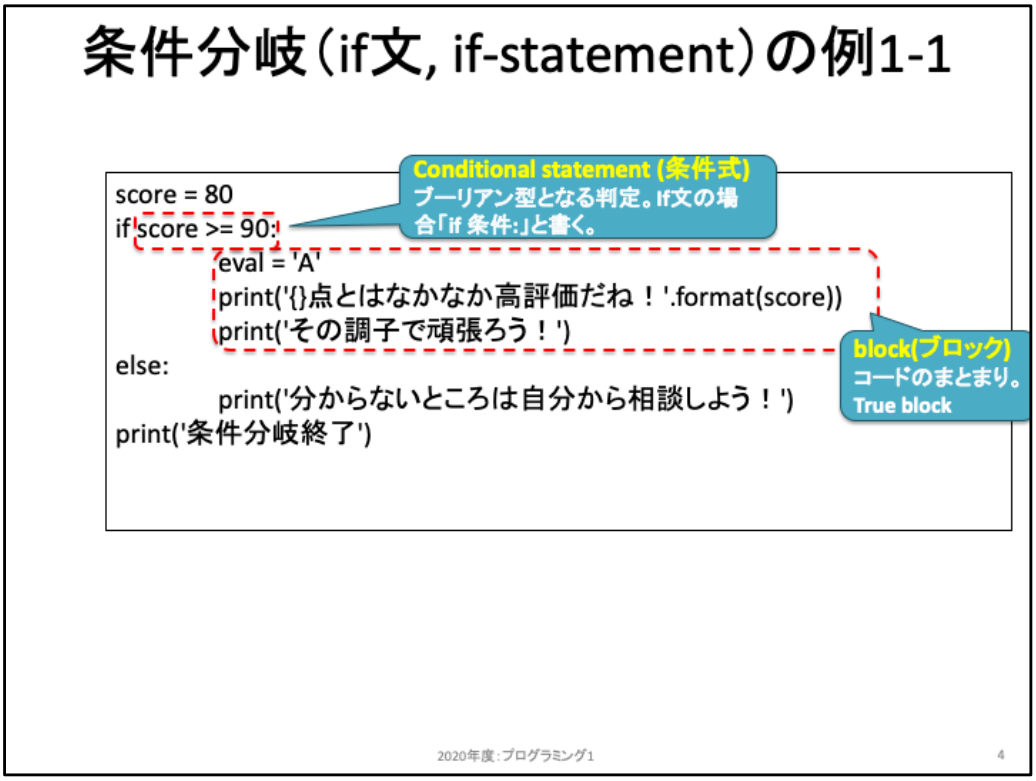


Fig. 2.3 Flow chart for conditional statement

2020年度：プログラミング1 3

プログラムを利用する現場では、ある特定条件を元にそれ満足する場合のコードと、満足しない場合のコードを分けて記述したい状況が良くある。条件分岐(conditional statement, if-statement)と呼ばれ、判定式(Test)とそれぞれの場合のコード(ブロック)とを分けて記述することになる。次に示す具体例を参照しながら条件分岐を学んでみよう。



このコード例は、変数scoreに保存された点数が90点以上ならば、変数evalに 'A' を保存し、2つのprint文を実行する。もしくは、点数が90点未満ならば1つのprint文を実行する。また、分岐を抜けた後は共通して「条件分岐終了」と出力する。というコードである。

ここで、条件分岐における「条件式」は、「score >= 90」と記述している箇所である。また、条件式の後ろにコロン「:」を記述することを忘れないようにしよう。条件式は、第2回で述べたような比較演算子を用いて表現する。例えば、変数 eval に保存された内容が文字列リテラル 'B' と等しいかどうか判断したい場合には次のように書く。

```
>>> eval == 'B'
```

この結果、等しいならば True となり、等しくないならば False という判定結果が得られる。

条件式の値は必ず bool 型リテラル（True, False）である。逆に言えば、評価結果がbool型とならないコードを記述すると、エラーになる（試してみよう）。条件式の結果が True のときに実行すべきコードが、少しスペースを加えて、同じ幅で揃えて記述した3行のコードである。このまとまりをブロックと呼び、ブロックを指定するためには(a)スペースを挿入することと、(b)複数コードをまとめたい場合には同一幅で揃えて書く必要がある。例えば以下のようなコードは True ブロックである。

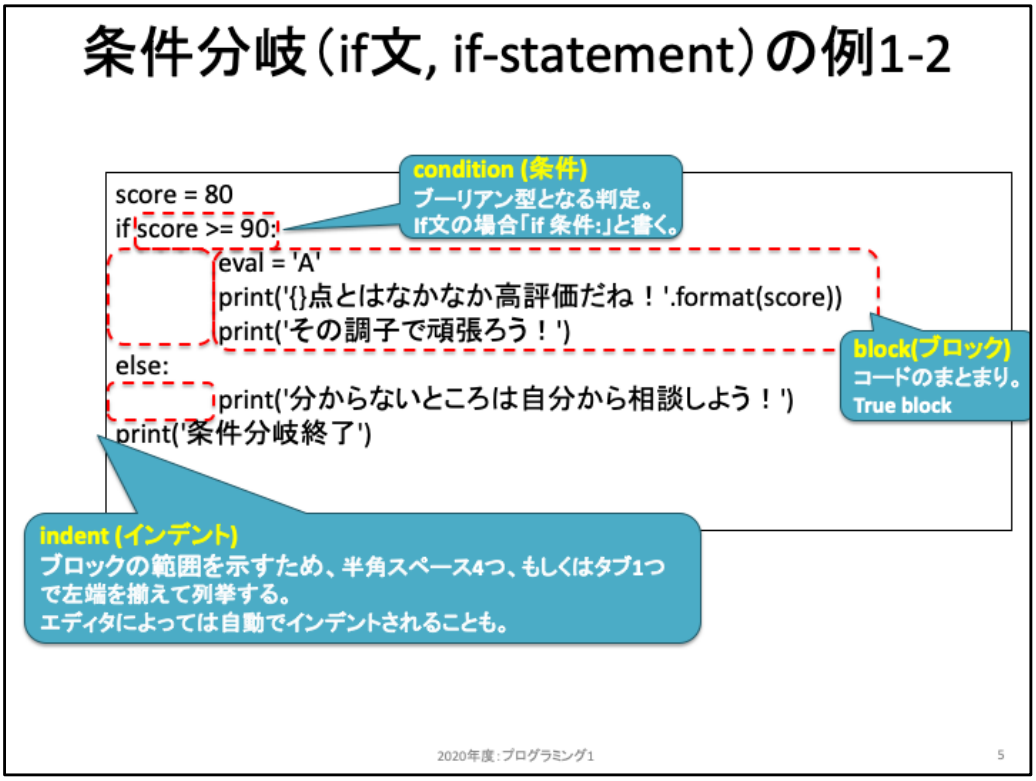
```
# 例1。Trueブロックのみで、ブロックに1行しか無い例。
if True:
   print('test')

# 例2。Trueブロックのみで、ブロックに2行ある例。
if True:
   print('test')
   print('test2')
```

上記に対し、以下のコードはブロックとして不適切であり、エラーになる（確認してみよう）。

```
# 例3。Trueブロックのみで、スペースが揃っていない例。
if True:
  print('test')
   print('test2')
```

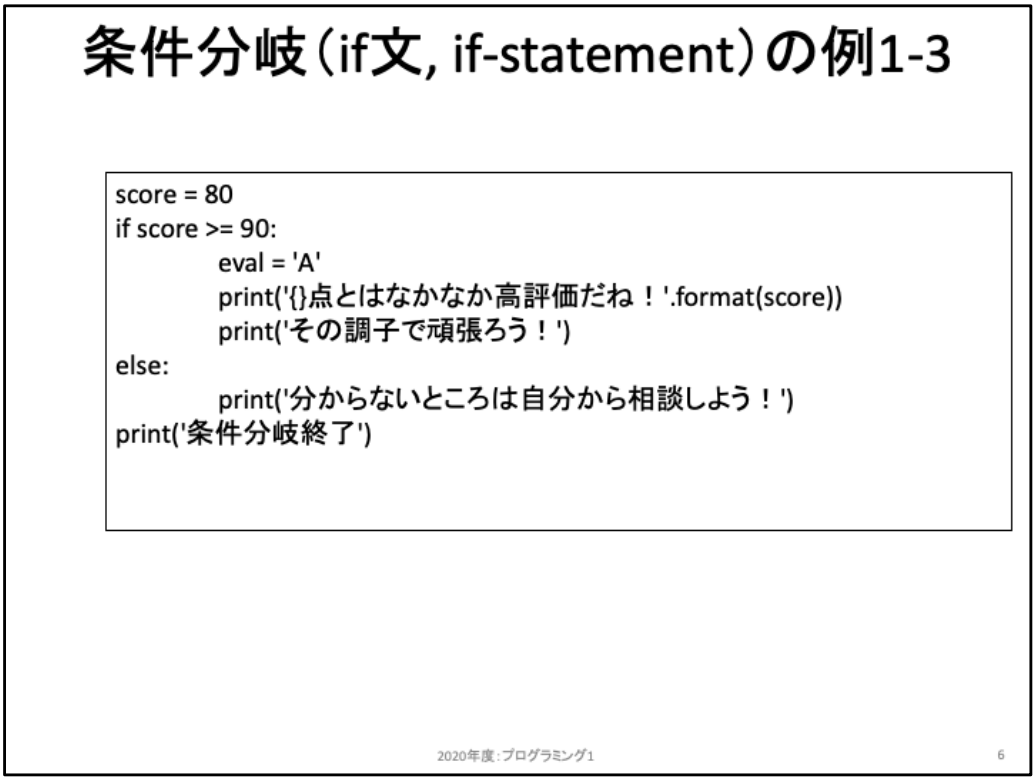
例3は、1つ目のprint文と2つ目のprint文の冒頭スペースが揃っておらず、異なる2つのブロックを記述していることになる。ところが、ブロックを書くためには条件分岐等特別な状況でしか書くことが許されていない。このため、例3はエラーとなる。



このスライドに示しているコードは、前ページと全く同じである。先頭示した「ブロックを示すために記述する冒頭スペース」のことをインデントと呼ぶ。基本的にはスペース4つを加えると覚えておこう。このようにブロックを記述することを「インデントを入れる」とか「インデントを揃える」というように呼ぶ。

注意してほしいのは、
・このコードを見ただけでどこがブロックになっているのか。
・そのブロックは True ブロックなのか、Falseブロックなのか。
を確認できるようになる必要があることだ。

例えば次のスライドを見てみよう。



このコードを見て、どこが True ブロックか分かるだろうか。どこがFalseブロックか分かるだろうか。なお、Falseブロックは必ずしも記述する必要はなく、Trueブロックのみのコードであることもある。更に、条件分岐は必ずしも1回で終わるとは限らない。次の例を見てみよう。

```
score = 80
eval = 'F'
if score >= 60:
   if score < 70:
      eval = 'D'
print(eval)
```

上記の例は、
・1つ目のif文のTrueブロック（1行のみ）の中に、さらにif文がある。
・2つ目のif文のTrueブロック（1行のみ）の中に、変数evalを設定する記述がある。
つまり、2つのブロックがそれぞれのif文におけるTrueブロックに対応している。ブロックの範囲はインデントが揃っているかどうかで判断される。

このように、どこからどこまでが1つのブロックなのか。そのブロックはどこに対応しているのか、を読み取れるようになろう。また、コードを書く際にはいきなり書くのではなく、スライド3ページのようにコード全体の分岐の流れをイメージしやすくなるように紙の上で図を書いて、流れを整理してみよう。この流れを検討する部分こそが、第1回授業資料における「理解・整理・翻訳」の前2つのステップに相当する。いきなり翻訳するのは慣れてきてからで十分だ。

条件分岐（if文, if-statement）の例2

```
score = 80
if score >= 90:
        eval = 'A'
        print('{}点とはなかなか高評価だね！'.format(score))
        print('その調子で頑張ろう！')
elif score >= 80:
        eval = 'B'
elif score >= 70:
        eval = 'C'
elif score >= 60:
        eval = 'D'
else:
        print('分からないところは自分から相談しよう！')
```

elif = else if。最初の条件「score>=90」に当てはまらない時（else時）に、改めて確認したい条件（if）があるという書き方。ここでは「80以上」という判定しか書いていないが、実際には「90以上ではない」ことが前提になったブロックになっている。

2020年度：プログラミング1 7

条件式を書く際、
条件1：90点以上ならば処理1を。
条件2：そうではなく、80点以上ならば処理2を。
条件3：そうではなく、70点以上ならば処理2を。
、、、
のように、条件2は「条件1でFalseになった上で追加したい条件」を書きたい場合には、「elif文」を使う。Elif は else ifの略であり、その前の条件式には合致しなかったことを踏まえて判断される。

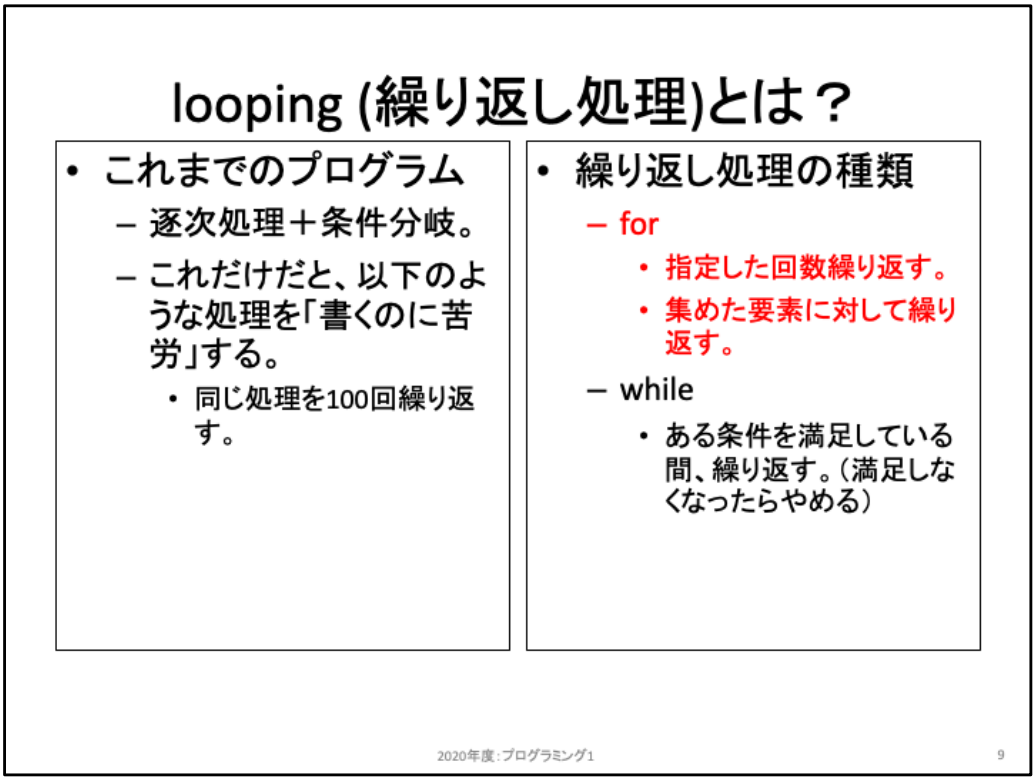
最後の else文 は、そこに到達するまでの全ての条件式に対しFalseとなったときだけ実行されることになる。今回のコード例では、変数scoreの中身が60未満のときだけelseブロックが実行される。

プログラミング1

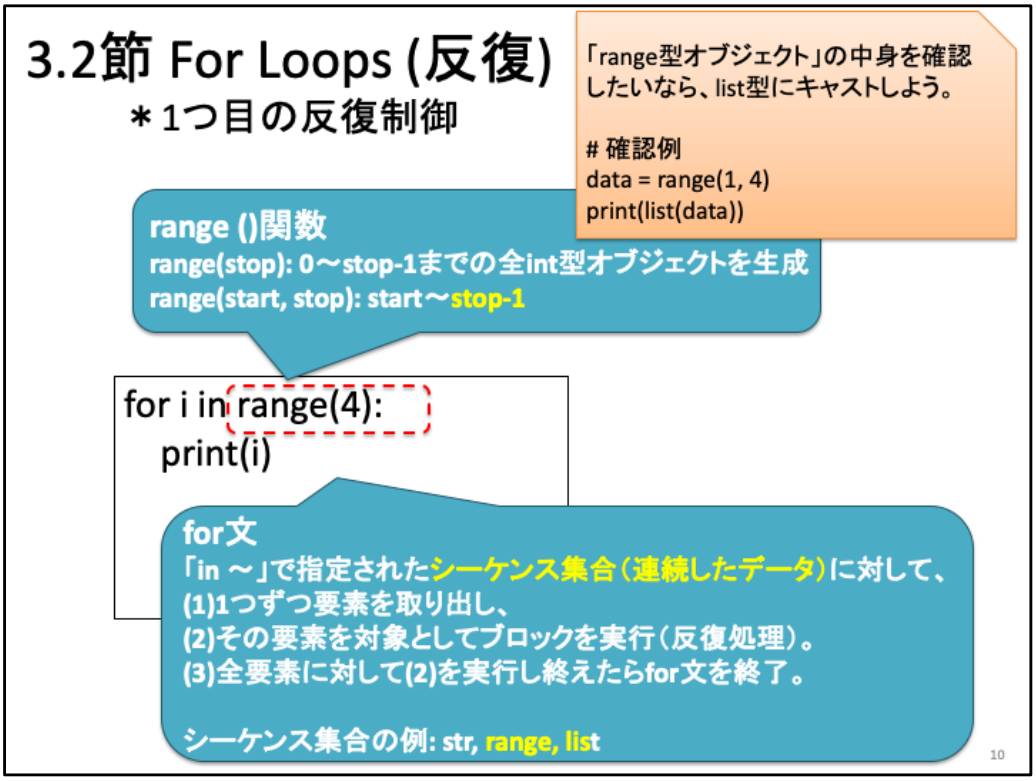
(第3回) インタプリタとスクリプトの体験2: 文字列とif文, 関数の利用

if文を使い、条件分岐できるようになろう。ブロックを指定するためのインデントを忘れずに。

1. Chapter 2.2, 3.2, 4.1.1の補足
   - 2.2 Branching Programs (条件分岐)
   - 3.2 For loop (反復処理)
   - 4.1.1 Function Definitions (関数定義)
   - Reserved words, 予約語
2. ペアプロ演習
3. 宿題



ある命令文を100回繰り返させる場合、同じ命令文を100回書くことで目的を達成することができる。しかし、何らかの理由でその命令文を修正したい場合、全ての命令文を修正する必要があるし、過不足なく100回書いたかどうかを確認することも難しい。これらの理由から、一般的にプログラミング言語においては「ある一定回数繰り返させたい」場合の処理をループ処理や繰り返し処理と呼び、専用の予約語を用いて記述することが多い。Pythonにおいては for, while の2つを用いて書くことができる。ここではfor文について学んでみよう。



For文は「for　変数　in　シーケンス:」という記述で書き始める。forと変数、変数とin、inとシーケンスの間は半角スペースが必要だ。

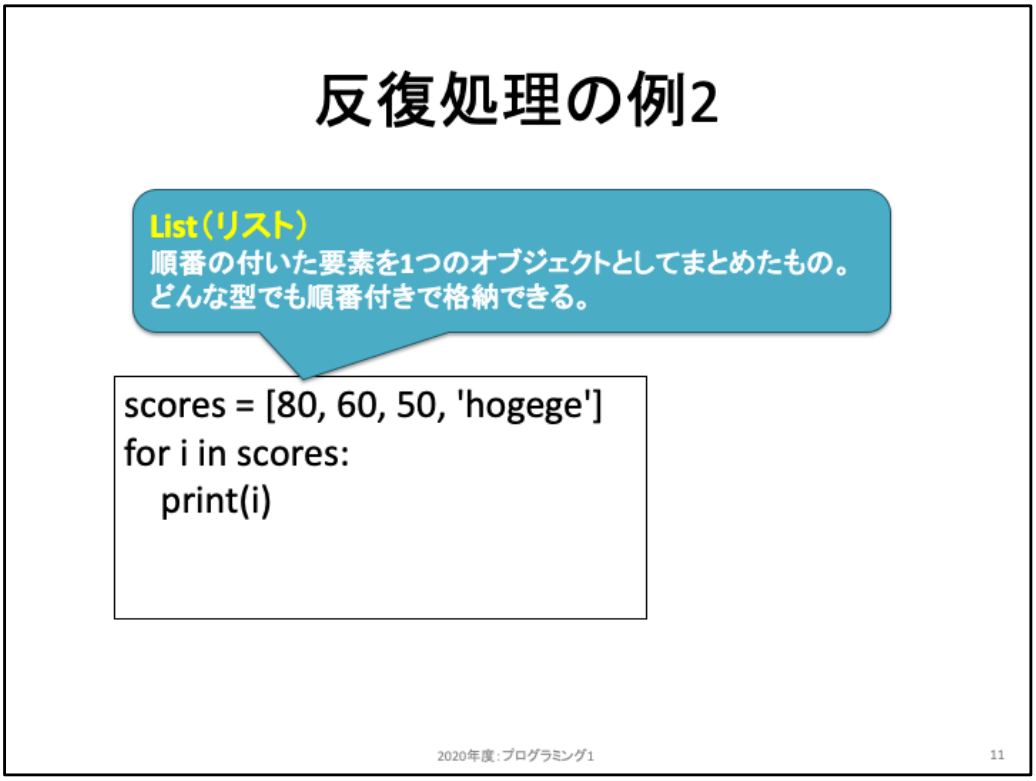
シーケンス（sequence）とは「順番の付いた集合」である。例えば、順番を考慮せず単に集合として「りんご、みかん、バナナ、、、」と果物を集めたものは集合ではない。これに対し「1番目にりんご、2番めにみかん、、、」のように順番が付いて並べられているものをシーケンスと呼ぶ。上記のコードでは「range(4)」がシーケンスに相当する。中身を確認するためには一度list型に変換すると分かりやすい。次のように実行してみよう。

```
>>> range(4)
range(0, 4)
>>> list(range(4))
[0, 1, 2, 3]
```

最初のコードがrange関数の結果であり、その結果は range(0,4) と出力されている。これはrange型オブジェクトと呼ばれ、そのままでは中身が分かりにくい。この中身を確認しやすくしたのが次の list(range(4)) である。[] はlist型と呼ばれており、中身が「0, 1, 2, 3」とint型リテラル0から3まで、合計4個の要素が並んでいる。List型は後日改めて確認するため、ここでは「range関数で引数を自然数kで指定して実行すると、0～k-1の数字が並んで用意される」ことを覚えよう。0から始まっているのは多くのプログラミング言語の慣習のようなものと捉えよう。

さて、range関数を用いてシーケンスを用意した。このシーケンスに対して、頭から一つずつ要素を取り出し、ブロック内の処理を実行するのがfor文である。Range関数で用意したシーケンスは「0, 1, 2, 3」であるため、最初の値は「0」だ。この値が変数 i に保存された状態でブロック内の処理が実行される。このため、その次は実行させたいブロックを、if文でも書いたようにインデントで指定してコードを書く必要がある。上記コードの例では print(i) となっており、変数iには0が保存されているため、最初は0が出力される。

ここでブロックは終わるため、for文はシーケンス内の次の値を確認する。次の値はint型リテラル1であり、これを変数iに保存してブロック内のコードを実行する。以下同様に、ブロック内のコードを処理し終えると次の値である2を変数iに保存してブロック内のコードを実行する。さらに、ブロック内のコードを処理し終えると次の値である3を変数iに保存してブロック内のコードを実行する。3に対する処理まで実行し終えると、シーケンスに残された値はなく、全ての値に対する実行をし終えていることが分かる。このようにシーケンス内全ての値に対して実行し終えると、for文を終了する（ブロックを抜けて、その下のコードに移動する）ことになる。ここではその下にはもうコードはないため、ここで動作終了となる。



For文はシーケンスに対して一つずつ要素を取り出して、処理を実行する。そのシーケンスを表現するためによく使う型がlistである。Listは他のオブジェクトを何でも並べて列挙することができる。

上記コード例では、1番目にint型リテラル80を、2番目にint型リテラル60を、3番目にint型リテラル50を、4番目にstr型リテたるhogegeをカンマ(,)で区切り、それら全体を[と]で囲うことでlist型オブジェクトを作成している。なお、ここでは分かりやすいように「1番目」から数え上げたが、Pythonのlistや他の多くの言語における配列と呼ばれるオブジェクトにおいては、殆どの場合「0番目」から数える。例えば次のように実行すると何が出力されるか確認してみよう。

```
>>> print(scores[0])
```

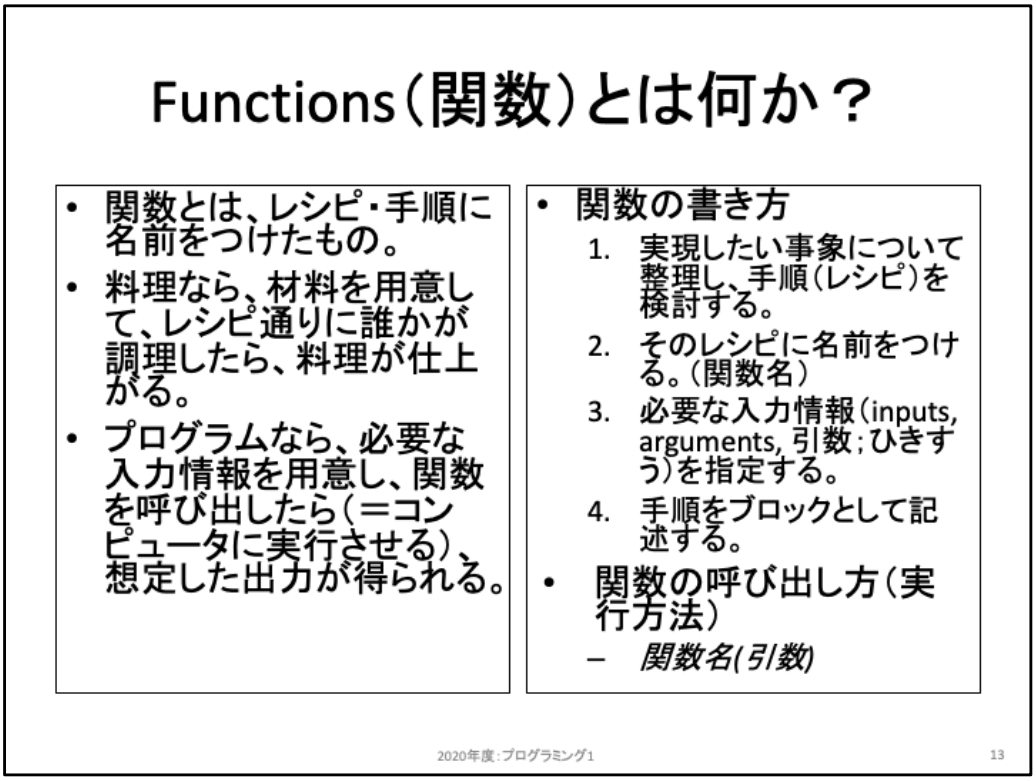
変数scoresの中に上記コード例1行目のように保存した後で、scores[0] をprint関数で出力させると、80が出力されるはずだ。このように0番目から数え始め、最後はscores[3] に 'hogege' が保存されている。

プログラミング1

(第3回) インタプリタとスクリプトの体験2: 文字列とif文, 関数の利用

1. Chapter 2.2, 3.2, 4.1.1の補足
   - 2.2 Branching Programs (条件分岐)
   - 3.2 For loop (反復処理)
   - 4.1.1 Function Definitions (関数定義)
   - Reserved words, 予約語
2. ペアプロ演習
3. 宿題

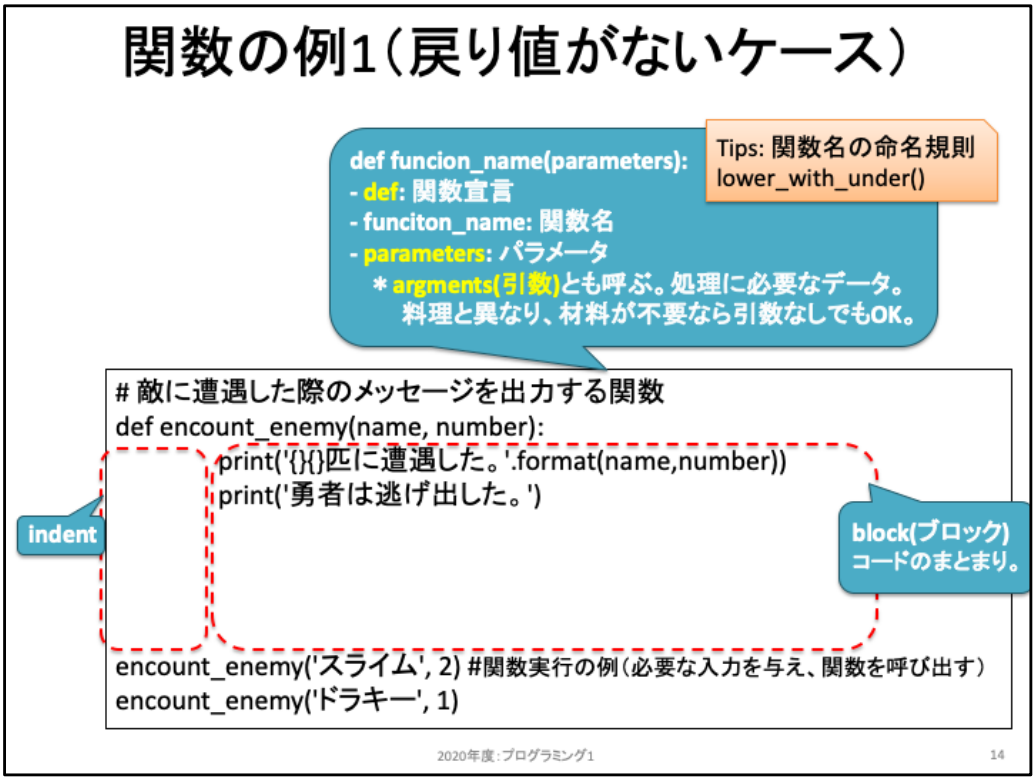
まずはシンプルな繰り返しに慣れよう。



これまでにprint(), type(), str(), str.format(), range(), list() ぐらいの関数を使ってきた。関数と呼んでいるが、数学における関数とは異なる。プログラムにおける関数とは何だろうか。

第2回の資料で「情報を導くためのレシピ、手順、手続き」のことを命令的知識と呼ぶことを紹介した。レシピとは、例えば卵焼きなどの料理を作る際のレシピを指しており、必要な材料を用意し、特定手順で調理することでその料理を作り上げることができる。これがレシピだ。これに対しプログラムにおける関数とは、必要な入力データを用意されると、特定手順で処理することで最終的にその関数が目標としている結果を算出することができる手順としてまとめられたものである。例えば、四角形の面積を求めたいならその四角形を構成する2辺の長さを入力データとして用意する。そのデータを元に、2つのデータを掛け合わせることで面積を求めることができる。この手続をまとめたものが関数だ。

関数を書くためには、スライド右側の手順で記述する必要がある。また記述した関数は「関数名(引数)」という形で実行することができる。具体例を次のスライドで眺めてみよう。

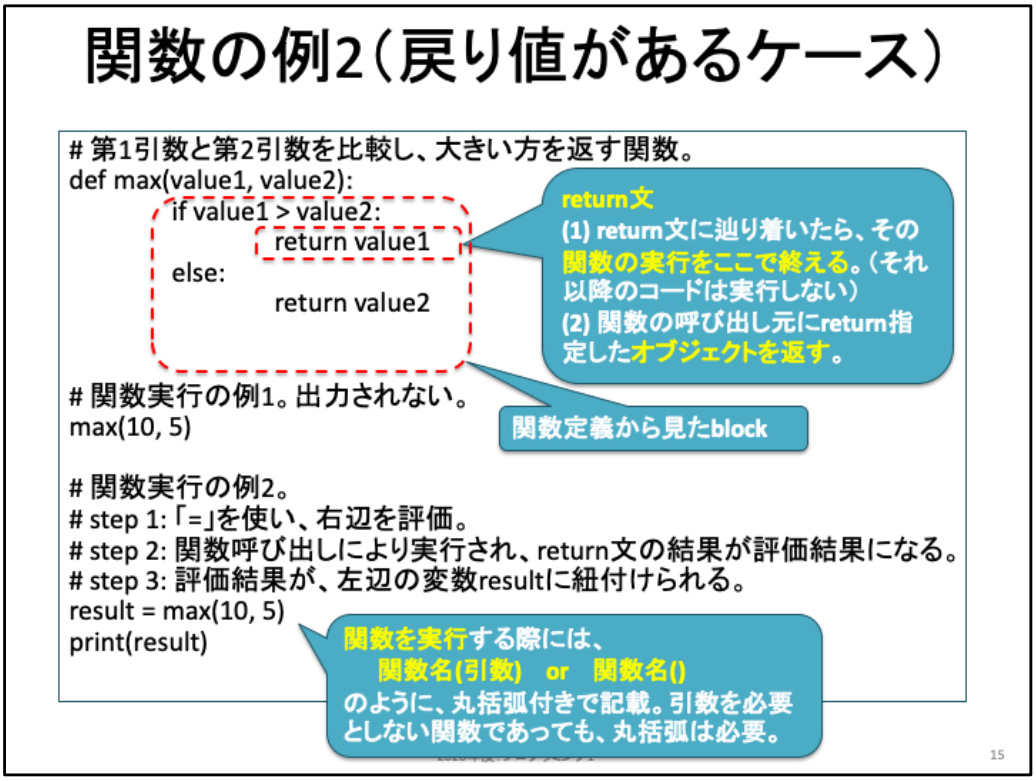


関数は、戻り値（または返り値、return values）の有無により記述方法が異なる。戻り値とは、この関数を実行した後に何かしらの処理結果を「関数呼び出し元に戻す値」のことである。例えば面積を求めた結果を受け取って利用したいなら、それは戻り値として命令を記述する必要がある。これに対し、このスライドの例は戻り値がない関数の例である。中身を見ていこう。

関数を定義する際には def 文で「def　関数名(引数):」のように記述する。関数名とはここで定義したい関数に対する名称であり、これまでのprint, type等のように動作をイメージしやすい名称をつけると読みやすいコードとなる。また、関数として実行すべきコードは、インデントで揃えてブロックとして記述する必要がある。if文、for文同様、コードからどこがブロックなのか読み取れる必要がある。

引数が複数必要な場合にはカンマで区切って列挙することになる。引数の名称も、中身をイメージしやすい名称をつけることが望ましい。なお、ここで指定した変数名は「関数の外部」からは参照することができない、一時的な変数である。例えば上記コードでは変数name, 変数numberを引数として設定しているが、ここで定義してしまったあとで他の関数でも同じ名前の引数を書いても動作が混乱することはない。あくまでも関数実行時に一時的に利用するための変数だということを覚えておこう。より詳細は後日扱う。

上記コード例では、「encount_enemy」という名前の関数を定義しており、2つの引数name, numberを必要とすることを定義している。この関数encount_enemyを使うためには、最後の2行のように「encount_enemy(第1引数, 第2引数)」として命令文を書き、実行することになる。この命令文を書く際に指定した第1引数が、関数定義の際に書いた変数nameとして保存され、第2引数が変数numberとして保存され、ブロック内のコードを実行する。ブロック内のコードを実行し終えると、実行命令を書いていた行に戻り、処理を続行する。上記コード例における空白を除くと、6行のコードがある。この処理順番を列挙していくと、1行目は関数定義であり、そのブロックである2～3行目はこの時点では実行しない。4行目は関数実行になっており、この時点でencount_enemy関数を一度実行する。関数を実行し終えると4行目を実行し終えたと状態となり、次の行である5行目に移動する。5行目は再び関数呼び出しであり、実行される。実行し終えると5行目を実行し終えた状態となり、次の行に移動しようとするがこれ以降はないため、これでプログラムの実行を終える。



関数を実行し終えた際に処理結果を関数呼び出し元に戻すためには「return文」により命令する。上記コード例においては、return文が2箇所に記述されているが、この場合にはどちらか片方を実行した時点で関数の実行を終える。また、インデントにより揃っているブロックが複数ある点に注意しよう。if文とelse文を含む4行が一つのブロックであり、このブロックが関数maxが呼び出されたときに実行するべきブロックである。return value1は、if文における True ブロックであり、return value2 は False ブロックである。

関数実行の例1は、
max(10,5) は関数呼び出しであり、max関数における第1引数value1=10、第2引数value2=5として実行される。この結果、value1はvalue2より大きいためTrueブロックが実行され、value1に保存されている値10が、関数呼び出し元に戻される。関数呼び出し元ではこの値を何に使うかは書いていないため、10という値が出力されることもなく次の命令文に移動する。

return文により値が戻されているかどうかを確認するためには、=演算子を用いて変数に保存するのが分かりやすい。

ここで、関数maxは2つの引数を受け取り、ブロック内のコードを実行することは既に述べたとおりだ。ただし、関数を呼び出す際に「どのような引数を与えられるか」は制限がかけられていないため、例えば「max('スライム', 'ゴブリン')」のようにstr型リテラルが指定されることもあり得る。このように想定外の値が渡された際にどうなるのか、試してみよう。

関数例3

```
1   scores = [80, 60, 100, 0, 90]
2
3   def change_score_to_rank(score):
4       if score >= 90:
5           result = 'A'
6       elif score >= 80:
7           result = 'B'
8       elif score >= 70:
9           result = 'C'
10      elif score >= 60:
11          result = 'D'
12      else:
13          result = 'F'
14      return result
15
16  for i in scores:
17      rank = change_score_to_rank(i)
18      print('{}の評価は{}です'.format(i, rank))
```

2020年度：プログラミング1 16

関数change_score_to_rankは、100点満点で記載された点数を引数として受け取り、90点以上ならば'A'、80点以上90点未満ならば'B'、70点以上80点未満ならば'C'、60点以上70点未満ならば'D'、それ以外ならば'F'を変数resultに保存し、関数呼び出し元に戻す。

実際の実行順序は、以下のように進む。これを理解できるようになろう。
1行目（リストを保存）。
3行目～14行目（関数定義を読み込むだけで実行はしない）。
16行目（for文の開始。最初はscores[0]が変数iに保存された状態でブロック内の命令を実行）。
　17行目（=演算子により、右辺の関数呼び出しを実行。実行後17行目に戻り、returnで戻された値が処理結果となる。この結果を変数rankに保存）。
　18行目（print出力）。
16行目（for文2度目。scores[1]が変数iに保存された状態でブロック内の命令を実行）。
　17行目（=演算子により、右辺の関数呼び出しを実行。実行後17行目に戻り、returnで戻された値が処理結果となる。この結果を変数rankに保存）。
　18行目（print出力）。
16行目（for文3度目。scores[2]が変数iに保存された状態でブロック内の命令を実行）。
　17行目（=演算子により、右辺の関数呼び出しを実行。実行後17行目に戻り、returnで戻された値が処理結果となる。この結果を変数rankに保存）。
　18行目（print出力）。
16行目（for文の4度目。scores[3]が変数iに保存された状態でブロック内の命令を実行）。
　17行目（=演算子により、右辺の関数呼び出しを実行。実行後17行目に戻り、returnで戻された値が処理結果となる。この結果を変数rankに保存）。
　18行目（print出力）。
16行目（for文の5度目。scores[4]が変数iに保存された状態でブロック内の命令を実行）。
　17行目（=演算子により、右辺の関数呼び出しを実行。実行後17行目に戻り、returnで戻された値が処理結果となる。この結果を変数rankに保存）。
　18行目（print出力）。
16行目（for文の5度目。しかしscores[5]は存在しないため、ここでループ実行を終了）。
19行目の以降のコードは存在せず、プログラムの実行を終了。

プログラミング1

(第3回) インタプリタとスクリプトの体験2: 文字列とif文, 関数の利用

1. Chapter 2.2, 2.3, 4.1.1の補足
   - 2.2 Branching Programs (条件分岐)
   - 3.2 For loop (反復処理)
   - 4.1.1 Function Definitions (関数定義)
   - Reserved words, 予約語
2. ペアプロ演習
3. 宿題

オリジナルの関数(≒レシピ)を定義できるようになろう。

Reserved words, 予約語

https://goo.gl/4TclUz

- 一覧（赤丸は今回出てきた予約語）

False	class	finally	is	return
None	continue	for	lambda	try
True	def	from	nonlocal	while
and	del	global	not	with
as	elif	if	or	yield
assert	else	import	pass	
break	except	in	raise	

演習

前回の続き: 初めてのペア・プログラミング

プログラミング1

(第3回) インタプリタとスクリプトの体験2: 文字列とif文, 関数の利用

1. Chapter 2.2, 2.3, 4.1.1の補足
   - 2.2 Branching Programs (条件分岐)
   - 3.2 For loop (反復処理)
   - 4.1.1 Function Definitions (関数定義)
   - Reserved words, 予約語
2. ペアプロ演習
3. 宿題

if文を使い、条件分岐できるようになろう。ブロックを指定するためのインデントを忘れずに。

まずはシンプルな繰り返しに慣れよう。

オリジナルの関数(≒レシピ)を定義できるようになろう。

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

宿題

- 復習
  - 教科書2章まで＋3.1節(for文)。
  - 課題レポート2　＊講義ページ参照。
- 予習: 教科書読み
  - 3章
    - 3.1 Exhaustive Enumeration
- 復習・予習(オススメ): progate

参考文献

- 教科書: Introduction to Computation and Programming Using Python, Revised And Expanded Edition
- Reserved words, https://goo.gl/4TclUz