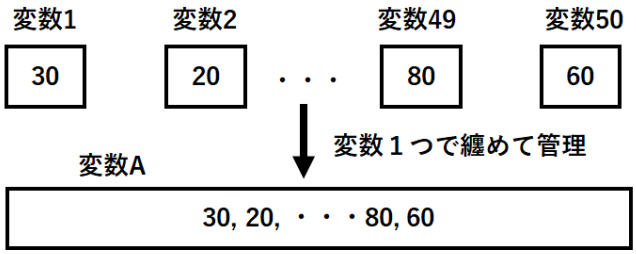
2021年度 東京成徳大学 特別講座 資料
東京工科大学コンピュータサイエンス学部
福西広晃

第02回 Python基礎（データの扱い方）

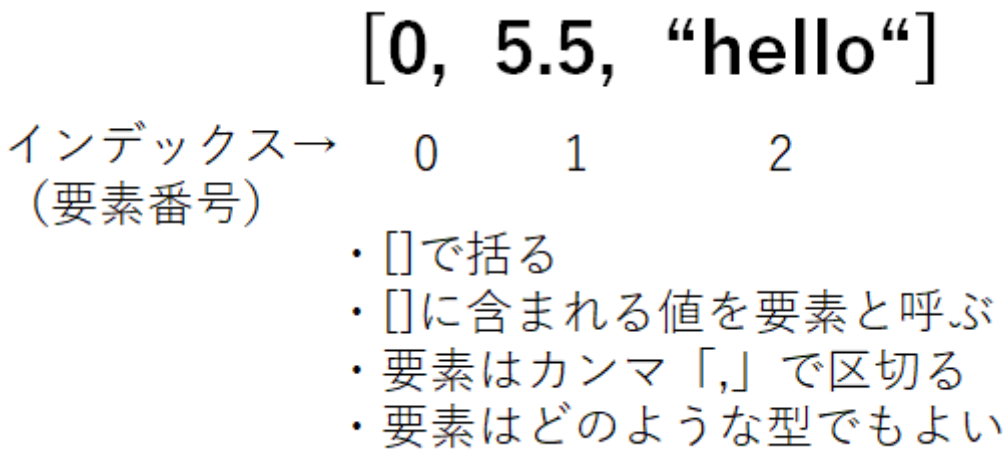
- 前回は、個々の数値や文字を扱ったが、プログラミングでは複数の値から成るデータを扱うことも多い
- 複数の値を1つの変数で管理しながら、個々の値（ 要素 と呼ぶ）を取り出したり、場合によっては変更したりできると便利なことがある。そのためにPythonでは **リスト** 、 **タプル** 、 **辞書** 、 **集合** のデータ型が提供されている
- 今回は、特に重要な、 **リスト** 、 **辞書** を学習する

【例】50人の学生のテスト結果をプログラムで処理したいときに、個々の点数を変数に代入していては変数の管理が大変になる。50人分の点数を纏めた１つの変数で管理するほうが扱い易い



リスト

- **リスト** は、数値や文字列などの基本データ型と並んでPythonでは高頻度で使われるデータ型である
- **リスト** は、複数の値をまとめて扱う場合に使用する
- **リスト** は全体を [ ] で囲み、各要素は , で区切る
- **リスト** はどのような型でも格納することができ、リストの中にリストを格納することも可能である
- **リスト** では **インデックス** (要素番号)を用いて、要素の追加や入れ替えなどが可能である



リストの操作の基礎

リストを変数に代入

In [ ]:
```
# 5人のテスト結果が90点、85点、75点、95点、80点であったとする
listA = [90, 85, 75, 95, 80]
print(listA)
```

要素数をカウント

- len関数 を用いる

  len(リスト)

In [ ]:
```
print("要素数:", len(listA))
```

要素の最大値、最小値を抽出

- 最大値の抽出には、 max関数 を用いる

  max(リスト)

- 最小値の抽出には、 min関数 を用いる

  min(リスト)

In [ ]:
```
print('最大値: ', max(listA))
print('最小値: ', min(listA))
```

要素の取り出し

- インデックス（要素番号）を指定
  リスト[インデックス]

In [ ]:
```
print(listA[0]) # インデックス0を取り出して出力
print(listA[1]) # インデックス1を取り出して出力
print(listA[2]) # インデックス2を取り出して出力
print(listA[3]) # インデックス3を取り出して出力
print(listA[4]) # インデックス4を取り出して出力
```

要素の置換

- インデックス(要素番号)で置換前の要素を指定し、置換したい値を代入

In [ ]:
```
# 変更前をリストを表示
print("置換前:", listA)

# インデックス(要素番号)0を置換
#  （最初の学生の点数は70点だった）
listA[0] = 70

# 変更後をリストを表示
print("置換後:", listA)
```

複数のリストの結合

- リストA = リストB + リストC
- 最初のテストの受験者5人の結果は、90点、85点、75点、95点、80点であったとする。その後、新たに4人がテストを受験して、70点, 80点, 75点, 90点であった。さらに、1人がテストを受験して85点であった。

In [ ]:
```
listA = [90, 85, 75, 95, 80]
listB = [70, 80, 75, 90]
listC = [85]

# listAとlistBを結合させてlistDを作成
listD = listA + listB

# リストを表示
print("1回目の結合: ", listD)

# listDとlistCを結合させてlistDを拡張
listD = listD + listC     # listD += listC でも可

# リストを表示
print("2回目の結合: ", listD)
```

要素の並び替え

- sorted関数 を用いる
  - 昇順に並び替え（ただし要素は数値のみか、文字列のみ）

    sorted(リスト)
  - 降順に並び替え

    sorted(リスト,reverse=True)`

要素が数値の場合

In [ ]:
```
listD = [90, 85, 75, 95, 80, 70, 80, 75, 90, 85]

# listDを昇順に並び替えてlistDに代入
listE = sorted(listD)
print("昇順に並び替え: ", listE)

# listDを降順に並び替えてlistFに代入
listF = sorted(listD, reverse=True)
print("降順に並び替え: ", listF)
```

要素が文字列の場合(アルファベット順)

In [ ]:
```
listG = ["dog", "cat", "tiger", "bird", "horse"]

listH = sorted(listG)
print("昇順に並び替え: ", listH)

listI = sorted(listG, reverse=True)
print("降順に並び替え: ", listI)
```

反復（for文）による要素の取得（1）

```
for 要素を代入する変数 in リスト:
    実行する処理
```

要素を1個ずつ変数に代入して繰り返す

要素の平均値を計算

10人のテストの点数の平均値

In [ ]:
```
# 10人のテストの点数が格納されたlistD
listD = [90, 85, 75, 95, 80, 70, 80, 75, 90, 85]

sum = 0                 # 合計値の初期値を0とする
for ele in listD:       # 要素を１個ずつ取り出して変数eleに代入
    sum = sum + ele     # 要素を足す　（sum += eleも可）

# 平均値 = 合計/要素数
ave = sum / len(listD)  # 平均値をaveという変数に代入

print("平均値：", ave)
```

反復（for文）による要素の取得（2）

- インデックス(要素番号)と要素のセットを同時に取り出して繰り返し処理を行う

  for インデックスを代入する変数, 要素を代入する変数 in enumerate(リスト):
  実行する処理

In [ ]:
```
for i, ele in enumerate(listD):
    print("インデックス", i, "要素", ele)
```

練習

5人の身長を測った結果、170cm、175cm、160cm、155cm、165cmであったとする

(1) 5人の身長を要素とするリストをhigh1の変数名で作成して、要素数を表示しよう

In [ ]:
```

```

(2) 前から4番目の人の身長に間違いがあったため150cmに変更して、リストを表示しよう

In [ ]:
```

```

(3) 新たな5人の身長が180cm、155cm、160cm、165cm、170cmとする。 新たな5人の身長を要素とするリストをhigh2の変数名で作成してみよう。 さらに、high1とhigh2を結合したリストをhigh3の変数名で作成し、表示しよう

In [ ]:
```

```

(4) 10人の身長を降順で並び替えたリストをhigh4で作成し、表示しよう

In [ ]:
```

```

(5) for文を利用し、10人の身長の平均値を計算して表示しよう

In [ ]:
```

```

解答例

In [ ]:
```
# (1)
high1 = [170, 175, 160, 155, 165]
print("要素数: ", len(high1))

# (2)
high1[3] = 150 # 4番目の人のインデックスは3であることに注意
print("修正:", high1)

# (3)
high2 = [180, 155, 160, 165, 170]
high3 = high1 + high2
print("結合:", high3)

# (4)
high4 = sorted(high3, reverse=True)
print("並び替え:", high4)

# (5)
sum = 0
for ele in high4:
    sum = sum + ele      # ele += eleでもOK
ave = sum / len(high4)
print("平均", ave)
```

メソッドを用いたリストの操作

- メソッド とはある処理のまとまりに名前をつけたもので、オブジェクト（変数や値のこと）の後にドット「.」を付けて使用する
  リスト.メソッド名()

新たな要素の追加

- 使用メソッド

  リスト.append(追加する要素)
- リスト同士の足し算でも可能

5人のテスト結果が90点、85点、75点、95点、80点であったとする

In [ ]:
```
# 追加前のリストを表示
listA = [90, 85, 75, 95, 80] # 5人の点数
print("追加前:", listA)

# 新たな受験者の70点を追加
listA.append(70)

# 追加後のリストを表示
print("1回目追加後:", listA)

# さらに、新たな受験者の60点を追加
listA.append(60)

# 実行後のリストを表示
print("2回目追加後:", listA)
```

リストの足し算で要素を追加する場合

In [ ]:
```
# 追加前のリストを表示
listA = [90, 85, 75, 95, 80] # 5人の点数
print("追加前:", listA)

# 新たな受験者の70点を追加
listA = listA + [70]

# 追加後のリストを表示
print("1回目追加後:", listA)

# さらに、新たな受験者の60点を追加
listA = listA + [60]

# 実行後のリストを表示
print("2回目追加後:", listA)
```

リストの拡張（リストに別のリストを結合）

- 使用メソッド

  リスト.extend(追加するリスト)
  `

- リスト同士の足し算でも可能

In [ ]:
```
listA = [90, 85, 75, 95, 80]  # 5人の点数
listB = [70, 60]              # 追加の2人の点数

# listAにlistBを結合させて拡張
listA.extend(listB)

# 追加後のリストを表示
print("追加後: ", listA)
```

リストの足し算で別のリストを追加する場合

In [ ]:
```
listA = [90, 85, 75, 95, 80]  # 5人の点数
listB = [70, 60]              # 追加の2人の点数

# listAにlistBを結合させて拡張
listA = listA + listB

# 追加後のリストを表示
print("追加後: ", listA)
```

特定の値の要素の削除

- 使用メソッド
  リスト.remove(削除する要素)`
- リストの中に該当する要素が見つからなかった場合はエラー(ValueError)が発生する
- 削除される要素は「最初に該当したもの」のため、同じ値が2回以上出てくると最初に出る要素だけ削除される

In [ ]:
```
# 削除前のリストを表示
listA = [90, 85, 75, 95, 80] # 5人の点数
print("削除前:", listA)

# 85点を削除
listA.remove(85)
```

```
# 削除後のリストを表示
print("削除後:", listA)
```

同じ値の2つの要素を削除する場合

In [ ]:
```
# 削除前のリストを表示
listC = [90, 85, 75, 95, 85] # 5人の点数
print("削除前:", listC)

# 1つ目の85点を削除
listC.remove(85)

# 削除後のリストを表示
print("1回目削除後:", listC)

# 2つ目の85点を削除
listC.remove(85)

# 削除後のリストを表示
print("2回目削除後:", listC)
```

インデックスを用いた要素の削除

- 使用メソッド

  リスト.pop(削除する要素のインデックス)

In [ ]:
```
# 削除前のリストを表示
listA = [90, 85, 75, 95, 80] # 5人の点数
print("削除前:", listA)

# インデックス(要素番号)0番目を削除
listA.pop(0)

# 削除後のリストを表示
print("削除後:", listA)
```

要素の並び順を反転させる

- 使用メソッド

  リスト.reverse()

In [ ]:
```
# 反転前のリストを表示
listA = [90, 85, 75, 95, 80] # 5人の点数
print("反転前:", listA)

# 要素の並び順を反転
listA.reverse()

# 反転後のリストを表示
print("反転後:", listA)
```

特定の値をもつ要素の個数をカウント

- 使用メソッド
  リスト.count(検索する値)

In [ ]:
```
listD =  [90, 85, 90, 85, 90] # 5人の点数

# 90点の人数をカウント
cnt = listD.count(90)

# 人数表示
print(cnt,"人")
```

要素の並び替え

- 使用メソッド
  - 昇順に並び替え（ただし要素は数値のみか、文字列のみ）

    リスト.sort()
  - 降順に並び替え

    リスト.sort(reverse=True)
- sorted関数 を用いても同じ（関数は引数としてリストを入れる）
  - 昇順に並び替え（ただし要素は数値のみか、文字列のみ）

    sorted(リスト)
  - 降順に並び替え

    sorted(リスト,reverse=True)`

In [ ]:
```
# 並び替え前のリストを表示
listA = [90, 85, 75, 95, 80] # 5人の点数
print("並び替え前:", listA)

# 昇順に並び替え
listA.sort()

# 昇順に並び替え後のリストを表示
print("昇順に並び替え後:", listA)

# 降順に並び替え
listA.sort(reverse = True)

# 降順に並び替え後のリストを表示
print("降順に並び替え後:", listA)
```

sorted関数を用いた場合

In [ ]:
```
# 並び替え前のリストを表示
listA = [90, 85, 75, 95, 80] # 5人の点数
print("並び替え前:", listA)

# 昇順に並び替え
sorted(listA)

# 昇順に並び替え後のリストを表示
print("昇順に並び替え後:", listA)

# 降順に並び替え
sorted(listA, reverse = True)

# 降順に並び替え後のリストを表示
print("降順に並び替え後:", listA)
```

練習

10人のテストの点数をまとめたリストとしてscore1 = [60, 70, 80, 60, "欠席", 90, 40, 60, 70, 80]がある。

(1) score1のリストの中で、欠席を削除して表示してみよう

In [ ]:

(2) socre1の70点が何人いるかをカウントして表示してみよう

In [ ]:

(3) 受験者の点数を降順に並び替えて表示してみよう。ただし、sortメソッドを使うものとする

In [ ]:

解答例

In [ ]:

```
score1 = [60, 70, 80, 60, 90, 40, 60, 70, 80, "欠席"]

# (1)
score1.remove("欠席")
print(score1)

# (2)
print(score1.count(70), "人")

# (3)
score1.sort(reverse=True)
print(score1)
```

リスト内の要素の統計計算

- 標準ライブラリの statistics を用いれば、平均、中央値、最頻値、分散、標準偏差を算出できる
- 標準ライブラリを使う場合、import文で呼び出す

```
import statistics
statistics.mean(リスト)      # 平均値
statistics.median(リスト)    # 中央値
statistics.median(リスト)    # 最頻値
statistics.pvariance(リスト) # 分散
statistics.pstdev(リスト)    # 標準偏差
statistics.variance(リスト)  # 不偏分散（母集団の分散の推定）
statistics.stdev(リスト)     # 不偏分散の平方根をとった標準偏差
```

In [ ]:

```
import statistics

listJ = [90, 85, 75, 95, 80, 70, 75, 90, 80, 75] # 5人の点数

mean = statistics.mean(listJ)
print("平均値: ",  mean)

median = statistics.median(listJ)
print("中央値: ", median)
```

```
mode = statistics.mode(listJ)
print("最頻値: ", mode)

pvariance = statistics.pvariance(listJ)
print("分散: ", pvariance)

pstdev = statistics.pstdev(listJ)
print("標準偏差: ", pstdev)

# listJは母集団から抽出した標本（サンプル）とした場合の
# 母集団の分散の推定値（不偏分散を使う）
variance = statistics.variance(listJ)
print("不偏分散: ", variance)

# 上記の不偏分散の平方根を取った値であるが、
# 母集団の標準偏差の推定値とはならないことに注意（詳細は割愛）
stdev = statistics.stdev(listJ)
print("不偏標準偏差: ", stdev)
```

練習

10人の身長をまとめたリストとしてhighA = [172, 175, 165, 168, 178, 172, 170, 166, 169, 164]がある。平均値、不変分散、不偏標準偏差を求めてみよう。ただし、statisticsのライブラリを用いるものとする。

In [ ]:

```
```

解答例

In [ ]:

```
highA = [172, 175, 165, 168, 178, 172, 170, 166, 169, 164]

mean = statistics.mean(highA)
print("平均値: ",  mean)

# listJは母集団から抽出した標本（サンプル）とした場合の
# 母集団の分散と標準偏差の推定量
variance = statistics.variance(highA)
print("不偏分散: ", variance)

stdev = statistics.stdev(highA)
print("不偏標準偏差: ", stdev)
```

練習

10人のテストの点数をまとめたリストとしてscore1 = [50, 35, 80, 60, 70, 90, 40, 60, 75, 10]がある。その後、全員の点数を50点満点に圧縮することになった。圧縮後の点数をまとめたリストとしてscore2を作成し、表示しよう。ここでは、内包表記を用いるものとする。

In [ ]:

```
```

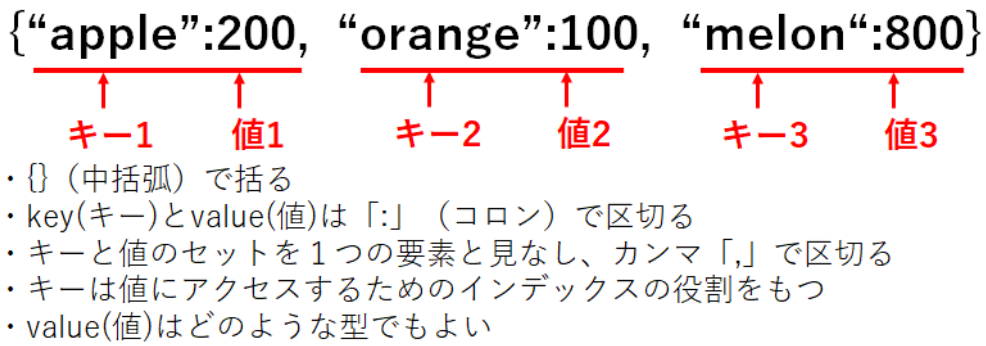
解答例

In [ ]:

```
score1 = [50, 35, 80, 60, 70, 90, 40, 60, 75, 10]
score2 = [i * 0.5 for i in score1]
print("圧縮後の点数: ", score2)
```

辞書（ディクショナリ）

- 辞書 は、 リスト と同じように複数の値をまとめて扱う場合に使用する
- 辞書 と リスト の違い
  - リスト: インデックス（要素番号）を使って要素を取り出す
  - 辞書: key (キー)を使って要素を取り出す(キーはユーザーが決める)
- 辞書 は、 key (キー)と value （値）のセットが1つの要素となり、1つの 辞書 の中で同じ key を使用することができない
- 辞書 は全体を {} (中括弧)で囲み、 key と値は ： （コロン）で区切り、各要素（キーと値のペア）は , （カンマ）で区切る
- 辞書 の要素の並びに順番の考え方がない(リストでは左の要素から順に順番がある、つまりインデックス)
- 辞書 のkeyを使って各要素の value (値)にアクセスする
- 辞書 にはどのような型の値でも格納することができる



辞書の操作の基礎

辞書を変数に代入

In [ ]:
```
#テストの点数がAさん90点、Bさん85点、Cさん75点、Dさん95点、Eさん80点であったとする
dicA = {"Aさん":90, "Bさん":85, "Cさん":75, "Dさん":95, "Eさん":80} # 5人の点数

# 辞書表示
print(dicA)
```

要素数（キーと値のペア）をカウント

- len関数 を用いる

  len(辞書)

In [ ]:
```
print("要素数: ", len(dicA))
```

値（value）の取り出し

辞書[キー]

- リストでは リスト[インデックス] で要素を取り出したことと考え方は同じ

In [ ]:
```
# Aさんの点数をprint関数で出力
print(dicA["Aさん"], "点")

# Aさんの点数と取り出して、valという変数に代入
val = dicA["Aさん"]
print(val, "点")

# キーを変数にしたパターン
key = "Aさん"
val = dicA[key]
print(val, "点")
```

値（value）の変更

辞書[キー] = 値

In [ ]:
```
# 変更前のdicAを出力
print("変更前", dicA)

# Aさんの点数を60点に変更
dicA["Aさん"] = 60

# 変更後のdicAを出力
print("変更後", dicA)
```

新しい要素（キーと値のペア）の追加

辞書[新しいキーの名前] = 新しい値

In [ ]:
```
# 追加前のdicAを出力
print("追加前", dicA)

# Fさん70点を追加
dicA["Fさん"] = 70

# 追加後のdicAを出力
print("追加後", dicA)
```

反復（for文）による要素（キーと値）の取得

```
for キー in 辞書:
    値 = 辞書(キー)
```

- 辞書のキーのみがfor文の変数に代入される
- for文の処理の中で、キーを用いて値を取得する

要素の平均値を計算

5人のテストの点数の平均値

In [ ]:
```
dicA = {"Aさん":90, "Bさん":85, "Cさん":75, "Dさん":95, "Eさん":80} # 5人の点数

sum = 0                     # 合計値の初期値を0とする
for key in dicA:            # キーを１個ずつ取り出して変数keyに代入
    sum = sum + dicA[key]   # キーを使ってdicAの値を取り出してsumに加算

# 平均値 = 合計/要素数
ave = sum / len(dicA)  # 平均値をaveという変数に代入
```

```
print("平均値: ", ave)
```

練習

5人の身長を測った結果、佐藤さん170cm、鈴木さん175cm、高橋さん160cm、田中さん155cm、伊藤さん165cmであったとする

(1) 5人の身長を要素とする辞書をdic_highの変数名で作成して、要素数を出力しよう

In [ ]:

(2) 高橋さんの身長に間違いがあったため150cmに変更して、辞書を出力しよう

In [ ]:

(3) 新たに渡辺さん180cmを追加し、辞書を出力しよう

In [ ]:

(4) 身長の平均値を計算し、出力しよう。ただし、for文を使って辞書の要素を取り出すものとする

In [ ]:

解答例

In [ ]:

```
#(1)
dic_high = {"佐藤さん":170, "鈴木さん":175, "高橋さん":160, "田中さん":155, "伊藤さん"
print(dic_high)

#(2)
dic_high["高橋さん"] = 150
print(dic_high)

#(3)
dic_high["渡辺さん"] = 180
print(dic_high)

#(4)
sum = 0
for key in dic_high:
    sum = sum + dic_high[key]
ave = sum / len(dic_high)
print("平均値:", ave)
```

メソッドを用いた辞書の操作

- メソッド とはある処理のまとまりに名前をつけたもので、オブジェクト（変数や値のこと）の後にドット「.」を付けて使用する
  辞書.メソッド名()

全てのキーの集合を取得

- 使用メソッド

  辞書.keys()

In [ ]:
```
#テストの点数がAさん90点、Bさん85点、Cさん75点、Dさん95点、Eさん80点であったとする
dicA = {"Aさん":90, "Bさん":85, "Cさん":75, "Dさん":95, "Eさん":80} # 5人の点数

# 全てのキーの集合を出力
print(dicA.keys())
```

In [ ]:
```
# for文でキーの集合から個別のキーを取得
for key in dicA.keys():
    print(key, end=",")
```

全ての値の集合を取得

- 使用メソッド

  辞書.values()

In [ ]:
```
# 全ての値の集合を出力
print(dicA.values())
```

In [ ]:
```
# for文でキーの集合から個別の値を取得
for val in dicA.values():
    print(val, end=",")
```

全てのキーと値の集合を取得

- 使用メソッド

  辞書.items()

In [ ]:
```
# 全てのキーと値の集合を出力
print(dicA.items())
```

In [ ]:
```
# for文でキーと値の集合から個別のキーと値を取得
for key, val in dicA.items():
    print(key, val)
```

値の取り出し

- 使用メソッド

  辞書.get(キー)

- 辞書.get(キー) と 辞書[キー] の違いは、該当するkeyがなかったときの処理方法
  - 辞書.get(キー) ：指定したキーがないと、None(=「ない」)を返す
  - 辞書[キー] ：指定したキーがないと、エラーになる

In [ ]:
```
# Fさんの値を取り出し
key = "Fさん"
print(key, dicA.get(key))
```

```
# Eさんの値を取り出し
key = "Eさん"
print(key, dicA.get(key))
```

要素（キーと値のペア）の削除

- 使用メソッド
  辞書.pop(キー)

In [ ]:
```
# 削除前の辞書を出力
print("削除前: ", dicA)

# Aさんに該当する要素（キーと値のペア）を削除
dicA.pop('Aさん')

# 削除後の辞書を出力
print("削除後: ", dicA)
```

練習

学生Aから学生Hまでの10人のテストの点数が、80, 70, 70, 60, 60, 90, 80, 80, 70, 60 (点)とする

(1) 学生をキー、点数を値としてdicBという辞書を作成して、表示しよう。

In [ ]:
```

```

(2) 学生Bの点数を表示しよう

In [ ]:
```

```

(3) for文を用いて点数を取り出して足し合わせた後、平均点を求めて表示しよう

In [ ]:
```

```

(4) 新たに受験した学生Iの点数は50点であった。辞書に追加して表示しよう

In [ ]:
```

```

(5) for文を用いて最高点の学生をチェックし表示してみよう

In [ ]:
```

```

解答例

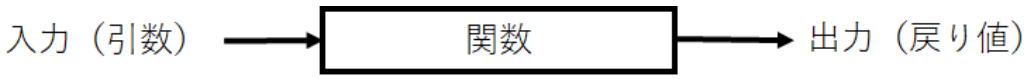
In [ ]:
```
#(1)
dicB = {"学生A":80, "学生B":70, "学生C":70, "学生D":60, "学生E":60, "学生F":90, "学生G
print(dicB)

#(2)
print(dicB["学生B"])

#(3)
sum = 0
for val in dicB.values():
```

```
    sum = sum + val # sum += valでも可
ave = sum / len(dicB)
print("平均値: ", ave)

# (4)
dicB["学生I"] = 50
print(dicB)

# (5)
max_val = 0 # 最高点を保持する変数（初期値0としておく）
for key, val in dicB.items():
    if(val > max_val): # もし、現在の最大点より大きな値が来たら最大点を更新する
        max_val = val  # 最大点の保存
        max_key = key  # 最大点のキーを保存
print("最高点", max_key, max_val)
```

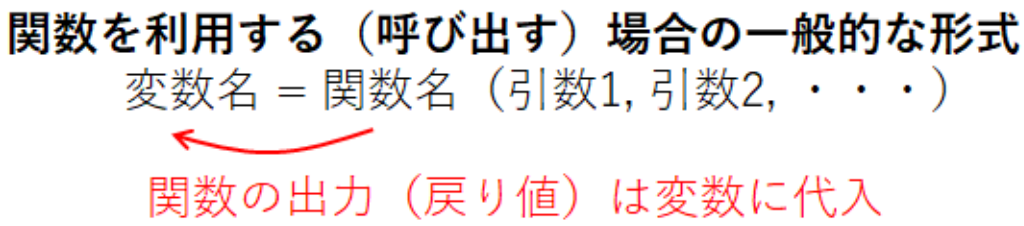
以降、付録

関数

- 様々な処理が機能として１つにまとまっているものを 関数 という
- 関数 は、特定の機能の再利用を可能にする仕組み
- 関数 は、Pythonに最初から準備されている 組み込み関数 （print関数など）と ユーザー定義関数 （自作の関数）に分けられる
- 関数 に、何らかの値を渡すと、その値に応じた何らかの値が結果として返される
  - 渡す値（関数への入力値）のことを 引数 と呼ぶ
  - 返される結果（関数からの出力値）のことを 戻り値 と呼ぶ（戻り値がない関数もある：例えば、print関数）
  - 関数の中でどのような処理が行われているかわからなくでも(ブラックボックス)、入力（引数）と出力（戻り値）が分かっているいればその機能は使える



- 引数 は関数名に続く () 内に記述し、引数が複数あるときにはそれらを , （カンマ）で区切って並べる。また、 関数 から返される結果（ 戻り値 ）を受け取るには、それを変数に代入する。



組み込み関数

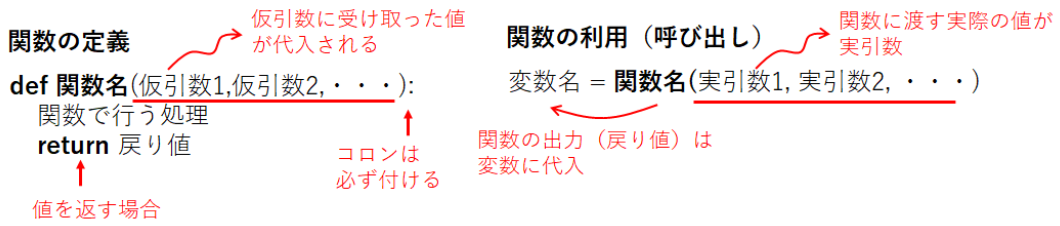
- 組み込み関数は、Pythonで最初から準備されている関数 (公式ドキュメント)
- これまでに出てきた組み込み関数
  - print()
  - len()
  - max()

- min()
- range()
- enumerate()
- sorted()

ユーザー定義関数（自作の関数）

- 独自に定義する関数を **ユーザー定義関数** と呼ぶ
- 関数を定義する理由は、プログラム中で何度も利用する処理を機能として1つに纏めて **再利用** するため
  - 関数を利用せずに何度も同じコードを記述するのは非効率
  - 関数にして必要な場所でそれを呼び出すようにするのが効率的

- 関数の定義方法
  - `def` のあとに関数名を記述、その後 `()` の中に引数を記述、最後に `:`（コロン）を記述する
  - 引数が複数ある場合は「,」（カンマ）で区切る）
  - 関数の定義するときに使用する引数は **仮引数** と呼ぶ。 **仮引数** は、関数で受け取る値のこと。通常、変数に代入して受け取る。
  - 関数を利用（呼び出し）するときに使用する引数は **実引数** と呼ぶ。仮引数は、関数に渡す実際の値のこと
  - `return` の後に戻り値を記述。 `return` がないと関数で処理した結果を取得できない。ただし、戻り値が不要な場合は、 `retrun` の記述は不要
  - 関数で行う処理は、字下げ（インデント）して記述（4つの半角スペース(空白)分）



関数の定義

In [ ]:
```
# 2つの値の差を計算する関数の定義
def calc(num1, num2):
    diff = num1 - num2
    return diff

# 関数の実行
result = calc(3,4)
print("結果", result)

result = calc(5,7)
print("結果", result)
```

仮引数では固定値を設定できる。固定値を設定すると、関数を呼び出す際にその引数を省略できる。
以下の例では、第2引数に固定値を設定している。

In [ ]:
```
# 第2引数に5を設定
```

```
def calc(num1, num2=5):
    diff = num1 - num2
    return diff

# 関数の実行
result = calc(3)
print("結果", result)

result = calc(5)
print("結果", result)
```

＊（アスタリスク）を付けたタプルを用いて、複数の引数を一度に渡すことができる
タプルは、一度設定した要素を変更できないリストのこと。リストは [] で括り、タプルは () （丸括弧）で括る

In [ ]:

```
# 第2引数に5を設定
def calc(num1, num2):
    diff = num1 - num2
    return diff

# 関数の実行
nums   = (3, 4) # 実引数のセットにしたタプル
result = calc(*nums) # 「*タプル名」で実引数のセットを渡す
print("結果", result)

nums   = (5, 7) # 実引数のセットにしたタプル
result = calc(*nums) # 「*タプル名」で実引数のセットを渡す
print("結果", result)
```

キーワード引数

- 複数の引数を持つ関数を使用するとき、引数の順番の間違い対策として 仮引数の変数名 = 値 とし、実引数を与えることができる。これを、 キーワード引数 と呼ぶ。これまでの値のみを実引数として与える場合、 位置引数 と呼ぶ

- キーワード引数を使う場合は、順番が変わっても問題ない

```
def 関数名(仮引数1, 仮引数2):
  関数で行う処理
  return 戻り値
変数名 = 関数名(仮引数2=実引数2, 仮引数1=実引数1)
```

- 位置引数とキーワード引数を混在させて関数を呼び出すこともできるが、その場合は最初に位置引数、その後でキーワード引数を記述する

```
def 関数名(仮引数1, 仮引数2, 仮引数3):
  関数で行う処理
  return 戻り値
変数名 = 関数名(実引数1, 仮引数2=実引数2, 仮引数3=実引数3)
```

In [ ]:

```
# 2つの値の差を計算する関数の定義
def calc(num1, num2):
    diff = num1 - num2
    return diff

# 関数の実行
result = calc(num1=3, num2=4)
print("結果", result)
```

```
result = calc(num2=4, num1=3) # 順番が逆になっても問題なし
print("結果", result)
```

変数のスコープは変数

- スコープとは変数が使える有効範囲のこと
- 変数は **ローカル変数** と **グルーバル変数** に分けられる
  - **ローカル変数** は、関数内で定義され、その関数内でのみ使用できる変数
  - **グローバル変数** は、関数外で定義され、どこからでも使用できる変数
- **グローバル変数** と同一の変数名を関数内でも使用する場合
  - 関数内で値を扱うだけなら **グローバル変数** 扱い
  - 関数内で値を代入した場合は **ローカル変数** に切り替えられ、 **グローバル変数** と同一の変数名でも区別される
  - 関数の仮引数として用いられた変数は **ローカル変数** となり、 **グローバル変数** と同一の変数名でも区別される

In [ ]:
```
def showNum():            # 関数の定義（引数も戻り値もなし）
    numIn = 456           # ローカル変数(関数内だけで使用される変数)
    print(numIn, numOut) # 関数外から定義されたグルーバル変数のnumOutは関数内でも使用

numOut = 123              # グローバル変数
showNum()                 # 関数呼び出し（引数も戻り値もなし）
```

関数内でグローバル変数に値を代入すると、ローカル変数に切り替えられる。
そして、同じ変数名でもグローバル変数とローカル変数は区別される

In [ ]:
```
def setLocal():
    num = 456          # 値の代入によりグローバル変数からローカル変数に切り替わったnum
    print("関数内:", num)

num = 123  # グルーバル変数のnum
setLocal()
print("関数外:", num)
```

関数内でグローバル変数の値を変更する場合は、 `global` を用いてその変数をグローバル変数として扱うように宣言する

In [ ]:
```
def setGlobal():
    global num  # 関数内でnumをグルーバル変数として扱うことを宣言
    num = 456
    print("関数内:", num)

num = 123   # グルーバル変数のnum
setGlobal()
print("関数外:", num)
```

練習

- 球の体積を求める関数をvolumeという関数名で作成してみよう
  - 引数は半径として変数名はrとする
  - 円周率は3.14とする
  - 体積を代入する変数名はvとする

    - 球の体積の公式：$3/4 \times \pi \times r^3$
- 関数の作成ができたら、関数を使って半径5のときの球の体積を計算して表示してみよう
- 各変数が **グローバル変数** と **ローカル変数** かを頭の中で整理してみよう

In [ ]:
```
```

解答例

In [ ]:
```
def volume(r):                  #r:ローカル変数
    v = 4/3 * 3.14 * r**3       #v:ローカル変数
    return v

r = 3                           # r:グルーバル変数
v = volume(r)                   # v:グルーバル変数
print("球の体積", v)
```

無名関数（lambda(ラムダ)式関数）

- 無名関数(lambda式関数)は、その名のとおりで関数名を定義しなくてもよい関数のこと
- わざわざ関数名を付けるまでもないが、簡単な処理を実行したいとき、lambda式を使用する

  `lambda 引数: 処理`
- 実用上は、if文と組み合わせると便利

通常の関数の定義

In [ ]:
```
# 引数値を半分にする関数を作成
def half_value(x):
    return x / 2


# 引数に2を渡して、関数を実行
result = half_value(2)

# 結果の出力
print(result)
```

lamda式関数の定義

In [ ]:
```
# lambda式関数の定義
half_value_v2 = lambda x: x / 2

# lambda式関数の実行
result = half_value_v2(2)

# 結果の確認
print(result)
```

lamda式関数とif文の組み合わせ

- 引数が2で割れるとき → そのまま引数を返す
- 引数が2で割れないとき → 2で割り切れないことを知らせる

通常の関数の定義

In [ ]:
```
def f(x):
    if x % 2 == 0:
        return x
    else:
        return '2で割り切れません'

result = f(2)
print(result)
```

lambda式関数の定義

In [ ]:
```
f2 = lambda x: x if x % 2 == 0 else '2で割り切れません'
result = f2(3)
print(result)
```

リストの発展的内容

リストの内包表記

- リスト内の全ての要素に5を足したり、3倍したい場合、 `list+5` や `list*3` のような処理はできないので、反復処理(for文)で要素を１つずつ取り出して5を足したり、3倍する必要がある
- 内包表記 というPython特有の表記方法を用いるとリストに対する反復処理を簡潔に記述することができる
- 内包表記 は、以下のように記述する

  新たなリスト = [要素への処理 for 要素 in リスト]

  リスト内の要素を1つ1つ取り出して、要素への処理を実行した上で新しいリストを作成する

全要素に特定の処理を行って新たなリストを作成

内包表記を使わない場合

In [ ]:
```
listA = [90, 85, 75, 95, 80] # 5人の点数


# 全員に3点の加点
listE = []                # 加点後の値を格納する空リストを準備
for i in listA:           # listAの要素をfor文で取り出し
    listE.append(i + 3) # appnedメソッドで加点した値を追加

# 加点前の要素を表示
print("加点前:", listA)

# 加点後の要素を表示
print("加点後:", listE)
```

内包表記を使う場合

In [ ]:
```
listA = [90, 85, 75, 95, 80] # 5人の点数
```

```
# 全員に3点の加点
listE = [i + 3 for i in listA]  # 内包表記

# 加点前の要素を表示
print("加点前:", listA)

# 加点後の要素を表示
print("加点後:", listE)
```

条件に合う要素を抽出して新たなリストを作成

- 条件を満たす要素（条件式がTrueとなる要素）のみ抽出したリストを生成

  [変数名 for 変数名 in 元のリスト if 条件式]

- 条件を満たさない要素（条件式がFalseとなる要素）のみ抽出したリストを生成

  [変数名 for 変数名 in 元のリスト if not 条件式]

In [ ]:

```
listA = [90, 85, 75, 95, 80] # 5人の点数

#-----------------------------------------
#  80以上の要素だけを抽出したリストを作成
#-----------------------------------------
# パターン1: 80以上なら
listF = [i for i in listA if i >= 80]
print("1: ", listF)

# パターン2: 80未満でないなら
listG = [i for i in listA if not i < 80]
print("2: ", listG)

#-----------------------------------------------------
#  偶数（2で割り切れる）の要素だけを抽出したリストを作成
#-----------------------------------------------------
# パターン1: 偶数なら
listF = [i for i in listA if i % 2 == 0]
print("3: ", listF)

# パターン2: 奇数でないなら
listG = [i for i in listA if not i % 2 != 0]
print("4: ", listG)

#-------------------------------------------------------------------
#  80以上または偶数（2で割り切れる）の要素だけを抽出したリストを作成
#-------------------------------------------------------------------
listF = [i for i in listA if i >= 80 or i % 2 == 0]
print("5: ", listF)

#-------------------------------------------------------------------
#  80以上かつは偶数（2で割り切れる）の要素だけを抽出したリストを作成
#-------------------------------------------------------------------
listF = [i for i in listA if i >= 5 and i % 2 == 0]
print("6: ", listF)
```

条件を満たす要素を変換または変換する

- 内包表記と三項演算子を組み合わせる。以下の () （丸括弧）内が三項演算子と呼ばれる。

  [(真の値 if 条件式 else 偽の値) for 任意の変数名 in 元のリスト]

() がなくてもよく、以下のように記述するのが一般的

[真の値 if 条件式 else 偽の値 for 任意の変数名 in 元のリスト]

In [ ]:
```
listA = [90, 85, 75, 95, 80] # 5人の点数

#--------------------------------------
#  90以上の要素は100に置換したリストを生成
#--------------------------------------
listH = [100 if i >= 90 else i for i in listA]
print("1: ", listH)

#--------------------------------------
#  偶数の要素には3を足したリストを生成
#--------------------------------------
listH = [i+3 if i % 2 == 0 else i for i in listA]
print("2: ", listH)

#------------------------------------------------------------
#  偶数の要素は3を足し、それ以外は2を足したたリストを生成
#------------------------------------------------------------
listH = [i+3 if i % 2 == 0 else i+2 for i in listA]
print("3: ", listH)
```

リストの応用例

- wikipediaのコンテンツを効率的に取得するプログラムを作成してみよう。また、これを使って新しいサービスが作れないかを考えてみよう。
- `wikipedia` というライブラリを使用する
  - ライブラリとは、よく使う機能・関数をまとめて、簡単に使えるようにしたもの
  - 関数で学習したように、ライブラリの中身のプログラムがブラックボックスであっても、入力と出力がわかっていれば簡単に実装し、実行することができる。
  - Pythonに便利な機能を持つライブラリが多数ある。

ライブラリのインストール

Google Colaboratory (略してGoogle Colab)を使用する場合

- 以下セルのコメントアウト//を削除して実行

Anacondaを使用する場合

- Anaconda promptで、 `conda install -c conda-forge wikipedia` を実行
- 参考

In [ ]:
```
# ! pip install wikipedia
```

単語の検索

- 東京成徳大学 に関連する単語を検索
- 関連する単語は リスト で出力される

In [ ]:
```
import wikipedia # ライブラリのインポート

wikipedia.set_lang('ja') # 日本語版wikipediaの指定
words = wikipedia.search("東京成徳大学") # wikipedia内の検索（引数に検索ワードを記述）
```

```
print(words) # 検索結果の表示
print() # 改行用
print("検索された単語数:", len(words)) # len関数を使って個数を表示
```

関連単語のurlを表示

```
検索結果 = wikipedia.page(検索単語).url
```

In [ ]:

```
for i in words: # 繰り返し文で関連単語を１つずつ取り出す
    url = wikipedia.page(i).url # urlを取得
    print(url) # urlの表示
```

関連単語の内容表示

```
検索結果 = wikipedia.page(検索単語).content
```

In [ ]:

```
# インデックス0番目のwikipediaの内容を表示
content = wikipedia.page(words[0]).content
print(content)
```