# スクリプト言語プログラミング
# Pythonによる数値解析

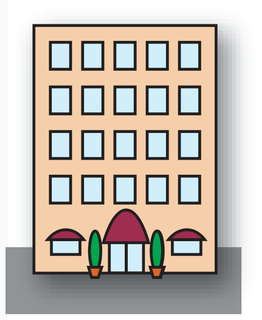
第６回講義資料

箕原辰夫

# 配列・リスト

- リスト・タプル・辞書・集合には１つの変数の中に複数のデータ値を入れることができる。

一戸建て（普通の変数）には一世帯



マンション（配列変数）には複数の世帯

- 通常の変数をスカラー変数、リストや配列をベクトル変数と呼ぶこともある。

# リストの記法

- 基本的には、リストを示す[ ]で値やオブジェクトを囲むだけ
- 例： xlist = [ ] # 空のリストが作成される

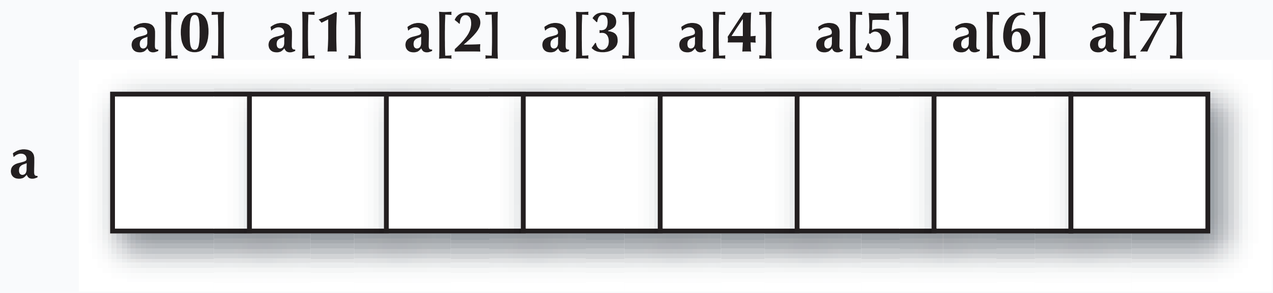
- 組込みのlist( ) 関数を使ってもリストを作成することができる
- 例： xlist = list( )

# リストの確保と初期値

- リストについては、各要素の初期値を指定しないと、サイズ分確保されない。
- 例： xlist = [ 1, 2, 4, 9 ] # 4つのサイズのリストが作られる
- 0クリアされたリストを作りたい場合は、次のようにリストに掛け算を行なって作る
- 例：
  - a = [ 0 ] * 8 # サイズは、8となる
  - b =[ 0.0 ] * 20 # サイズは、20となる

# リストの要素へのアクセス

- 確保されたリストには、各データを参照するための部屋番号がついている。これをインデックス（指標・添え字）と呼ぶ。




- インデックスは、「0〜サイズ-1」の範囲の整数、マイナスの場合には、「-1〜-サイズ」の範囲の整数に限られる。範囲外だと実行時エラー（例外）が発生する。

# 要素への代入・参照

- リストの要素は、通常の変数のように用いれる。
- 要素を参照するには、次のような書式を用いる（インデックスは整数の式である）。
  - 変数名[ インデックス ]
  - 例： c.create_line( x[ 1 ], y[ 1 ], x[ 2 ], y[ 2 ] );
  - z = x[ 4 ] * 3

- 要素に個々に代入するには次のような書式を用いる
  - 変数名[ インデックス ] = 式
  - 例： a[ 3 ] = 10

- スライスで代入する場合は、元の要素のその範囲の個数に関係なく、代入ができる
  - 変数名[ スライス式 ] = リスト
  - 例：a[ 3:5 ] = [ 4, 5, 6 ]

    a = list( range( 1, 8 ) )
    a[2:4] = [ 8, 9, 10, 11]　⇒ [1, 2, 8, 9, 10, 11, 5, 6, 7]

    a = [ 12, 23, 34, 45, 56]
    a[2:4]=[]　⇒ [12, 23, 56]

# インデックスの式

- インデックスは整数の式なので、変数や計算するものであっても良い。
  - x = 3
  - 例： a[ x ] = 5
  - 例： a[ x+3 ] = 10

- リストの要素を参照する場合は、まずはそのインデックスの式から評価される。
  - 例： a[ a[ x+2 ] ] = 20
    - x = 3;  a[ 5 ] = 4　→　a[ a[ x+2 ] ] ⇒ a[ a[ 5 ] ] ⇒ a[ 4 ] = 20

# リストの要素への代入

- 要素へ代入するとき、代入文の省略形も使える
  - a[ 4 ] += 67
  - a[ 4 ] =  a[ 4 ] + 67

  - a[ 5 ] += 1
  - a[ 5 ] = a[ 5 ] + 1

- スライスで代入した場合には、その要素の値が替わる訳ではなく、リストが途中に追加される
  - a = [ 12, 23, 34, 45, 56]
  - a[2:4]+=[89,99]　⇒　[12, 23, 34, 45, 89, 99, 56]

# リストと繰返し

- リストは繰返しを用いて操作する。
- 代入をしてみる（アプリケーションで）
  - インデックスの値を使う
  - 漸化式を使う
  - 乱数を使う
    - int ( random.random( ) * d ) + a
    - [ a, a+d )の間の整数を発生させる

# 乱数ライブラリ

- **import** randomで、パッケージを使う準備をする
- 以下の関数が使える。一様乱数の場合
  - random.random() …0 ～ 1未満の実数
  - random.randint( a, b )…a以上b以下の整数
  - random.randrange( a, b, d )…a以上b未満の整数で、間隔dで発生する
  - random.uniform( a, b )…a以上b以下の実数
  - random.gauss( m, d )…平均m、標準偏差がdの正規分布乱数

# リストへの初期値代入

- 代入時に各要素の初期値が代入できる
  - リスト名 = [ 初期値, .... , 初期値 ]
  - 例：a = [ 10, 7, 4, 6, 3 ]
    - 等価：

      a =[ 0 ] * 5
      a[ 0 ]=10; a[1]=7; a[2]=4; a[3]=6; a[4]=3

- list関数とrange( )関数を利用してリストを作成することができる
  - 例：nlist = list( range( 10 ) )

# リストをfor文で作る

- for文を使って初期化されたリストを作成することができる
  - 例：[ x **for** x **in** range( 1, 10 ) ] ⇒

    [1, 2, 3, 4, 5, 6, 7, 8, 9]

- このときにif文やif式も利用することが可能
  - 例：[ x **for** x **in** range( 1, 10 ) **if** x % 2 == 0 ] ⇒

    [2, 4, 6, 8]
  - 例：a = [ n **if** n % 2 == 0 **else** n*10 for n  **in** range( 1, 11 ) ]

# リストのサイズ

- リストはlen関数を使えば、リストのサイズが計算される。
  - len( リスト名 )
  - 例：len( a )→リストaのサイズが整数で求まる

- Javaの配列では、配列aの属性として、a.lengthがあるので、この書式を使えば、リストのサイズがどのようなものでも対応できる。
- C/C++にはないので不便だが、以下の書式で計算できる
  - **sizeof**( 配列名 ) / 4 を使う(整数が4バイトのとき)

# リストへの演算

- リスト + リスト
  - リストの追加になる
  - 例： [ 1, 2 ] + [ 3, 4 ] ⇒ [ 1, 2, 3, 4 ]
- リスト * 整数
  - そのリストが、整数回分繋がったものになる
  - 整数が0以下だと空のリストが作成される
  - 例：["a","b" ] * 4 ⇒ [ "a","b", "a","b", "a","b", "a","b" ]
- 値 **in** リスト
  - その値が、リストの要素として含まれていればTrue
- 値 **not in** リスト
  - その値が、リストの要素として含まれていなければTrue

# リストの走査（総和と平均）

- 総和と平均

```
summ = 0
for n in alist:
    summ += n

average = summ / len( alist )
```

# 最大値・最小値

- 最大値・最小値

```
maxi, mini = 0, 0
for i, ele  in enumerate( alist )
  if  alist[ maxi ] < ele : maxi = i
  if  alist[ mini ] > ele : mini = i # miniは不等号が逆
```

maxi →最大値のある要素のインデックス

alist[ maxi ] →最大値

# 検索とカウント

- 特定の値の場所を検索、何個あるか

```
target = int( input( "見つけ出す値: " ) )
alist  = [ 5, 2, 565, 222, 111, 344, 22, 99, 348, 22, 2 ]

index, count = 0, 0
for i, n in enumerate( alist ):
    if  n == target : count+=1; index = i

if  count > 0 :
    print( target  +  " は最後に " + index + \
    " で見つかりました " + count + “ 個あります”)
else:
    print( target  +  " はありません。 " )
```

# 頻度表

- 誕生月の頻度を求める

```
birth  = [ 1, 3, 5, 2, 8, 11, 4, … , 6, 7, 12, 4, 2 ]
month = [ 0 ] * 13

for index in range( len( birth ) ) :
    month[  birth[ index ]  ] += 1

for m in range( 1, len( month ) ):
    print( m  +  "月の誕生日の人は" + month[ m ] +"人いました" )
```

# リストの並べ替え

- ランダムにシャッフルする

```
a  =[ index + 1 for index in range( 100 ) ]

for n in range( len( a ) * 3 ) :  # 何回繰り返すかは、長さの2倍から3倍以上
    index1 = random.randint( 0, len( a )-1 )
    index2 = radom.random( 0, len( a )-1 )
    a[  index1 ], a[ index2 ] = a[ index2 ], a[  index1 ]
```

- ランダムにシャッフルする

```
import random
random.shuffle( nlist )
```

# リストの最大値を再帰で

- リストのサイズが１個だったら、
  - その要素が最大値であるとして返す
- リストのサイズが２個以上だったら、
  - サイズを１つ減らして、最大値を求める
  - 求められた最大値と、最後の要素の値を比較して、大きい方を返す

# リストのソート

- ソート・ソーティング・整列…順番に並び替えること
- 直接選択法
  - 小さいものを選び、入れ替えをする

```
for i in range( len(a)-1 ) :
    mini = i
    for j in range( i+1, len( a ) ):
        if  a[ mini ] > a[ j ]  : mini = j

    a[ i ], a[ mini ] =  a[ mini ], a[ i ]
```

# 直接選択法でのソーティング

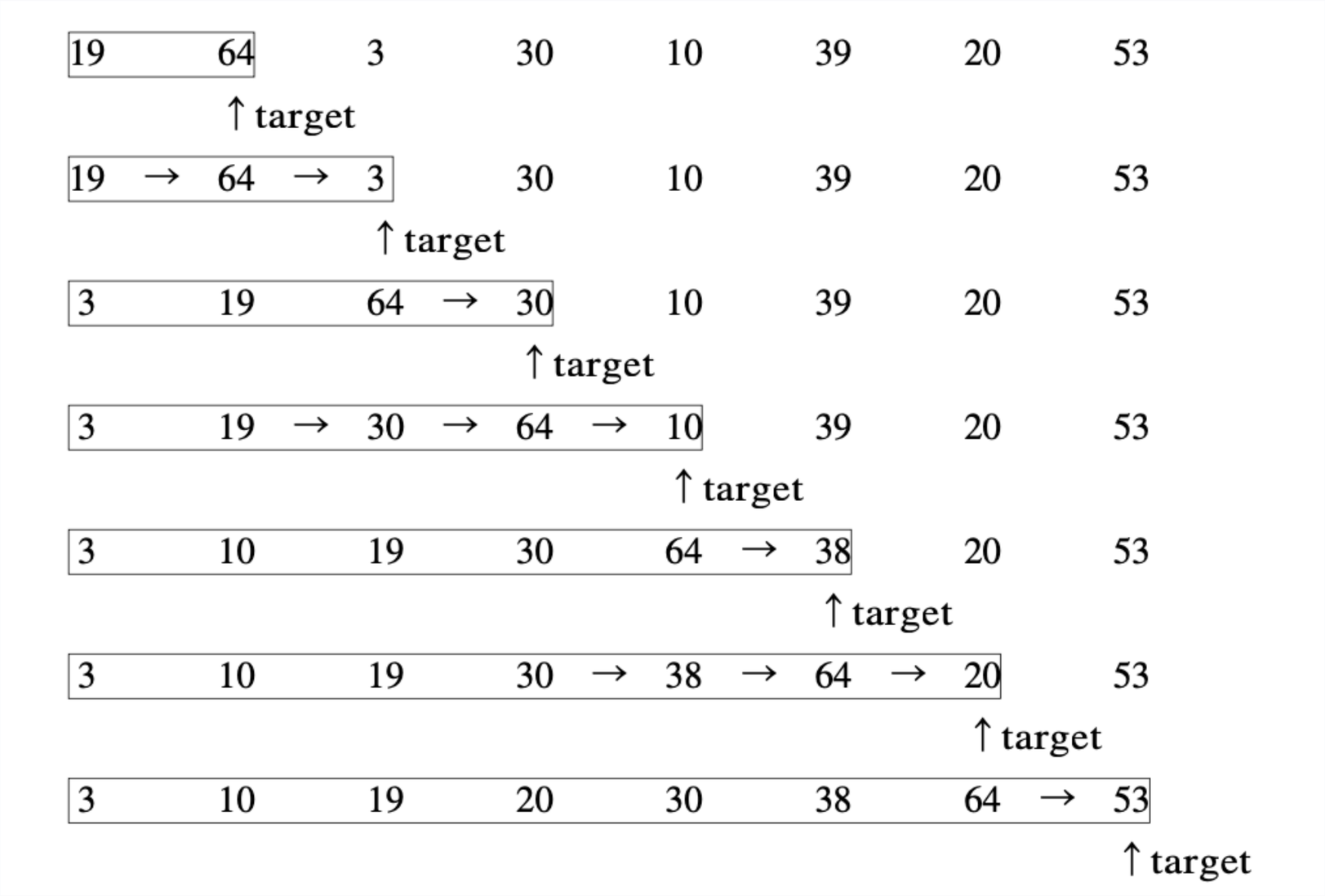
直接選択法のソーティングの様子：

```
19   64   3   30   10   39   20   53
↑i        ↑min
3   64   19   30   10   39   20   53
     ↑i             ↑min
3   10   19   30   64   39   20   53
          ↑i, min
3   10   19   30   64   39   20   53
               ↑i              ↑min
3   10   19   20   64   39   30   53
                    ↑i         ↑min
3   10   19   20   30   39   64   53
                         ↑i, min
3   10   19   20   30   39   64   53
                              ↑i   ↑min
```

# 直接插入法（非再帰版）

```
def sisort( alist ):
    for i in range( 1, len( alist ) ):
        target = alist[ i ]
        j = i-1
        while j >= 0:
            if target < alist[ j ]:
                alist[ j+1 ] = alist[ j ]
            else:
                break
            j -= 1
        alist[ j+1 ] = target
```

# 直接挿入ソートの例


19 64 3 30 10 39 20 53
↑target
19 → 64 → 3 30 10 39 20 53
↑target
3 19 64 → 30 10 39 20 53
↑target
3 19 → 30 → 64 → 10 39 20 53
↑target
3 10 19 30 64 → 38 20 53
↑target
3 10 19 30 → 38 → 64 → 20 53
↑target
3 10 19 20 30 38 64 → 53
↑target

# 直接交換法（バブルソート）

- 直接交換法のソートの関数

```
def bubbleSort( a ):
    for i in range( len( a ) ):
        for  j in range(len( a )-1,  i-1, -1 ) {
            if a[ j-1 ] > a[ j ]:
                a[ j-1 ], a[ j ] = a[ j ], a[ j-1 ]
```

# 直接交換法でのソーティング

| | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 19 | ← | 64 | ← | 3 | | 30 | ← | 10 | | 39 | ← | 20 | | 53 |
| 3 | | 19 | ← | 64 | ← | 10 | | 30 | ← | 20 | | 39 | | 53 |
| 3 | | 10 | | 19 | | 64 | ← | 20 | | 30 | | 39 | | 53 |
| 3 | | 10 | | 19 | | 20 | | 64 | ← | 30 | | 39 | | 53 |
| 3 | | 10 | | 19 | | 20 | | 30 | | 64 | ← | 39 | | 53 |
| 3 | | 10 | | 19 | | 20 | | 30 | | 39 | | 64 | ← | 53 |
| 3 | | 10 | | 19 | | 20 | | 30 | | 39 | | 53 | | 64 |

# 計算量$O$

- 計算量をオミクロンの大文字（$O$）の関数で表わす。
- n個のデータがある場合

  $O(1)$ … データの個数に依存しない

  $O(\log n)$ … 高速なアルゴリズム

  $O(n)$ … 高速な部類

  $O(n \log n)$ … やや高速

  $O(n^2)$ … 低速
- 実際にn個のデータがあった場合、処理の回数が

  $2n^2+n+1$回だった場合

  →係数は無視する・一番次数の高いものを選ぶ

  →よって、$O(n^2)$

# ソートアルゴリズムの分類

| | 選択系 | 交換系 | 挿入系 | 併合系 |
|---|---|---|---|---|
| 単純 | 単純選択法 | 単純交換法<br>（バブルソート） | 単純挿入法 | |
| 修正 | 安定化単純選択法 | 改良バブルソート<br>シェーカーソート | 2分挿入法 | |
| 高速 | ヒープソート | コームソート<br>クイックソート | シェルソート | マージソート |

# 最良と最悪の場合

- データの値によって、アルゴリズムが高速になったり、低速になったりするものがある。
- データの値によって、アルゴリズムが極度に低速になるものは使用方法や実現方法について注意しなければならない。
- 例：Quick Sort…整列するのと逆順に並んでいると、ほぼ$O(n^2)$になってしまう。
- 例：直接挿入ソート…通常だと$O(n^2)$の計算量だが、データがほとんど整列されている状態で、整列を行なうと、$O(n)$で処理が終わってしまう

# バブル（単純交換）ソートの問題点

- 隣り合った要素しか比較しないので、要素は1回の交換で、隣にしか移動しない
  - これを改良したのが、コーム（Comb）ソート
  - 間隔を変えながら（最初は大きく、最後は小さく）、交換を行なう
- 配列の先頭の方の大きい値が、なかなか後方に移動しない
  - これを改良したのが、シェーカー（Shaker）ソート
  - インデックスの小さい方から、大きい値を後に移動させる回を設ける
- 整列されている部分でも無駄な比較が行なわれる
  - これを改良したのが、改良バブルソート
  - 整列されているところまできたら、その回は終了させる

# ソートの性能比較表

| | 平均 | 整列順（最良） | 逆順（最悪） | 安定 | メモリ使用量 |
|---|---|---|---|---|---|
| 単純選択ソート | $O(n^2)$ | $O(n^2)$ | $O(n^2)$ | △ | $O(1)$ |
| ヒープソート | $O(n \log n)$ | $O(n)$ | $O(n \log n)$ | × | $O(1)$ |
| 修正単純交換（バブル）ソート | $O(n^2)$ | $O(n)$ | $O(n^2)$ | ○ | $O(1)$ |
| シェーカーソート | $O(n^2)$ | $O(n)$ | $O(n^2)$ | ○ | $O(1)$ |
| コームソート | $O(n \log n)$ | $O(n)$ | $O(n^2)$ | × | $O(1)$ |
| クイックソート | $O(n \log n)$ | $O(n \log n)$ | $O(n^2)$ | × | $O(\log n)$ |
| 単純挿入ソート | $O(n^2)$ | $O(n)$ | $O(n^2)$ | ○ | $O(1)$ |
| シェルソート | $O(n^{1.25})$ | $O(n \log n)$ | $O(n^2)$ | × | $O(1)$ |
| マージソート | $O(n \log n)$ | $O(n)$ | $O(n \log n)$ | ○ | $O(n)$ |

# コームソート・シェルソートの間隔

- ここでいう間隔とは、インデックスの値の差を示す
- コームソートの場合
  - 間隔の収縮率を1.3にすると一番効率が良いことがわかっている、間隔は、次のような形で小さくしていく

    ① 9, 6, 4, 3, 2, 1

    ② 10, 7, 5, 3, 2, 1

    ③ 11, 8, 6, 4, 3, 2, 1
  - 11を使うものは、コームソート11と呼ばれる
- シェルソートの場合
  - …, 364, 121,40, 13, 4, 1 ($h_{k-1}=3h_k+1, h_t=1$)　この場合、$O(n^{1.25})$
  - …, 63,31,15,7,3, 1 (($h_{k-1}=2h_k+1, h_t=1$)　この場合、$O(n^{1.5})$

# スライス

- リスト[ 最初 : 終わりの次 ]
  - 部分的なリストが生成される
  - 例：alist[ 4: 7 ]
- リスト[ : 終わりの次 ]
  - 最初の要素の位置は、0と仮定される
  - 例：alist[ 7 ]
- リスト[ 最初: ]
  - 指定された最初の位置から、最後までになる
  - 例：alist[ 4: ]
- リスト[ -インデックス ]
  - 最後の要素の次から、順次インデックスの値が引かれていく
  - 例：alist[ -5 ],  alist[ -5: ]

# スライスによる要素の代入

- スライスで一定の部分に代入ができる
  - a[ 1:3 ] = [2, 3]
- 同じサイズでなくても構わない、自動的に拡張される
  - a[ 1:2 ] = [ 2, 3 ]

# 飛び飛びのスライス

- リスト[ 最初: 最後の次: 間隔 ]
  - 最初から最後までの間で、間隔が空いたサブリストが作成される
- リスト[ : : 間隔 ]
  - 間隔が空いたサブリストが作成される

# リストに適用できる組込み関数

- all( 論理値のリスト )…すべての要素がTrueのときだけTrue
- any( 論理値のリスト )…すべての要素がFalseのときだけFalse
- enumerate( リスト, start=0 )…インデックスと要素の対のシーケンスを発生させる
- len( リスト )…リストの長さを求める
- list( リスト )…新たなリストを生成する
- max( リスト )…リスト中の最大値を求める
- min( リスト )…リスト中の最小値を求める
- sorted( リスト )…ソートされた新たなリストを返す
- sum( リスト [, 初期値 ] )…要素の総和を求める
- reversed( リスト )…反対順になった新たなリストを返す
- zip( リスト, リスト, … )…各リストの先頭からの要素を順番にまとめたタプルから構成されるシーケンスを発生させる

# リストの追加・削除・検索

- リスト.append( value )…最後に要素を追加
- リスト.remove( value )…該当する値の要素を削除
- リスト.insert( i, value )…i番目に、その値をもつ要素を挿入
- リスト.sort()…そのリスト自体をソートする
- リスト.reverse()…要素の順番を反転させる
- リスト.extend( リスト )…リストの結合（+演算と同じ）
- リスト.clear( )…空リストにする

- リスト.index( value )…その値を持つ要素のインデックスを返す
- リスト.count( value )…その値を持つ要素が何個あるか返す
- リスト.copy( )…浅いコピーで、コピーのリストを作って返す

# del文による要素の削除

- del s[ i ]
  - i番目の要素を削除する
- del s[i:j]
  - i番目からj-1番目までの要素を削除する
- del s[i:j:k]
  - i番目から、kずつインデックスを増やしながら、j-1番目までの要素を削除する

# リストを使った高階関数

- filter( 関数, リスト )
  - 関数は、引数を１つ必要とし、論理値を返すもの
  - 返された論理値がTrueの要素だけを持つリストが生成されるようなオブジェクトが返される
- map( 関数, リスト, … )
  - 関数は、引数を１つ必要とし、何らかの値を返すもの
  - 返された値から構成されるリストが生成されるようなオブジェクトが返される
  - リストが複数ある場合は、関数は、その個数分だけの引数を必要とする
- functools.reduce( 関数, リスト )
  - 関数は、引数を２つ必要とし、何らかの値を返すもの（１つの引数に、演算結果が保存される形になる）
  - リストの各要素に引数が適用された値が返される

    **import** functoolsが必要

# キー関数を利用した走査

- sorted( リスト, キー関数 )
- max( リスト, キー関数 )
- min( リスト, キー関数 )
  - キー関数を使って、比較すべき要素の値を指定することができる

- 利用例：

```
nlist = [ "111", 5.12, 67, True, 143 ]  # 異種の型の要素を持つリスト
print( nlist )
print( max( nlist, key=int ) )
print( min( nlist, key=lambda n: float( n ) ** 0.5 ) )

wlist = ["hilbert", "iwan", "Anne", "John", "cathy"]
print(sorted( wlist, key=str.lower) )
```

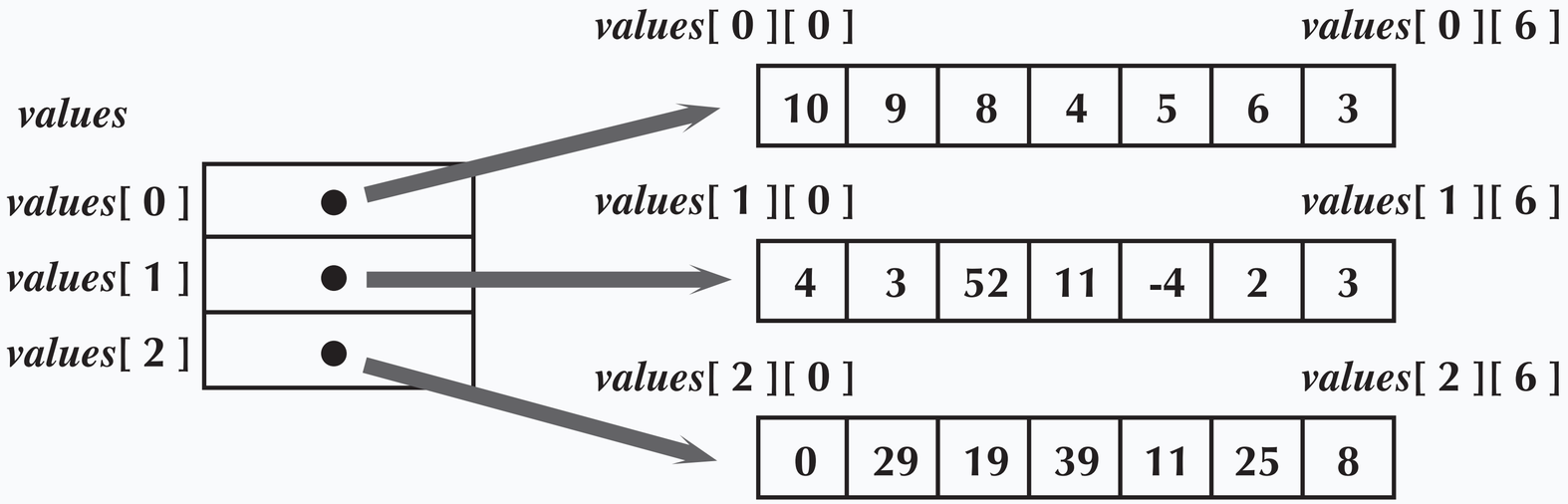
# filterを利用したエラトステネスの篩

- 2から順次に昇順のリストを用意する
- 以下をリストが空になるまで繰り返す
  - リストの最初の要素を、素数リストに追加する
  - リストの最初の要素で割り切れるものをリストから除く
- プログラム記述例

```
nlist = list( range( 2, 101 ) )
primelist = []
while len( nlist ) > 0:
    prime = nlist[ 0 ]
    primelist.append( prime )
    nlist = list( filter( lambda n: not n%prime==0, nlist) )
print(primelist )
```

# ２次元リスト（入れ子型）

- パズルなどのゲームやデータの統計を行なうには、必要



# 2次元リストの宣言

- 空の２次元リストの例
  - values = [ [ ] ]
- 領域を確保した２次元リストの確保
  - values = [ [0] * 10 ] * 10 # ２次元目が共有されてしまう
  - values = [ [ ] ] * 10　　# １次元目だけを確保する
- 共有されない２次元リスト
  - values = [ [0 **for** n **in** range( 10 ) ] **for** m **in** range( 10 ) ]
  - values = [ [ 0 ] * 10 **for** _ **in** range( 10 ) ]

# 2次元リストの初期値代入

- 初期値代入の例
  - values = [ \
        [ 10, 9, 8, 4, 5, 6, 3 ], \
        [ 4, 3, 52, 11, -4, 2, 3 ], \
        [ 0, 29, 19, 39, 11, 25, 8 ] \
    ]
- 長さが違う２次元リスト
  - values = [ [ 10, 20 ],  [ 7 ],  [ 47,  27, 18 ], [ 8, 2 ] ]

# ２次元リストの長さ

- 例題のリスト
  - matrix=[ [ 0 ] * 5 ] * 10
- len( リスト名 ) … 1 次元目のリストの長さが求まる
  - len( matrix) ⇒ 10
- len( リスト名[ インデックス ] ) … ２次元目のリストの長さ
  - len( matrix[ 2 ] ) ⇒ 5
- ２次元目の長さが異なるような場合にも対処する必要

# 繰返しで２次元リストの要素を参照

- C++/Java/C#的な記述の仕方（代入するときなど）

```
for i in range( len( matrix ) ):
    for j in range( len( matrix[ i ] ) ):
        matrix[ i ][ j ] = i * j
```

- Python記述の仕方（参照するだけの場合）

```
for rows in matrix:
    for cell in rows:
        print( "%d  " % ( cell ) )
```

# 2次元リストの代入とコピー

- 代入は、共有される
  - a = [ [ 1, 2, 3], [ 4, 5, 6], [7, 8 ] ]
  - a = b # リストは同一のものとなる a **is** b → True
- copyモジュールにある→**import** copyが必要
- copy関数は、浅いコピーなので2次元目は共有されてしまう。
- deepcopy関数は、2次元目もコピーが作られる
- 例：

```
import copy
amat = [ [1, 2, 3], [5, 6, 8], [4, 9, 7] ]
bmat = copy.copy( amat )  # bmat = amat.copy( )    amat is bmat  → False
cmat = copy.deepcopy( amat )

cmat[ 0 ][ 0 ] = 100       # amatの要素には何も影響がない
bmat[ 0 ][ 0 ] = 300       # amat[ 0 ][ 0 ]も300になってしまう
```

# 2次元リストの平坦化

- ２次元リストを１次元リストにする方法
  - mat = [ [1,2,3], [4,5,6], [7,8] ]を
  - [1, 2, 3, 4, 5, 6, 7, 8]にする
- reduce高階関数を使う
  - **import** functools
  - alist = functools.reduce( lambda a, b: a+b, mat )
- sumで、初期結果を空リストにする
  - sum( mat, [ ] )

# 1次元リストの2次元リスト化

- 組込み関数などはないので、自分で関数などを作成する必要がある
- 作成例：

```
def nesting( alist, n ):
    result = [ ]
    for i in range( 0, len(alist), n ):
        result.append( alist[i:i+n] )
    return result
```

# 2次元リストの走査

- max, min, sortedのキー関数を使って、比較対象を選ぶようにする
- 利用例：

```
nestedlist = [ [7, 3], [4, 5], [3, 8], [2, 9] ]
print( sorted( nestedlist, key=lambda x:x[0] ) )
print( max( nestedlist, key=lambda x:x[1] ) )
print( min( nestedlist, key=lambda x:x[0] ) )
```

# 文字列とリストの変換

- list( 文字列 )… 1 文字ずつ分解したリストができる
- str( リスト )…print関数で表示されるような文字列ができる

- 文字列.split( 区切り文字 )…文字列を区切り文字で区切って、リストにする、区切り文字がないと空白やタブなどで区切る形になる
- 区切り文字.join( 文字列のリスト )…リストの各要素の文字列を区切り文字で繋げて文字列にする

# ファイルからの読込み

- f = open( ファイル名, モード )
  - モードは、"r"…読み、"w"…新規作成・上書き
  - "r+"…更新用に読み、"w+"…更新用に書込み、
  - "a"…追加書込み, "b"…バイナリモード
  - encoding="Shift_JIS" … Shift JIS(cp932)のテキストファイルを読み込んで変換してくれる
  - 標準は、encoding="UTF-8"
- f.read( )…テキスト全体を読み込む
- f.readlines( )…テキストを改行で区切って 1 行ずつのリストとして読み込む
- f.readline( )…テキストを 1 行ずつ読み込む
- f.close( )…ファイルへのアクセス終了

# ファイルへの書込み

- f.seek( バイトのオフセット )
  f.seek( バイトのオフセット, 起点 )
- ファイル内の任意の場所に書込み（読込み）位置を移動する
- 起点は、0…ファイルの先頭、1…現在の位置、2…ファイルの最後、起点が1の場合は、オフセットの値は負の値でも良い、2の場合は通常負の値
- f.write( データ )
- ファイルの現在の位置に、データを書き込む

# CSVファイルの読み書き

- csvのモジュールだと１次元リストとCSVファイルの１行データを直接読み書きできる。文字列データで、途中に空白が入ってもOKなのが特徴
- 読込み

```
import csv
matrix = [ ]
f = open( filename, "r" )
reader = csv.reader( f )
for row in reader:  matrix.append( row )
f.close( )
```

- 書込み

```
f = open( filename, "w" )
writer = csv.writer( f )
for row in matrix: writer.writerow( row )    # writer.writerows( matrix )
f.close( )
```

# インターネットからの読込み

- urllib.requestモジュールを利用する
- httpsでのアドレス指定の場合、sslのcertificateを回避する設定を入れる
- readで取り出した内容は、byte文字列になっているため、decode()関数でUnicodeとして解釈させる
  - 記述例

```
import urllib.request
import ssl
ssl._create_default_https_context = ssl._create_unverified_context

responce = urllib.request.urlopen( "https://www.sfc.keio.ac.jp")
content = responce.read( )
responce.close( )

lines = content.decode( ).splitlines()
for i, line in enumerate( lines, start=1 ):
    print (str( i ).zfill( 4 ), ":", line )
```

# Beautiful Soupを用いる

- インストール
  - pip3 install bs4  (Windowsの場合はpipで大丈夫)
  - M1/M2 cpuを持つmacの場合：sudo arch -arm64 pip3 install bs4
- 利用例：前スライドのように、contentにWebページから読み込まれているとする

```
from bs4 import BeautifulSoup
webtext = content.decode()
soup = BeautifulSoup( webtext, "html.parser" )
print( soup.title )
items = soup.find_all( "h2" )
for item in items:
    print( item )
```

# Beautiful SoupでXMLデータの読込み

- xmlの解析には、lxmlパッケージが必要
  - pip3 install lxmlでインストール
    macOSの場合、sudo arch -arm64 pip3 install lxml
- findも使えるが、xmlのタグが 1 つしかない場合、そのタグ名でアクセス可能
- 例：  "https://www.w3schools.com/xml/plant_catalog.xml"にあるXMLデータ

```
xmltext = content.decode( )
soup = BeautifulSoup( xmltext, "xml" )
items = soup.find_all( "PLANT" )
commonlist = [ item.COMMON.text for item in items] # item.find( "COMMON" ).textでもOK
bottalist = [ item.BOTANICAL.text for item in items ] # item.find( "BOTANICAL" ).textでもOK
plantmatrix = list( map( list, zip( commonlist, bottalist ) ) )
```