農学生命情報科学特論 I

ICT や IoT 等の先端技術を活用し、効率よく高品質生産を可能にするスマート農業への取り組みは世界的に進められています。その基礎を支えている技術の一つがプログラミング言語。なかでも、習得しやすくかつ応用範囲の広い Python がとくに注目されています。本科目では、農学生命科学の分野で利用される Python の最新事例を紹介しながら、Python の基礎文法の講義を行います。

孫　建強　https://aabbdd.jp/
農研機構・農業情報研究センター

OCT
03 13:15–16:30

Python 基礎

第 1 回目の授業では、プログラミング言語の基本であるデータ構造とアルゴリズムを簡単に紹介してから、Python の基本構文を紹介する。Python のスカラー、リスト、ディクショナリ、条件構文と繰り返し構文を取り上げる。

OCT
10 13:15–16:30

テキストデータ処理

バイオインフォマティックスの分野において、塩基配列やアミノ酸配列などの文字列からなるデータを扱うことが多い。第 2 回目の授業では、Python を利用した文字列処理を紹介し、FASTA や GFF などのファイルから情報を抽出する方法を取り上げる。

OCT
17 13:15–16:30

データ分析

第 3 回目の授業では、Python ライブラリー（NumPy や Pandas）を利用して、CSV ファイルの処理などのデータ分析やデータ可視化を中心に取り上げる。

OCT
24 13:15–16:30

スマート農業

Python のライブラリー（PyTorch 等）を利用して、深層学習による物体分類や物体検出モデルを実装する例を示す。

成績評価

- 出席点はありません。講義資料を見て、各自の判断で出席・欠席・途中退室なさってください。
- 成績は、講義最終日に出題する演習課題で評価する。
- 演習課題を解く際に、ChatGPT や Copilot 等のコーディング補助ツールを利用してもよい。

質問

- Anaconda, Jupyter Notebook, Python 全般, レポート課題などに関する質問は、下記のメールアドレスに直接問い合わせていただいて構いません。また、アグリバイオ事務局を通して質問していただいても構いません。

  ✉ report＠aabbdd.jp

- 講義中、個人のコンピュータにインストールした Jupyter Notebook などが正しく動作しなくなった場合、すぐに Google Colab に切り替えることを推奨します。

農学生命情報科学特論Ⅰ

1

- プログラミング言語
- 基本オブジェクト
- 基本文法

農学生命情報科学特論Ⅰ

1

- プログラミング言語
- 基本オブジェクト
- 基本文法

プログラミング言語

- プログラミング言語
- データ構造とアルゴリズム

プログラミング言語

特定のタスクを達成させるための一連の操作を、
コンピューターに理解可能な形で記述するための言語。

プログラミング言語とゲーム

プログラミング言語をゲームに例えるとわかりやすい。例えば、ゲームではコントローラーを通じて入力し、上に進む、左に進む、右に進む、下に進むなど基本操作（命令）を入力し、キャラクターを制御し、ミッションをクリアする。これらの一連の入力（上下左右の順序や回数）がプログラミング言語である。

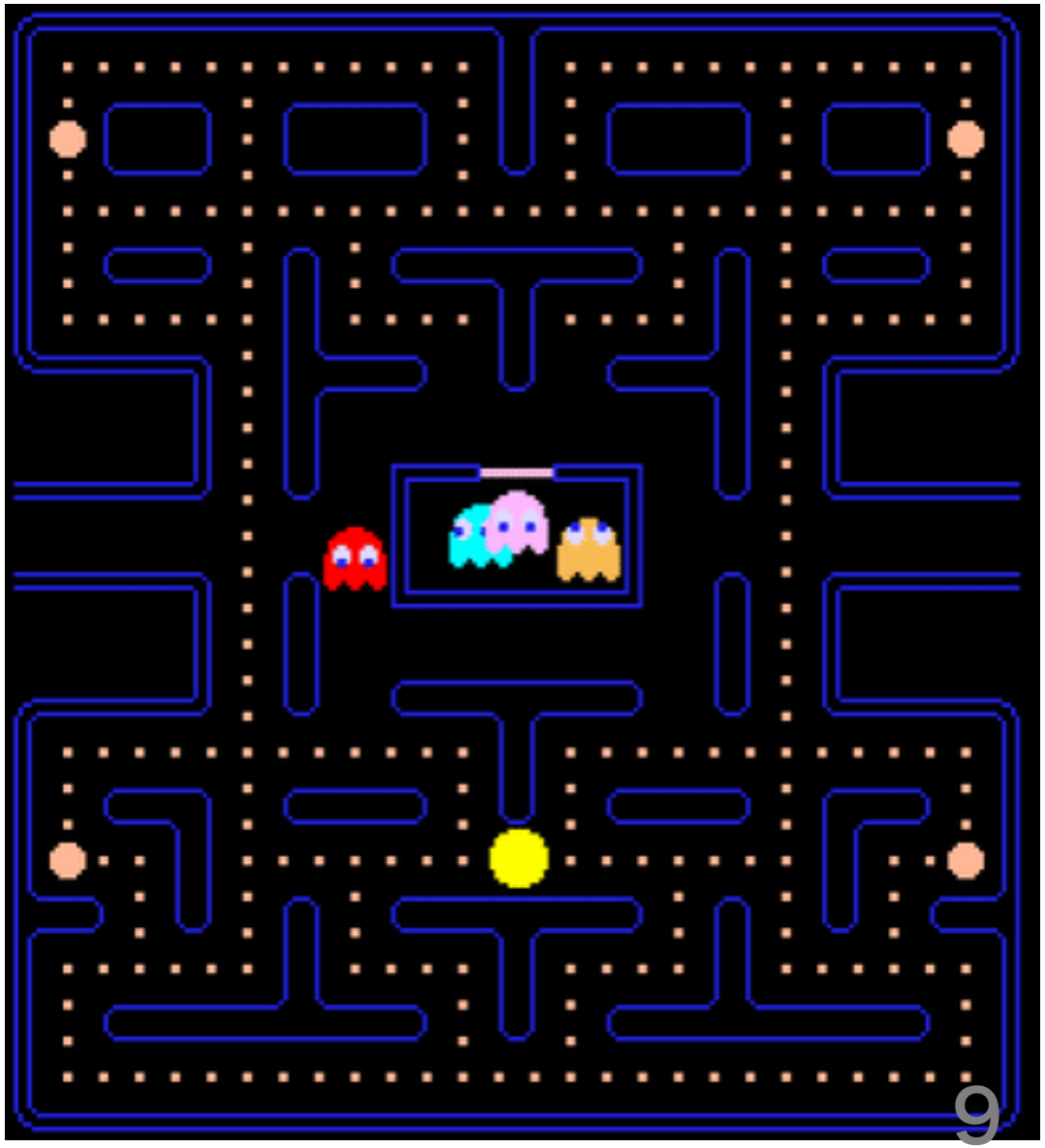
言語仕様

U	上へ進む
D	下へ進む
L	左へ進む
R	右へ進む
0-9	進むマス数

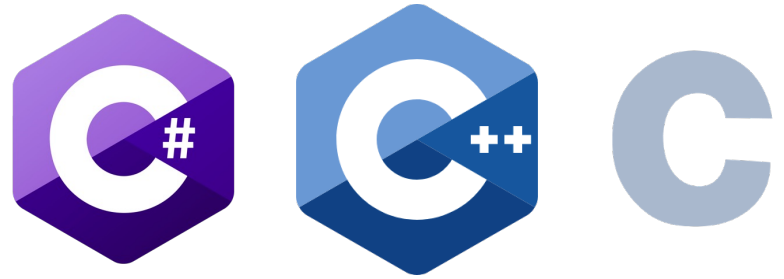
→

L4
D3
R3
D3
R3
U3
R3
U3
R3

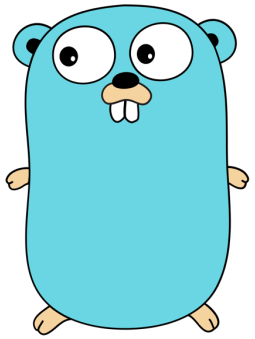
→

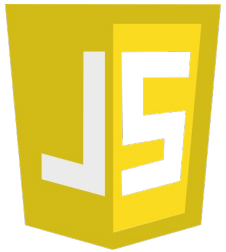


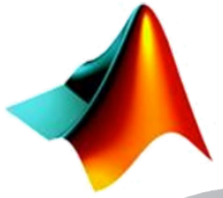
プログラミング言語



julia







MATLAB

現在、数百種類以上のプログラミング言語が使われている。その機能に着目すると、様々な目的に使用できる汎用型言語と統計やウェブページ作成などの特定の目的で使用できる専用型言語に分けることができる。プログラミング言語同士に優劣はなく、目的に応じて使い分けされている。

YouTube

Instagram

django

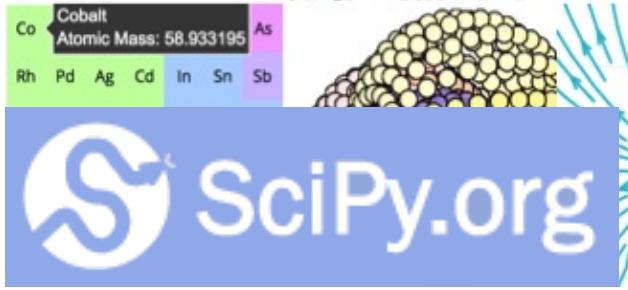
Dropbox

NumPy

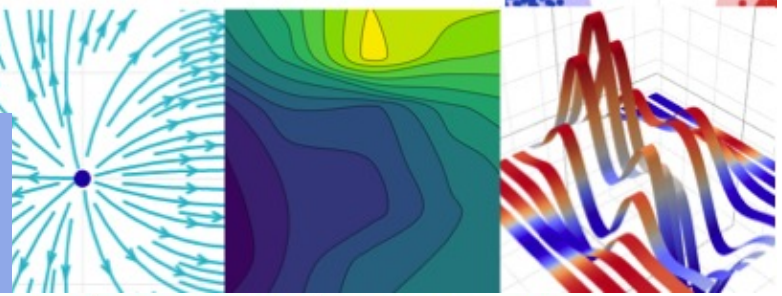
pandas

matplotlib

plotly

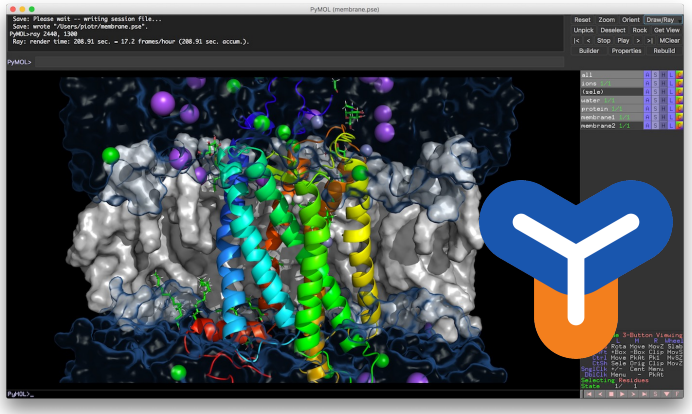


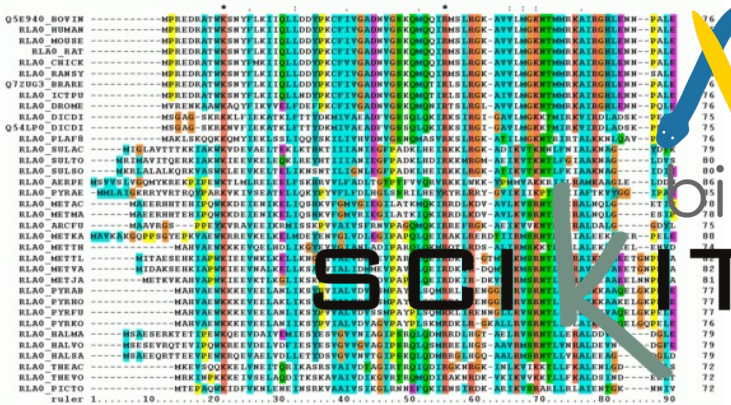
SciPy.org



Apache OpenOffice™

scikit learn





biopython

scikit-bio

PyTorch

TensorFlow

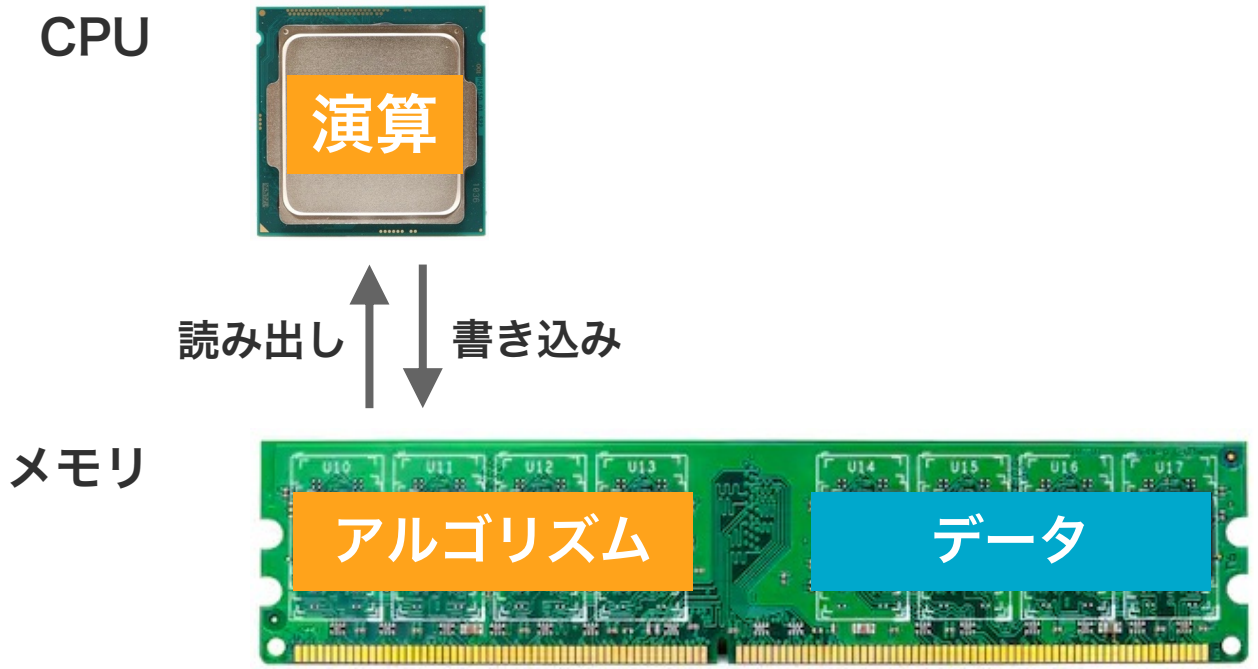
python™

プログラミング言語

- プログラミング言語
- データ構造とアルゴリズム

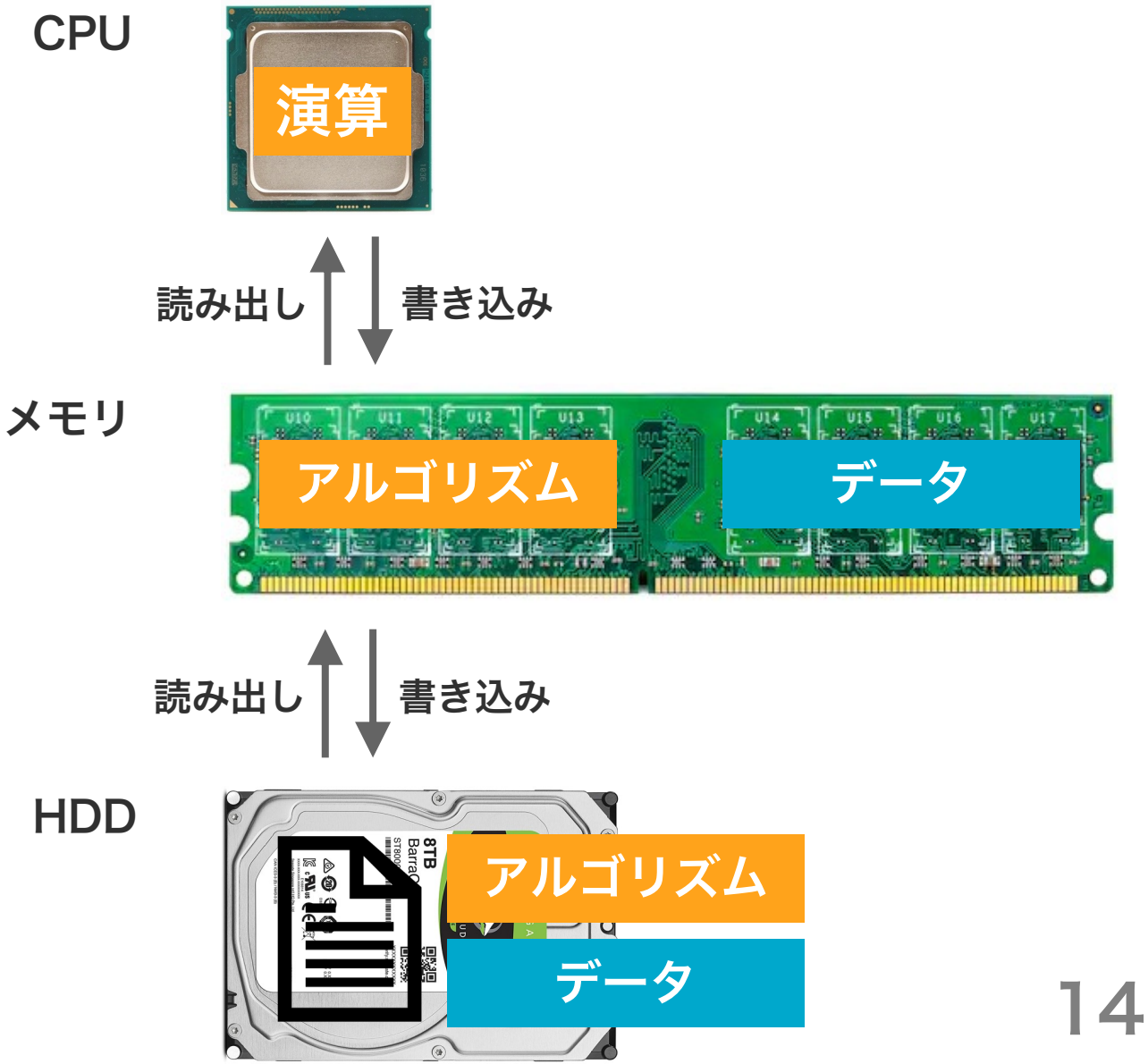
データ構造とアルゴリズム

コンピュータでデータを処理するためには、データや処理手順（アルゴリズム）をコンピュータのメモリ上に保持する必要がある。CPU はメモリ上に保持されたアルゴリズムに従い、指定された位置にアクセスしてデータを読み込み、計算を行い、その計算結果をまたメモリ上の指定された位置に保存する。



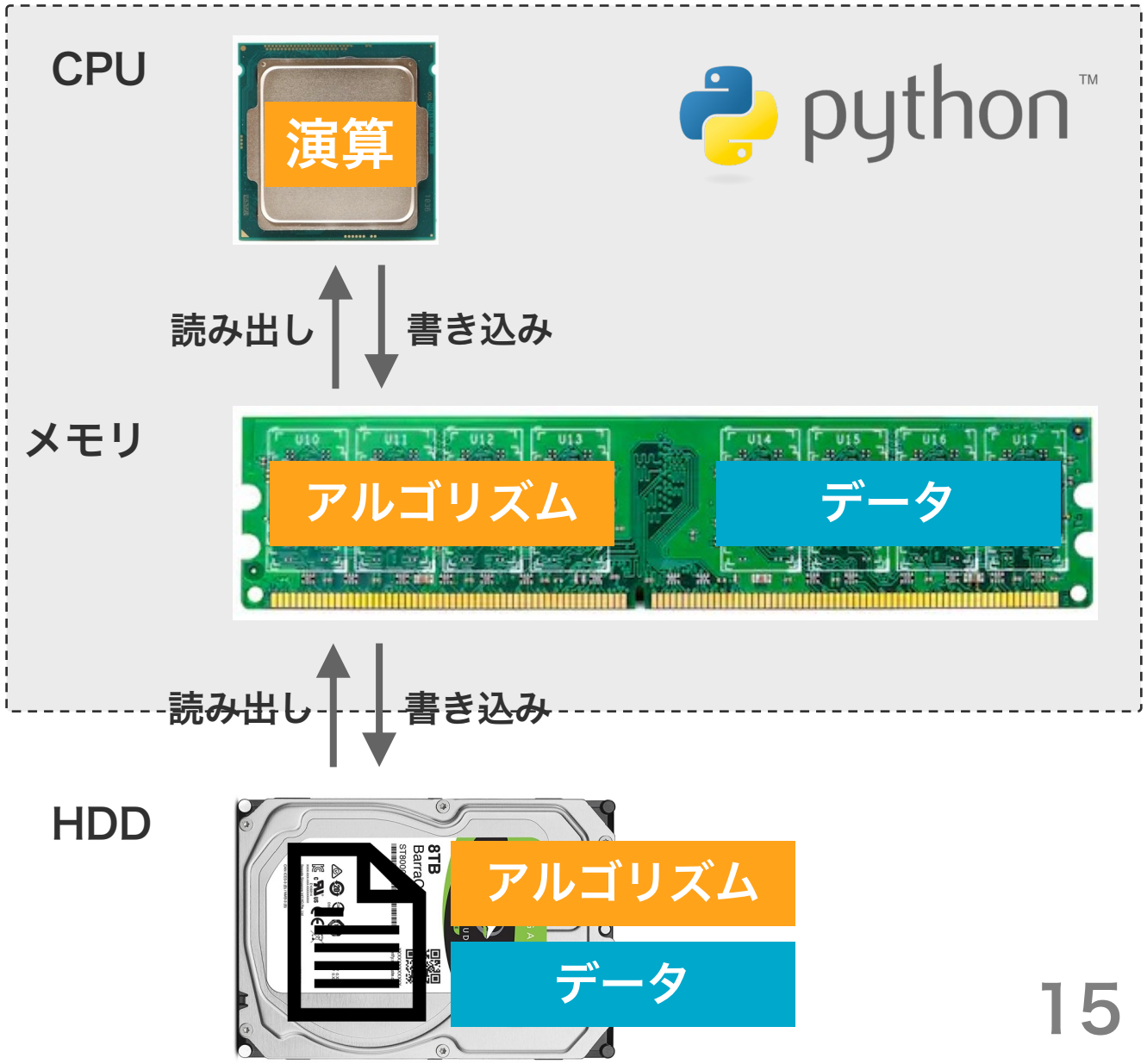
データ構造とアルゴリズム

メモリ上に保持された内容は、電源を切ると、すべて消える。そのため、解析する度に、アルゴリズムやデータを直接メモリに書き込むよりも、アルゴリズムやデータを不揮発性メモリであるハードディスクに保存し、必要に応じてハードディスクからメモリに自動的にコピーしてから解析した方が効率がいい。



データ構造とアルゴリズム

アルゴリズムおよびデータを Python 文法に従ってファイルに記述すれば、Python エンジンが、そのファイルの情報をメモリ上に展開し、CPU とメモリ間の制御を行いながら演算を行う。

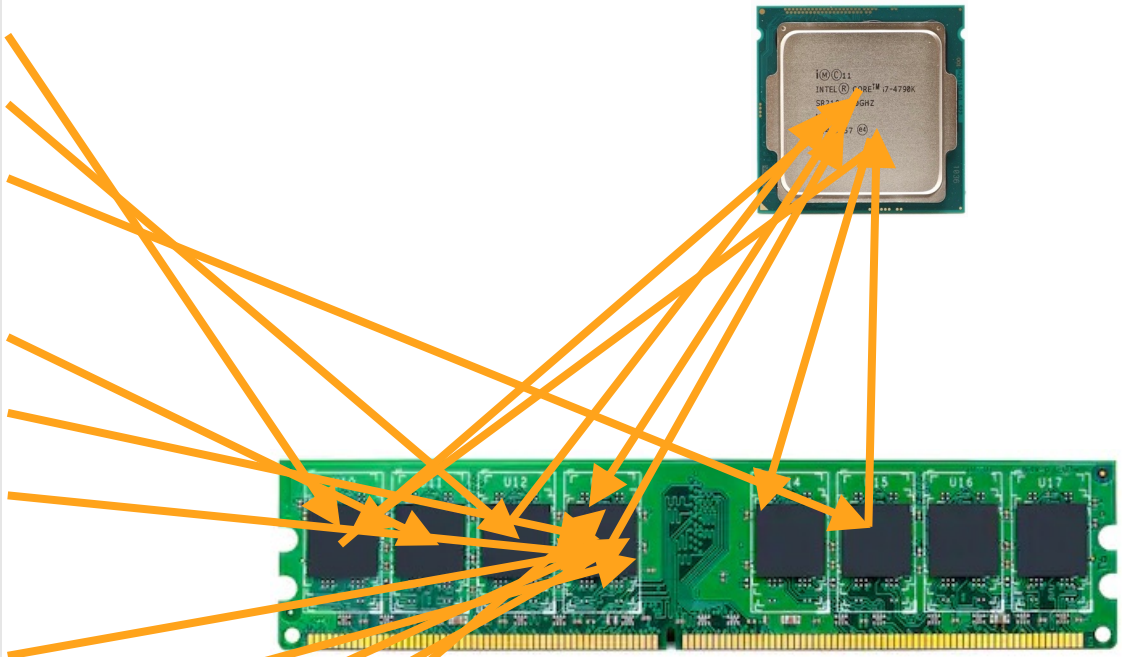


データ構造とアルゴリズム

データ

アルゴリズム

```
a = 120
takeout = True
has_coupon = True

s = 0
if has_coupon:
    s = a * 0.95

if takeout:
    s = a * 1.08
else:
    s = a * 1.10

print(s)
```



python™

データとアルゴリズムを 1 つのファイルに記述し、それを Python に実行させることで、Python がメモリや CPU 間の様々な制御を行う。

データ構造とアルゴリズム

データ →

- 整数
- 小数
- 文字
- ベクトル
- 行列

→

- スカラー
- リスト
- ディクショナリ
- セット

アルゴリズム →

- 並べ替え
- 探索
- 線型計画法
- 待ち行列理論
- 動的計画法

→

- 条件構文
- 繰り返し構文
- 関数

探索アルゴリズム

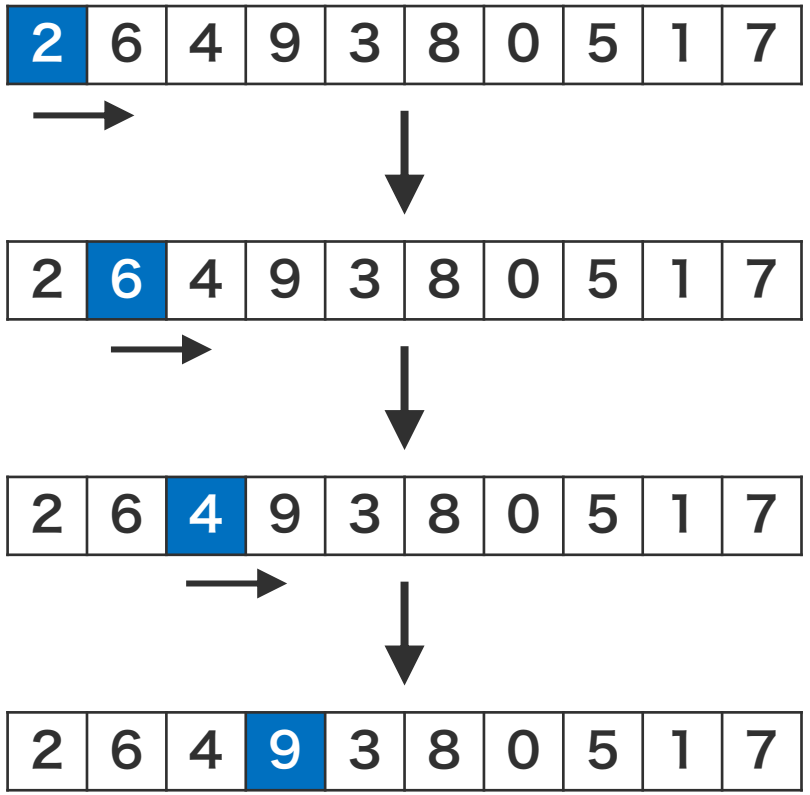
次の数列の中に 9 が存在するかどうかを調べよ。ただし、コンピュータは、複数の作業を同時に実行できないことに注意せよ。

2	6	4	9	3	8	0	5	1	7

探索アルゴリズム

次の数列の中に 9 が存在するかどうかを調べよ。ただし、コンピュータは、複数の作業を同時に実行できないことに注意せよ。

2	6	4	9	3	8	0	5	1	7



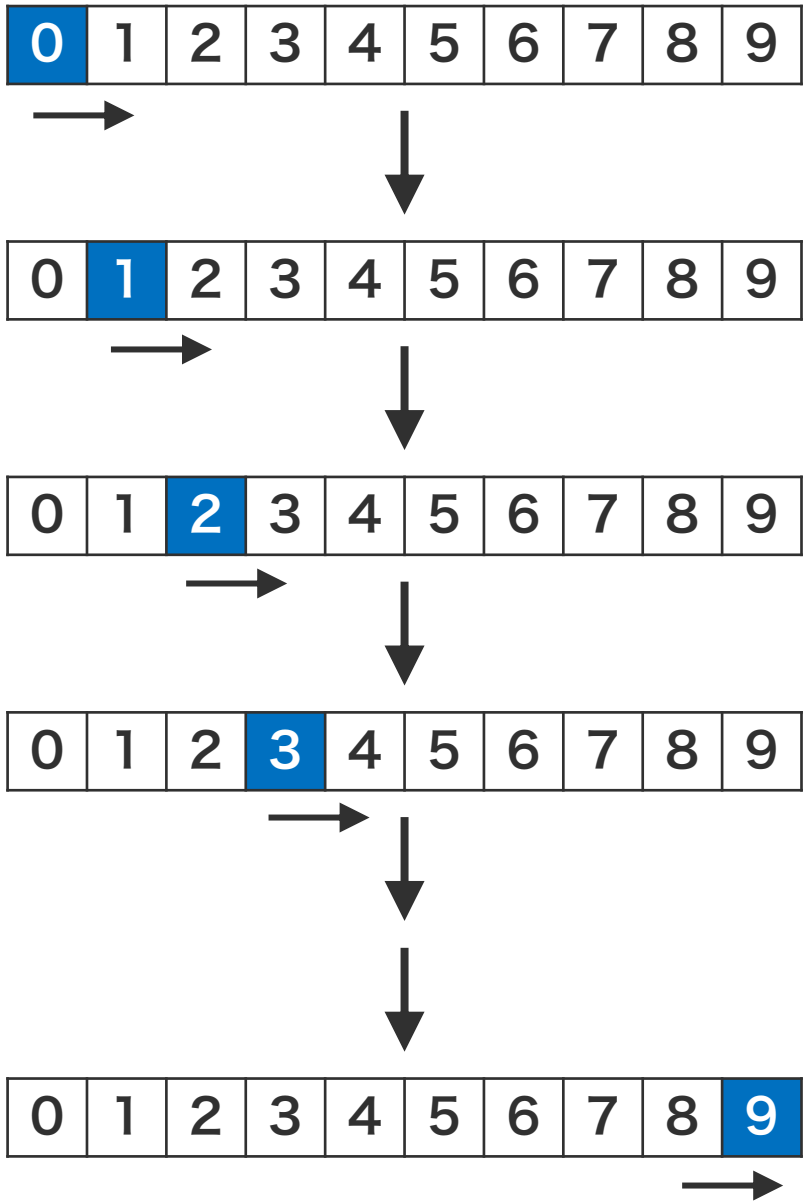
探索アルゴリズム

次の数列の中に 9 が存在するかどうかを調べよ。ただし、コンピュータは、複数の作業を同時に実行できないことに注意せよ。

0	1	2	3	4	5	6	7	8	9

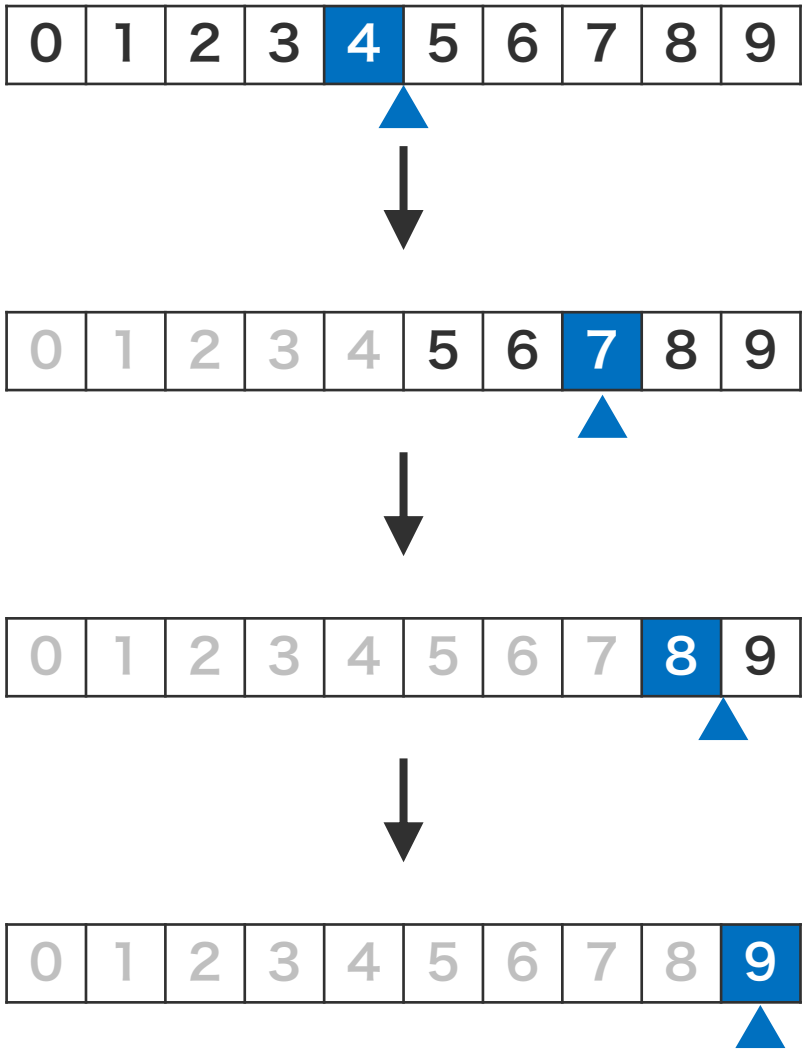
探索アルゴリズム

次の数列の中に 9 が存在するかどうかを調べよ。ただし、コンピュータは、複数の作業を同時に実行できないことに注意せよ。



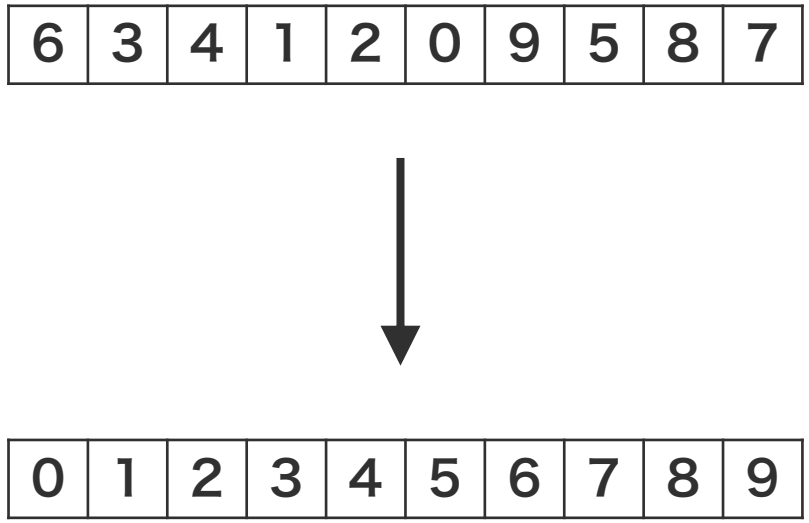
探索アルゴリズム

次の数列の中に 9 が存在するかどうかを調べよ。ただし、コンピュータは、複数の作業を同時に実行できないことに注意せよ。



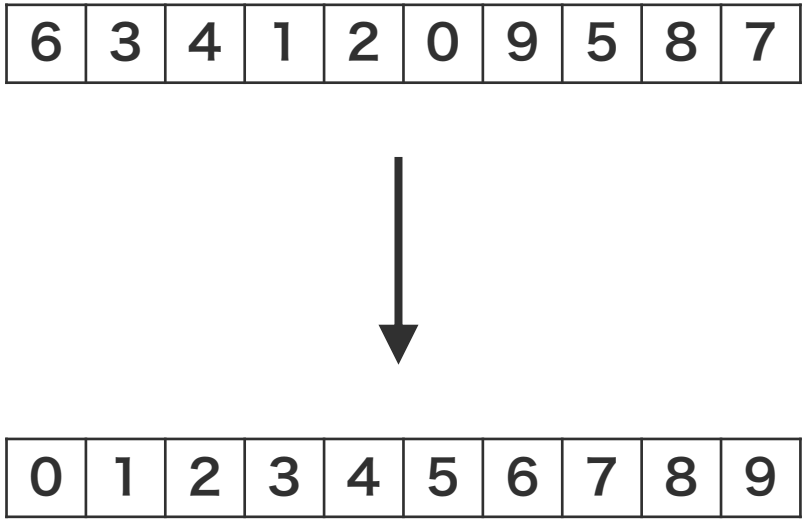
並べ替えアルゴリズム

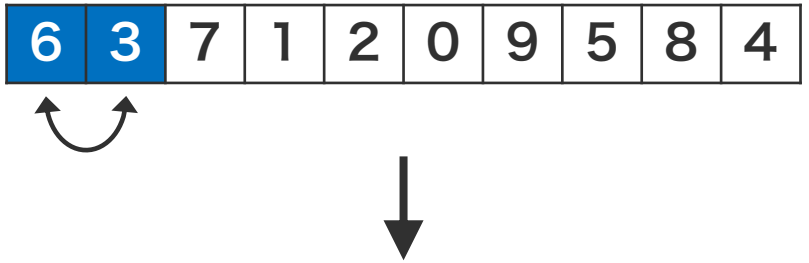
次の数列の各要素を昇順に並べ替えよ。ただし、コンピュータは、複数の作業を同時に実行できないことに注意せよ。



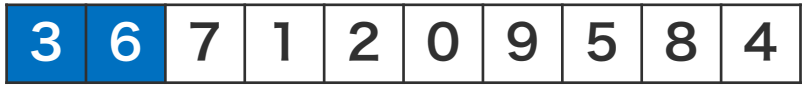
並べ替えアルゴリズム

次の数列の各要素を昇順に並べ替えよ。ただし、コンピュータは、複数の作業を同時に実行できないことに注意せよ。



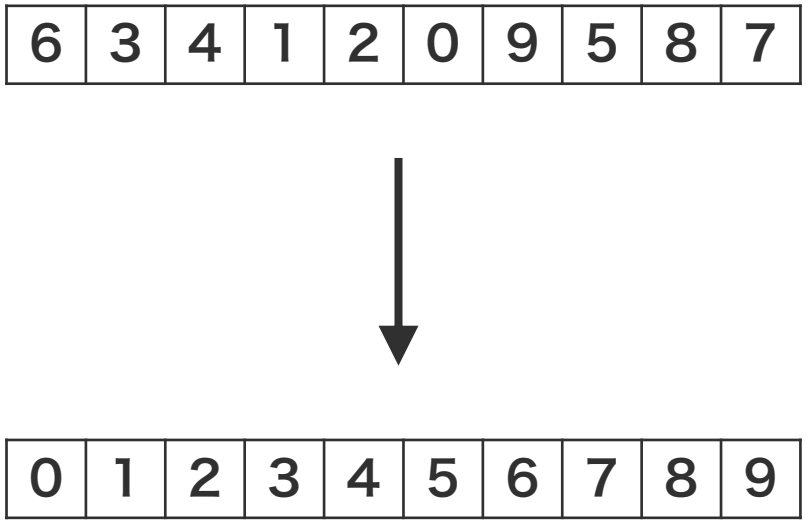


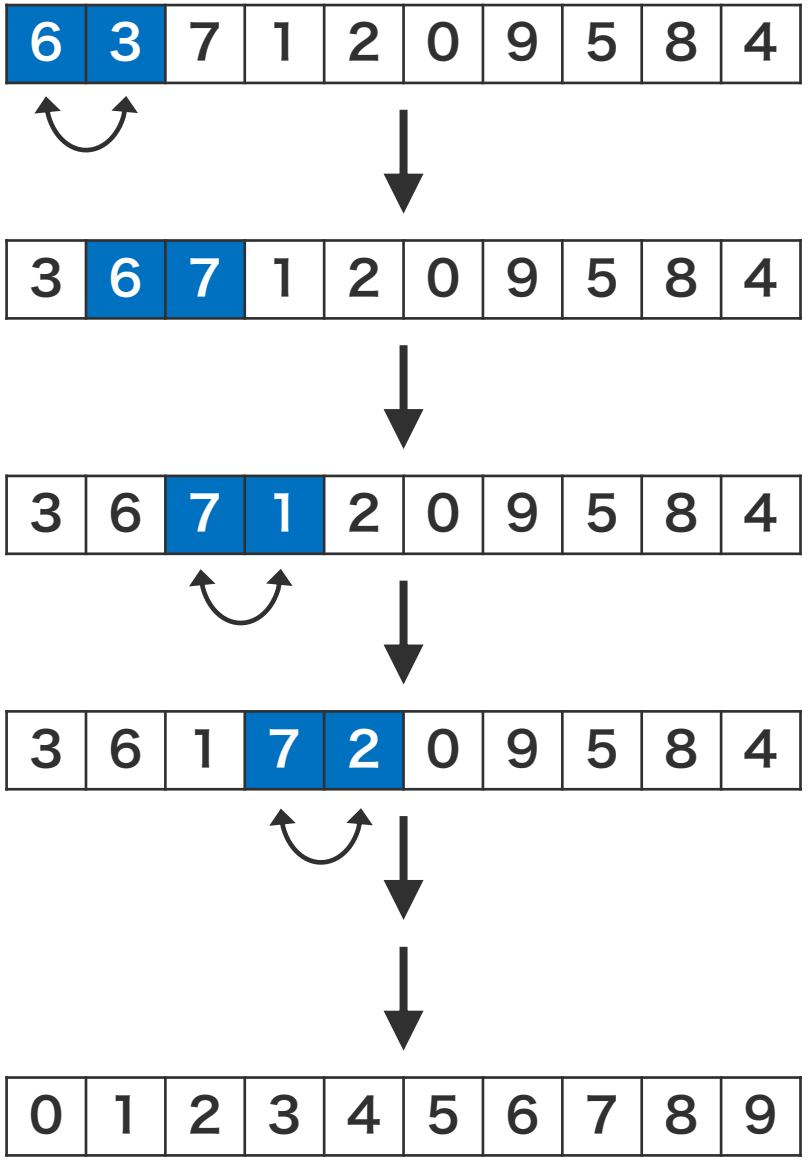
1) 1 番目の要素と 2 番目の要素を比較する。
2) 2 番目の要素が大きければ、何もしない。
3) 1 番目の要素が大きければ、
   i. 2 番目の要素を変数 tmp に移動する。
   ii. 2 番目の位置に空きができるので、1 番目の要素をそこに移動する。
   iii. 1 番目の位置に変数 tmp の値を移動する。



並べ替えアルゴリズム

次の数列の各要素を昇順に並べ替えよ。ただし、コンピュータは、複数の作業を同時に実行できないことに注意せよ。





農学生命情報科学特論Ⅰ

1

- プログラミング言語
- **基本オブジェクト**
- 基本文法

基本オブジェクト

- スカラー
- リスト
- ディクショナリ

変数

変数は、値を保存するための箱のようなものである。変数に値を保持させるためには、その変数に値を付与する必要がある。この付与作業を代入という。Python では、変数に値を代入するとき、代入演算子 "=" を使う。代入演算子 "=" は、右辺の値を、左辺の変数に代入する機能を持つ。代入演算子 "=" は、数学における等しいという意味を持たない。なお、変数のことをオブジェクトともいう。

右のコードは、1 という値を、a という名前の変数に代入することを表している。このコードを実行することによって、プログラムが終了するまで、a は 1 を保持している状態になる。

```
a = 1
```

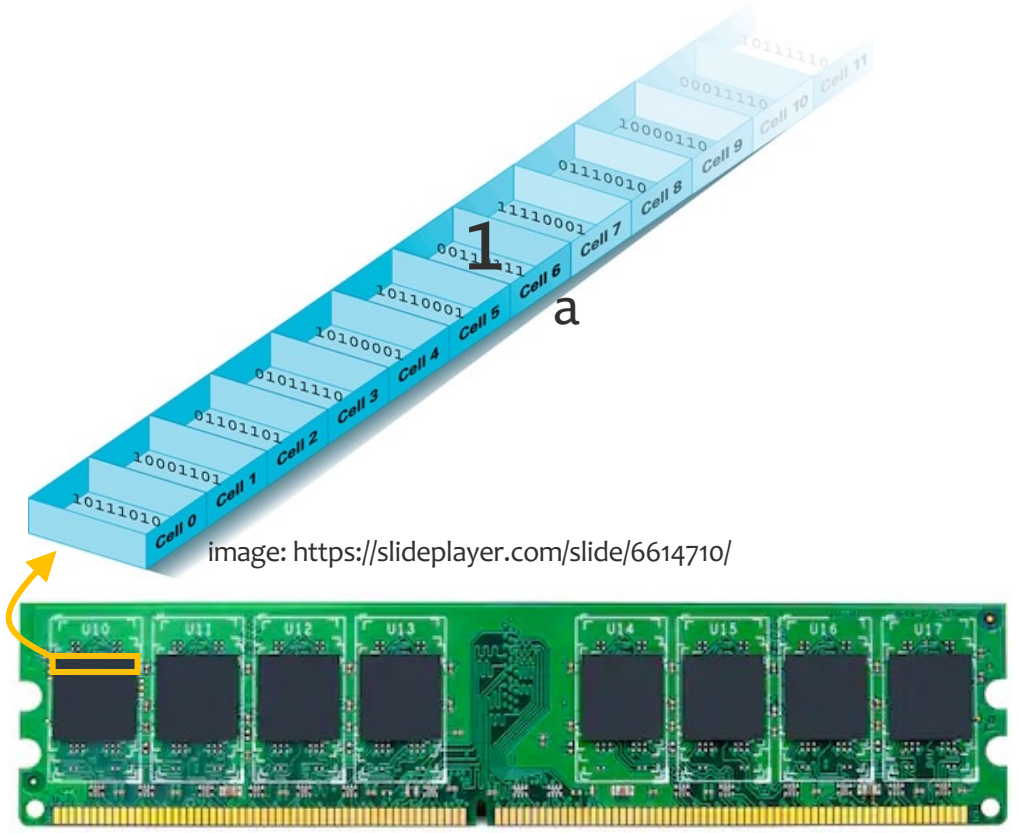
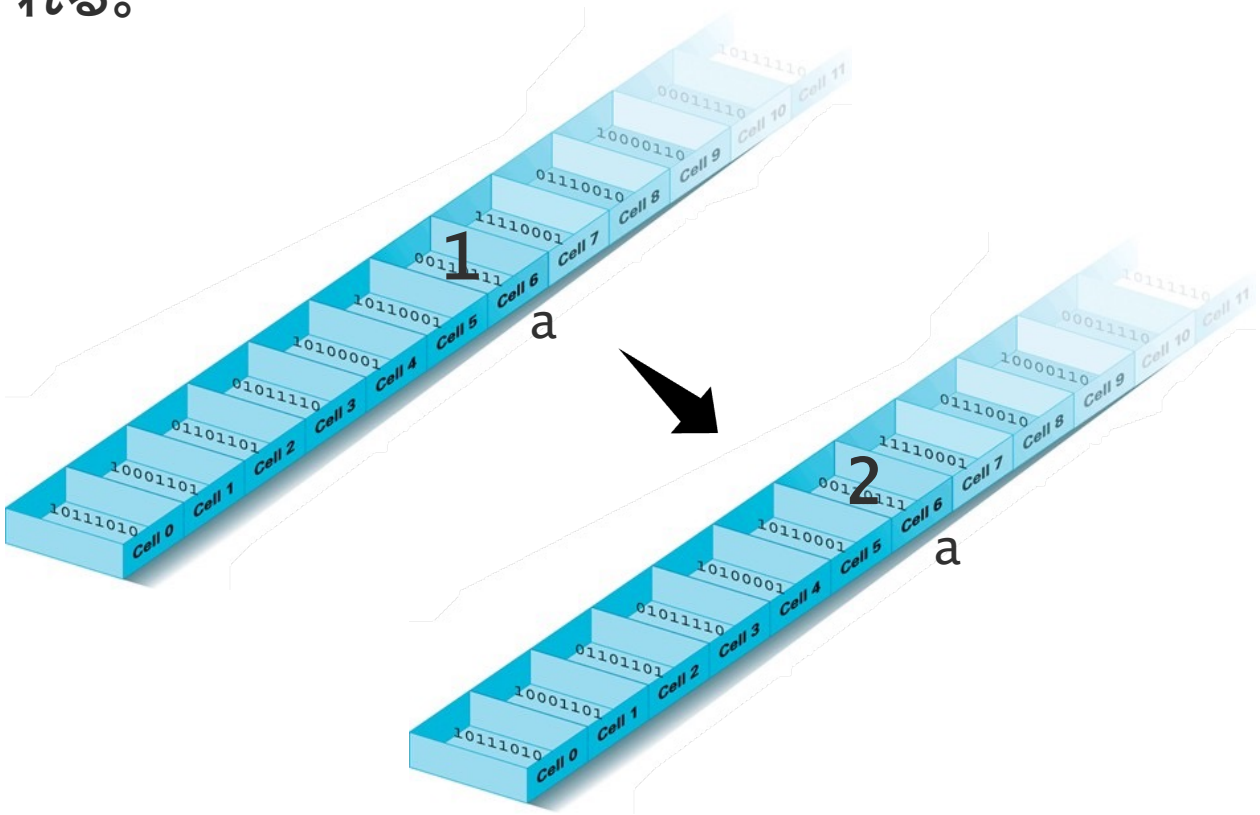


image: https://slideplayer.com/slide/6614710/

図はイメージ。実際は、整数 1 つを保存するのに複数個の箱が必要。

変数

代入演算子 "=" の右辺が計算式の場合は、その計算結果が左辺に代入される。また、すでに値を保持している変数に、他の値を新たに代入すると、既存の値が上書きされる。



※ a という箱が 1 つしかないので、そこに 1 つの値しか保存できない。

```
a = 1
b = 1

c = a + b
# 2

d = c - 2
# 0

e = a + 2
# 3

f = d + e
# 3

a = f - 1
# 2
```

標準出力

変数に代入された値は、コンピューターのメモリ上に保存される。出力命令を出さない限り、ディスプレイ（標準出力）に出力されない。出力命令をだすとき、print関数を使用する。

※この資料では、これ以降 print 関数を省略する。ただし、Jupyter Notebookなどのアプリケーションでは print 関数を使わなくても、変数の内容をアプリケーション上に表示させる機能がある。

```
a = 1
b = 1

c = a + b
print(c)
# 2

d = c - 2
print(d)
# 0

e = a + 2
print(e)
# 3

f = d + e
print(f)
# 3
```

変数の使い回し

代入演算子 "=" の右辺が計算式の場合は、その計算結果が左辺に代入される。つまり、順序として、まず代入演算子の右辺の計算式を計算する。次に、その計算結果を代入演算子の左辺の変数に代入する。このように代入演算子の動作に順序があるため、右のサンプルコードのように、代入演算子の左右に同じ変数を書いても正しく動作する。

```
a = 1


a = a + 1
a
# 2


a = a - 2
a
# 0
```

算術演算子

Python で四則演算を行うのに次のような演算子を使用する。割り算に関して、余と商を求める演算子もある。

演算子	意味	例
+	加	7 + 4 → 11
-	減	7 - 4 → 3
*	乗	7 * 4 → 28
/	除	7 / 4 → 1.75
%	余	7 % 4 → 3
//	商	7 // 4 → 1
**	累乗	2 ** 10 → 1024

※小数に対しても余と商を求めることができる。a//b は、a を b で割った後に整数部分を返す。また、a%b は、a - a//b*b で計算された結果が返される。

```
a = 11
b = 3

a + b
# 14

a - b
# 8

a * b
# 33

a / b
# 3.6666666666665

a % b
# 2

a // b
# 3
```

変数名の命名規則

変数の名前に使える文字は、数字・アルファベット・アンダーバーである。ただし、変数名の最初の文字を数値以外の文字にする必要がある。また、予約語（if, for, def など）を変数名にすることができない。この規則を守れば、変数名を自由につけることができるが、実際には、この規則のほかに慣用的なルール（PEP8）に従って変数を命名することが一般的である。

- 変数名、関数名をその作用がわかるように小文字の英単語で命名する。単語が複数ある場合、アンダーバーで繋げる。
- 定数をその意味がわかるように大文字の英単語で命名する。単語が複数ある場合、アンダーバーで繋げる。

```
def calc_odd_sum(x):
    x = np.array(x)
    s = np.sum(x[x % 2 == 1])
    return s


fib_n = [1, 1, 2, 3, 5, 8, 13, 21, 34]

fib_odd_sum = calc_odd_sum(fib_n)



SIMULATION_TRY = 100

sim_results = []

for (i in range(SIMULATION_TRY)) {
    _result = simulate_calc_pi()
    sim_results.append(_result)
}
```

問題 O1-1

コシヒカリは複数の県で植えられている。各県における
コシヒカリの 10a あたりの収量を次の表にまとめた。
これらの平均収量を計算せよ。

※ 1a (アール) は 10m x 10m。10a は 10x100m²。

都道府県名	収量 (kg/10a)
新潟	528.3
茨城	520.7
福島	538.8
栃木	535.2

```
niigata = 528.3

ibaraki = 520.7

fukushima = 538.8

tochigi = 535.2
```

本来はこれらのデータを CSV ファイルから直接読み込むが、初めは練習のために直打ちで！

変数

変数に数値のほか文字や文字列を代入することもできる。

文字を変数に代入するとき、その文字が、変数の名前ではなく、文字のデータであることを明示するために、文字データの両側を引用符で囲む必要がある。

```
s = 'a'



t = 'apple'



u = 't'
u
# 't'



v = t
v
# 'apple'
```

基本オブジェクト

- スカラー
- リスト
- ディクショナリ

リスト

同じ属性をもつ複数の値をまとめて扱うときにリストを使う。たとえば、ある種のどんぐりの重さの分布を調べたいとき、どんぐりを 3 つ拾い、それぞれの重さを測ったのち、それらの平均や分散を求めていくことになる。この際に、3 つのどんぐりの重さをそれぞれ別々の変数に保存するよりも、それらをまとめて 1 つの変数に保存した方がわかりやすい。このときリストが使われる。


```
a = [2, 3, 2]
```

1. 複数の値をカンマ区切りで並べる

2. 角括弧で全体をまとめる

リスト

複数のスカラーの平均を求める場合、すべての変数の合計値を計算し、その合計値を変数の個数で割る計算を行う。

```
weight_1 = 2
weight_2 = 3
weight_3 = 2

s = weight_1 + weihgt_2 + weight_3
n = 3

mean = s / n
```

リストの全要素に対して平均値を求めたいとき、全要素を束ねて一括で計算することができる。

```
weights = [2, 3, 2]

n = len(weights)
s = sum(weights)


mean = s / n
```

あるオブジェクトを代入すると、他のオブジェクトが返ってくるオブジェクトを関数という。

関数	機能
len(x)	リスト x の要素数を調べる
sum(x)	リスト x の全要素の合計を求める

リスト

複数のスカラーの平均を求める場合、すべての変数の合計値を計算し、その合計値を変数の個数で割る計算を行う。

```
weight_1 = 2
weight_2 = 3
weight_3 = 2

s = weight_1 + weihgt_2 + weight_3
n = 3

mean = s / n
```

リストの全要素に対して平均値を求めたいとき、全要素を束ねて一括で計算することができる。

```
weights = [2, 3, 2]

n = len(weights)
s = sum(weights)


mean = s / n
```

あるオブジェクトを代入すると、他のオブジェクトが返ってくるオブジェクトを関数という。

関数	機能
len(x)	リスト x の要素数を調べる
sum(x)	リスト x の全要素の合計を求める

問題 O2-1

コシヒカリは複数の県で植えられている。各県におけるコシヒカリの 10a あたりの収量を次の表にまとめた。これらの平均収量を計算せよ。

都道府県名	収量 (kg/10a)
新潟	528.3
茨城	520.7
福島	538.8
栃木	535.2
千葉	512.0

```
koshihikari = [528.3, 520.7, 538.8,
               535.2, 512.0]
```

リスト

リストは、同じ属性の要素が複数あったとき、それらの要素を束ねて扱うときに利用される。これらの要素には、添字（index）とよばれる番号が振り当てられている。添字を使用することで、特定の位置にある要素を取り出して、表示したり、再代入したりすることができる。リストの添字は、リストの先頭から 0、1、2、などのように整数が振り当てられている。添字を使って特定の要素を取り出すとき、w[0] のように、変数名のすぐ後に角括弧で添字を囲むように使用する。この使用方法は Python を使用する上での定義であり、括弧の形を変えることはできない。

```
w = [12, 10, 11, 13, 11]


w[0]


w[1]


w[5]
```

リスト

リストは、同じ属性の要素が複数あったとき、それらの要素を束ねて扱うときに利用される。これらの要素には、添字（index）とよばれる番号が振り当てられている。添字を使用することで、特定の位置にある要素を取り出して、表示したり、再代入したりすることができる。リストの添字は、リストの先頭から 0、1、2、などのように整数が振り当てられている。添字を使って特定の要素を取り出すとき、w[0] のように、変数名のすぐ後に角括弧で添字を囲むように使用する。この使用方法は Python を使用する上での定義であり、括弧の形を変えることはできない。

```
w = [12, 10, 11, 13, 11]


w[0]
# 12

w[1]
# 10

w[5]
# IndexError: list index out of range
```

リストのサイズを超えた添字を与えると、添字が範囲外にあるという IndexError が発生する。

リスト

リストは、同じ属性の要素が複数あったとき、それらの要素を束ねて扱うときに利用される。これらの要素には、添字（index）とよばれる番号が振り当てられている。添字を使用することで、特定の位置にある要素を取り出して、表示したり、再代入したりすることができる。リストの添字は、リストの先頭から 0、1、2、などのように整数が振り当てられている。添字を使って特定の要素を取り出すとき、w[0] のように、変数名のすぐ後に角括弧で添字を囲むように使用する。この使用方法は Python を使用する上での定義であり、括弧の形を変えることはできない。

```
w = [12, 10, 11, 13, 11]


w[0]
# 12

w[1]
# 10

w[5]
# IndexError: list index out of range

w[2] = 9


w
```

リスト

リストは、同じ属性の要素が複数あったとき、それらの要素を束ねて扱うときに利用される。これらの要素には、添字（index）とよばれる番号が振り当てられている。添字を使用することで、特定の位置にある要素を取り出して、表示したり、再代入したりすることができる。リストの添字は、リストの先頭から 0、1、2、などのように整数が振り当てられている。添字を使って特定の要素を取り出すとき、w[0] のように、変数名のすぐ後に角括弧で添字を囲むように使用する。この使用方法は Python を使用する上での定義であり、括弧の形を変えることはできない。

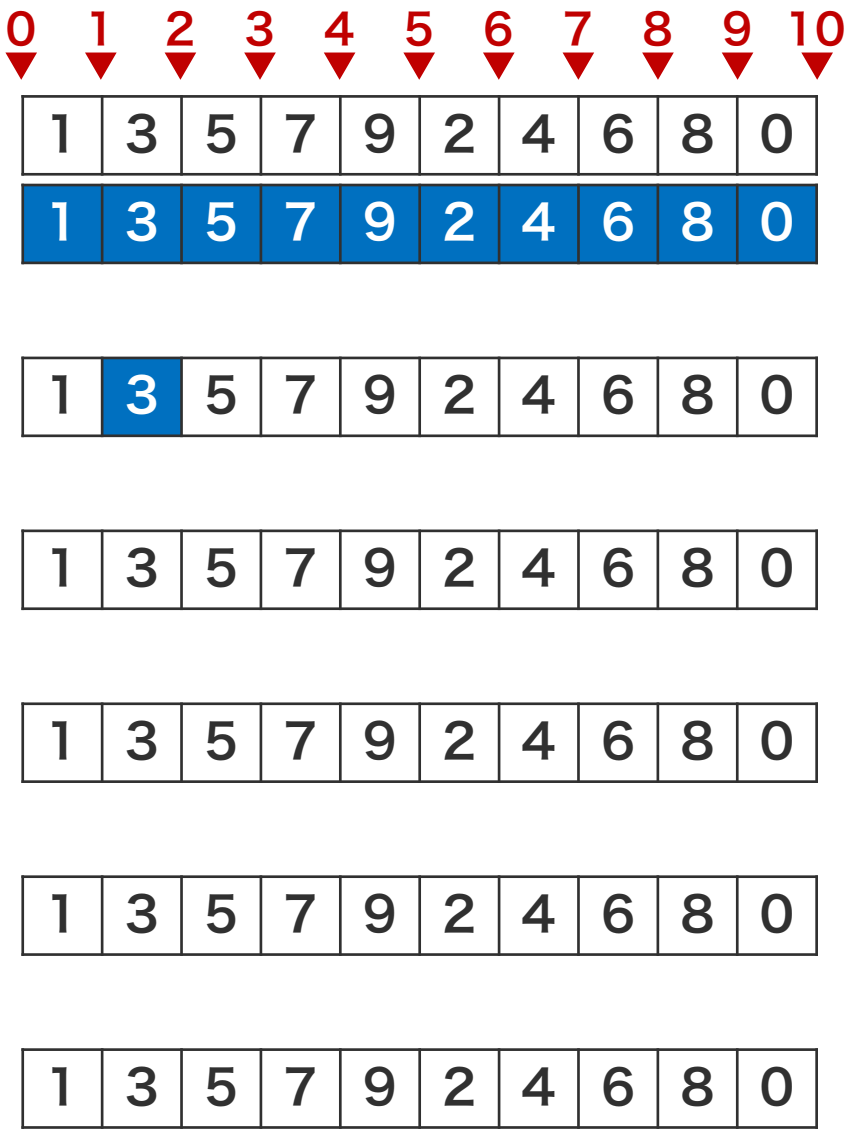
```
w = [12, 10, 11, 13, 11]



w[0]
# 12

w[1]
# 10

w[5]
# IndexError: list index out of range

w[2] = 9


w
# [12, 10, 9, 13, 11]
```

リスト

リストの中から、連続した要素をまとめて取り出す（スライス）とき、":" を使うと便利である。

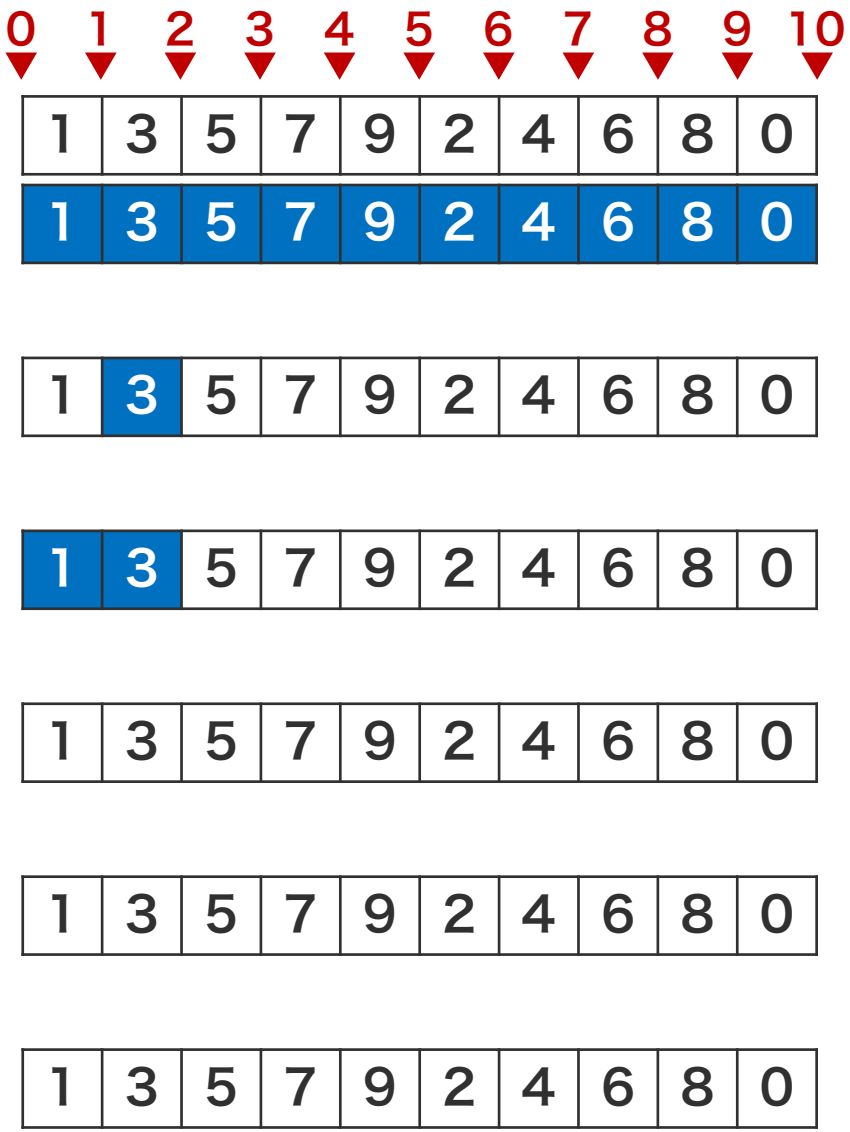
```
a = [1, 3, 5, 7, 9, 2, 4, 6, 8, 0]



a
# [1, 3, 5, 7, 9, 2, 4, 6, 8, 0]

a[1]
# 3

a[0:2]


a[2:6]


a[:4]


a[5:]
```

リスト



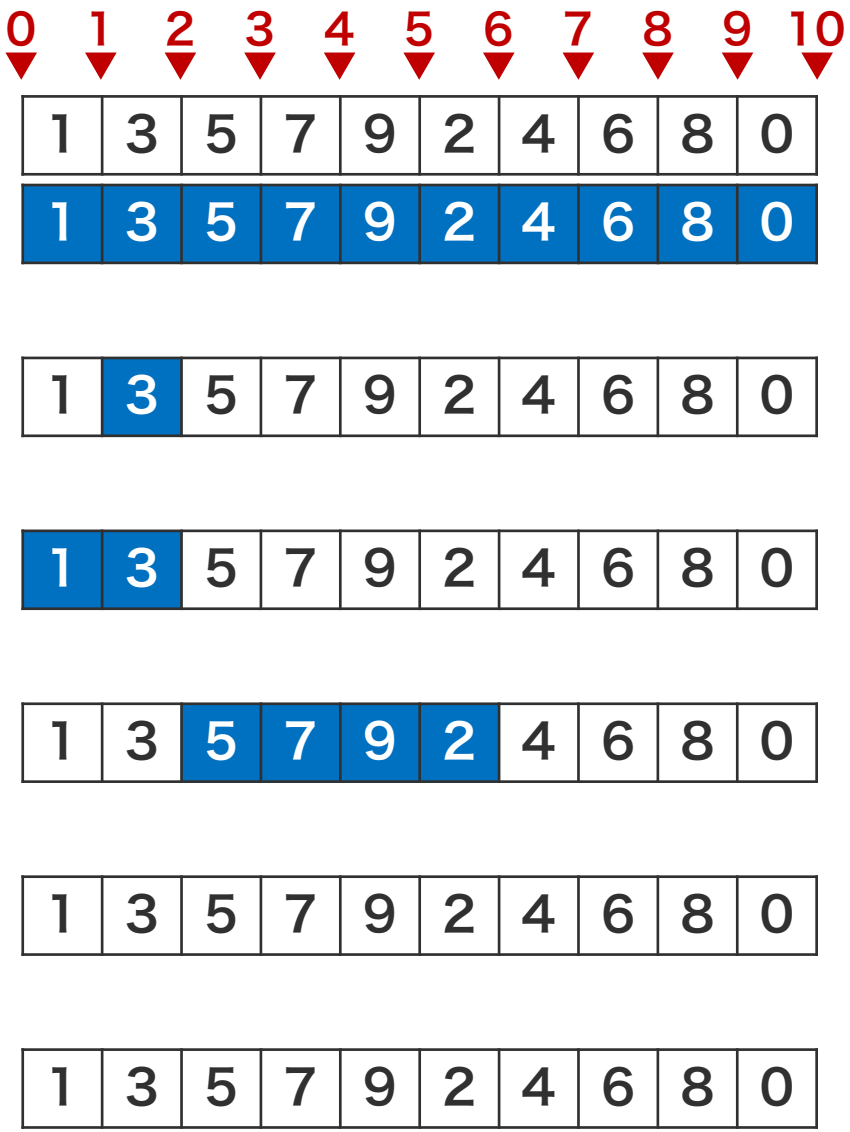

```
a = [1, 3, 5, 7, 9, 2, 4, 6, 8, 0]


a
# [1, 3, 5, 7, 9, 2, 4, 6, 8, 0]

a[1]
# 3

a[0:2]


a[2:6]


a[:4]


a[5:]
```

リスト



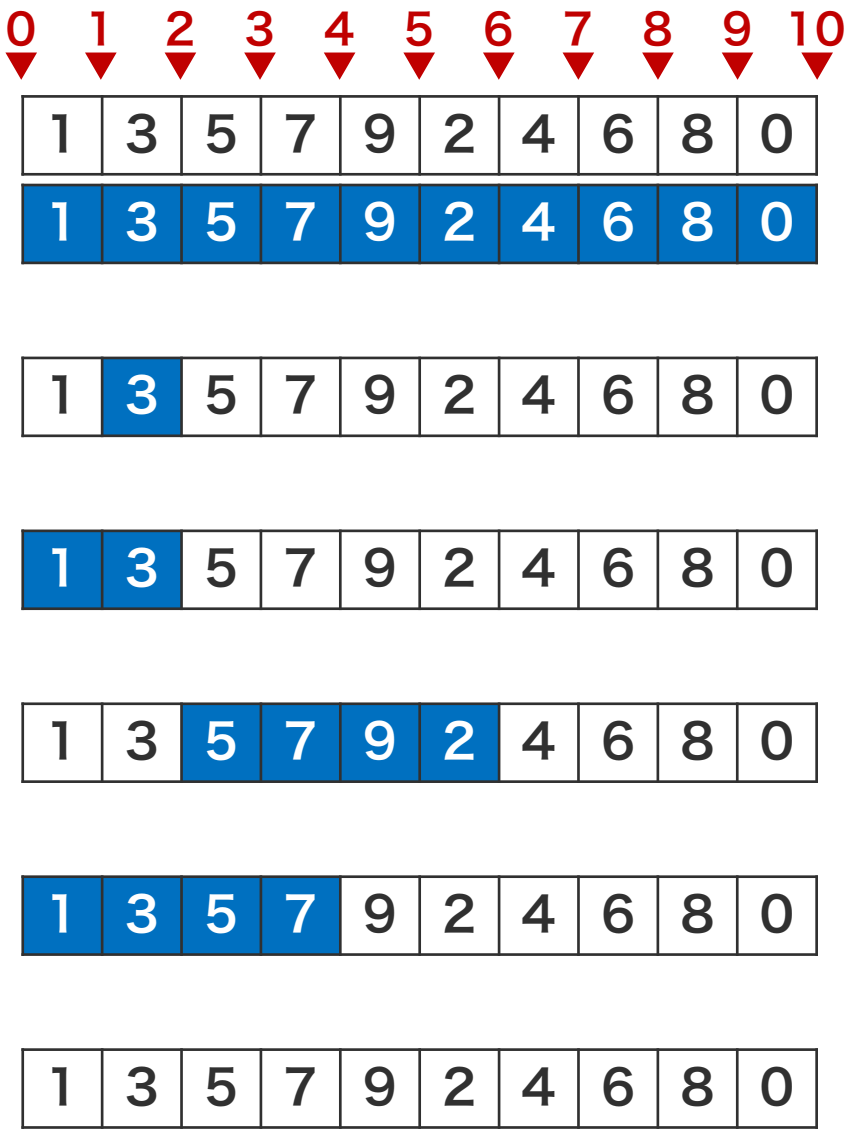

```
a = [1, 3, 5, 7, 9, 2, 4, 6, 8, 0]


a
# [1, 3, 5, 7, 9, 2, 4, 6, 8, 0]

a[1]
# 3

a[0:2]
# [1, 3]

a[2:6]


a[:4]


a[5:]
```

リスト



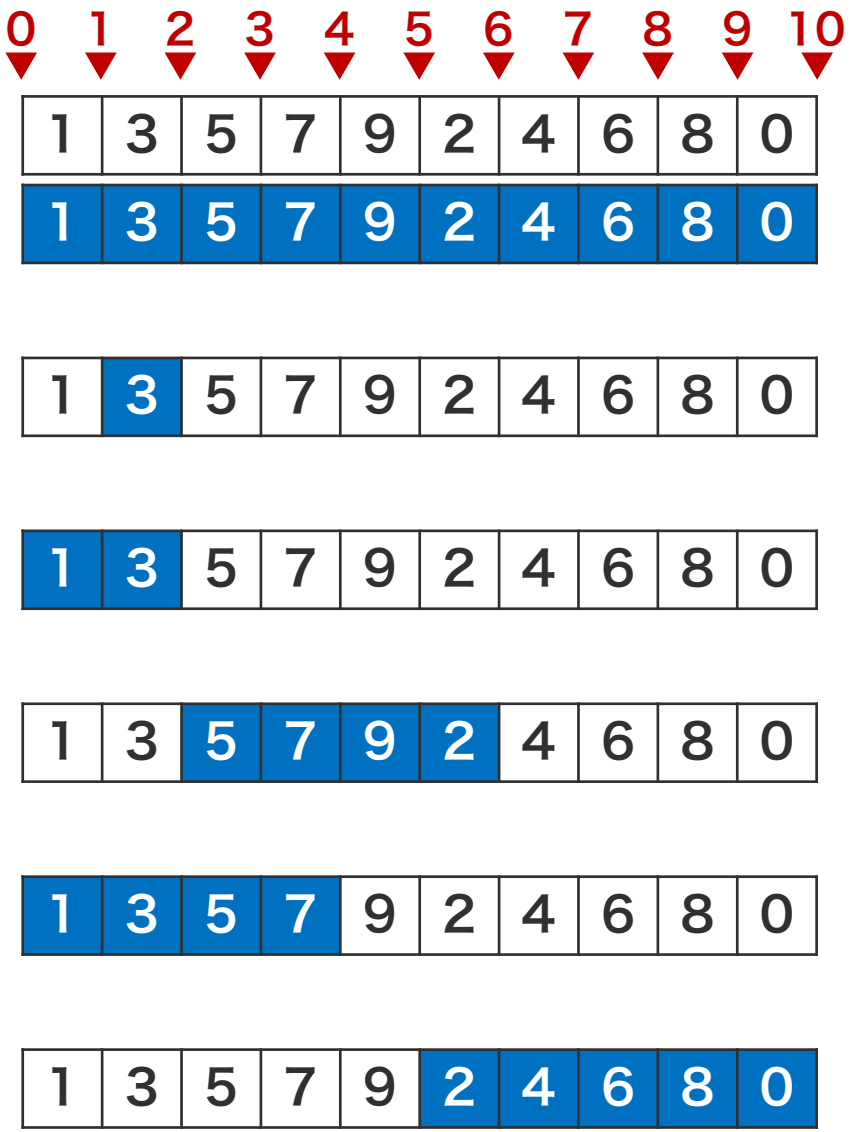

```
a = [1, 3, 5, 7, 9, 2, 4, 6, 8, 0]


a
# [1, 3, 5, 7, 9, 2, 4, 6, 8, 0]

a[1]
# 3

a[0:2]
# [1, 3]

a[2:6]
# [5, 7, 9, 2]

a[:4]


a[5:]
```

リスト




```
a = [1, 3, 5, 7, 9, 2, 4, 6, 8, 0]


a
# [1, 3, 5, 7, 9, 2, 4, 6, 8, 0]

a[1]
# 3

a[0:2]
# [1, 3]

a[2:6]
# [5, 7, 9, 2]

a[:4]
# [1, 3, 5, 7]

a[5:]
```

リスト




```
a = [1, 3, 5, 7, 9, 2, 4, 6, 8, 0]


a
# [1, 3, 5, 7, 9, 2, 4, 6, 8, 0]

a[1]
# 3

a[0:2]
# [1, 3]

a[2:6]
# [5, 7, 9, 2]

a[:4]
# [1, 3, 5, 7]

a[5:]
# [2, 4, 6, 8, 0]
```

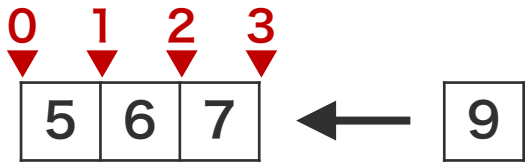
リスト操作

すでに作られたリストに新しい要素を追加することができる。リストの後尾に要素を追加するには append 関数（メソッド）を使用する。また、リストの先頭あるいは指定した位置に要素を挿入するには insert 関数（メソッド）を使用する。

関数	機能
append	リストの後尾に要素を 1 つだけ追加する。
extend	リストの後尾に要素を複数個追加する。
insert	リストの位置 i に要素を 1 つだけ挿入する。
pop	リストの位置 i にある要素を削除する。

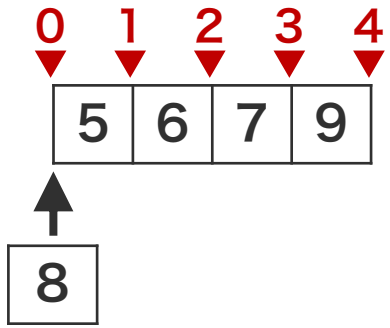
```
a = [5, 6, 7]

a.append(9)
a



a.insert(0, 8)
a



a.insert(2, 1)
a



a.append(4)
a
```

リスト操作



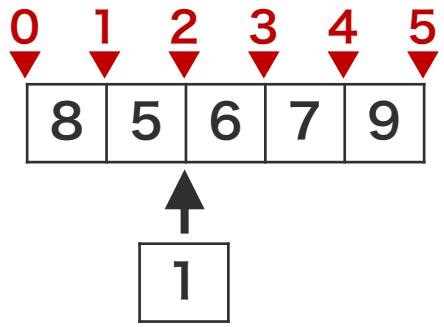

```
a = [5, 6, 7]

a.append(9)
a
# [5, 6, 7, 9]


a.insert(0, 8)
a



a.insert(2, 1)
a



a.append(4)
a
```

リスト操作



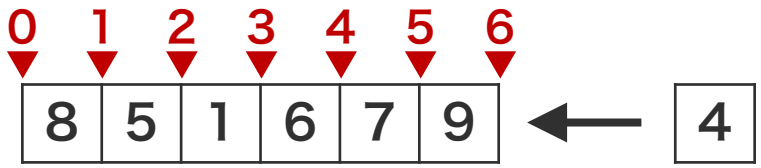

```
a = [5, 6, 7]

a.append(9)
a
# [5, 6, 7, 9]


a.insert(0, 8)
a
# [8, 5, 6, 7, 9]


a.insert(2, 1)
a



a.append(4)
a
```

リスト操作




```
a = [5, 6, 7]

a.append(9)
a
# [5, 6, 7, 9]


a.insert(0, 8)
a
# [8, 5, 6, 7, 9]


a.insert(2, 1)
a
# [8, 5, 1, 6, 7, 9]


a.append(4)
a
```

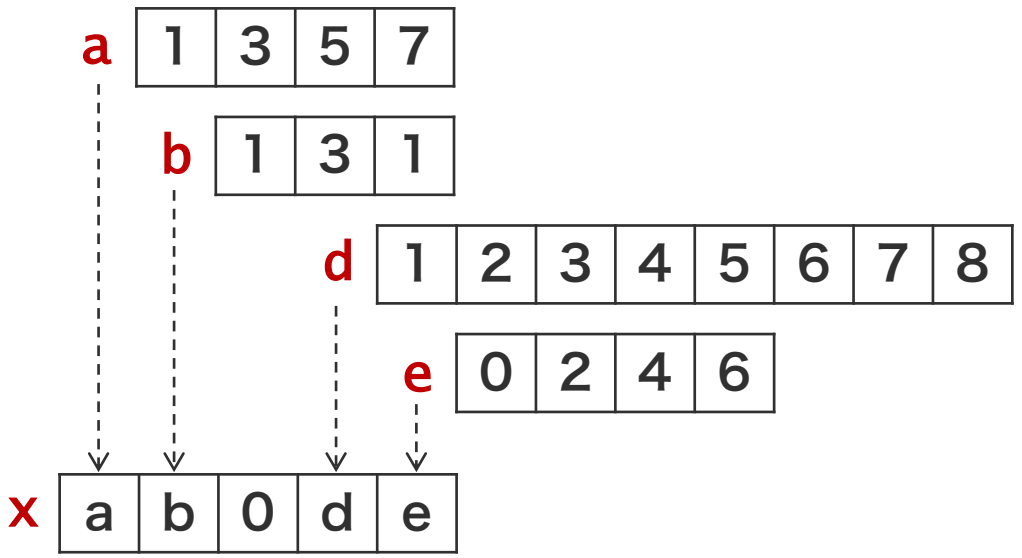
リスト操作




```
a = [5, 6, 7]

a.append(9)
a
# [5, 6, 7, 9]


a.insert(0, 8)
a
# [8, 5, 6, 7, 9]


a.insert(2, 1)
a
# [8, 5, 1, 6, 7, 9]


a.append(4)
a
# [8, 5, 1, 6, 7, 9, 4]
```

リストのリスト

リストは複数の要素を持つことができる。これまでに見てきた各要素は、一つ一つの数値であった。実は、リストに代入できる要素は、Python のオブジェクトであればよい。つまり、リストの中に、リストというオブジェクトを代入することもできる。

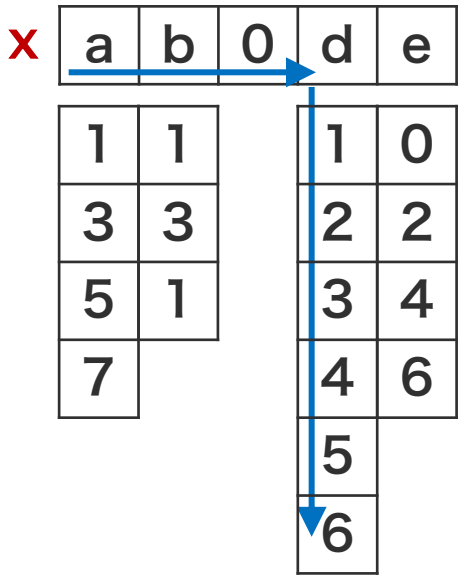



```
a = [1, 3, 5, 7]
b = [1, 3, 1]
d = [1, 2, 3, 4, 5, 6, 7, 8]
e = [0, 2, 4, 6]

x = [a, b, 0, d, e]
```

二次元リスト

リストは複数の要素を持つことができる。これまでに見てきた各要素は、一つ一つの数値であった。実は、リストに代入できる要素は、Python のオブジェクトであればよい。つまり、リストの中に、リストを代入することもできる。このようなリストを二次元リスト、多次元リストと呼んだりする。

```
a = [1, 3, 5, 7]
b = [1, 3, 1]
d = [1, 2, 3, 4, 5, 6, 7, 8]
e = [0, 2, 4, 6]

x = [a, b, 0, d, e]

x[3][5]
# 6
```



基本オブジェクト

- スカラー
- リスト
- ディクショナリ

ディクショナリ

ディクショナリは、リストと同じく、複数の要素を束ねて保存できる変数である。リストは各要素を index と呼ばれる添字で管理しているのに対して、ディクショナリは各要素を名前（キー）で管理している。ディクショナリには順序の概念が存在しない。また、キーは文字列であるため、そのまま入力するとオブジェクト名に間違えられる。そのため、キーを " または ' で囲む必要がある。

樹木	櫟	樫	楢
どんぐり			
重さ	2.0	1.3	2.8

1. キーと値をペアで与える
2. 複数のペアをカンマ区切りで並べる
3. 波括弧 (curly bracket) で全体をまとめる

```
a = {
'buna': 2.0,
'kashi': 1.3,
'nara': 2.8
}
```

ディクショナリ

ディクショナリの要素を取り出すときは、変数名に続けて、角括弧を書き、角括弧の中にキー入れる。括弧の形を変更することはできない。

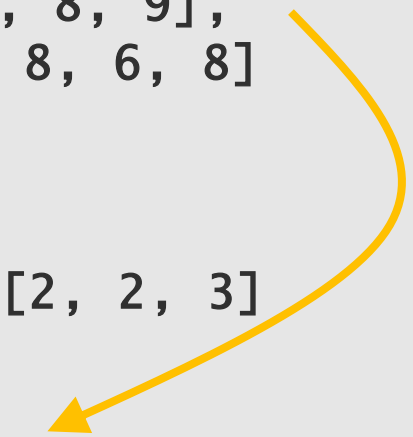
```
weights = {
    'buna': 2.0,
    'kashi': 1.3,
    'nara': 2.8
}


weights['buna']
# 2.0


weights['kashi']
# 1.3


weights['shii']
# KeyError: 'shii'
```

存在しないキーを与えると、キーが見つからないというKeyError が起こる。

ディクショナリ

既存のディクショナリの値を更新するときは、キーを指定し、代入演算子で新しい値を代入する。

```
weights = {
    'buna': [12, 11, 13, 14, 13],
    'kashi': [9, 9, 8, 9],
    'nara': [6, 7, 8, 6, 8]
}


weights['buna'] = [2, 2, 3]

weights
# {'buna': [2, 2, 3], 'kashi': [9, 9,
8, 9], 'nara': [6, 7, 8, 6, 8]}
```



ディクショナリ

既存のディクショナリの値を更新するときは、キーを指定し、代入演算子で新しい値を代入する。また、ディクショナリに存在していないキーに対して、値を代入すると、そのキーと値のペアは、ディクショナリに新規追加される。

```
weights = {
    'buna': [12, 11, 13, 14, 13],
    'kashi': [9, 9, 8, 9],
    'nara': [6, 7, 8, 6, 8]
}


weights['shii'] = [2, 2, 3]

weights
# {'buna': [12, 11, 13, 14, 13],
'kashi': [9, 9, 8, 9], 'nara': [6, 7,
8, 6, 8], 'shii': [2, 2, 3]}
```

問題 O3-1

ペプチド PGWR の分子量を計算せよ。ただし、各アミノ酸の分子量は以下の表に示したものを使うこと。

アミノ酸	分子量
PRO / P	115.13
GLY / G	75.07
TRP / W	204.23
ARG / R	174.20

```
aa2mw = {
  'P': 115.13,
  'G':  75.07,
  'W': 204.23,
  'R': 174.20
}
```

農学生命情報科学特論Ⅰ

1

- プログラミング言語
- 基本オブジェクト
- 基本文法

基本文法

- ○ 予約語
- ○ 条件構文
- ○ 繰り返し構文

予約語

条件判定		条件構文	繰り返し構文	関数		例外処理	いろいろ	
True	is	if	for	def	import	try	lambda	async
False	not	elif	while	return	from	except	del	await
None	in	else	pass	yield	as	finally	zip	global
	or		break		class	raise	assert	with
	and		continue				nonlocal	
			enumerate					

基本文法

- ○ 予約語
- ○ 条件構文
- ○ 繰り返し構文

if 構文

条件構文は、条件に応じて処理を切り分けたい場合に利用する構文の一つである。例えば、「晴れていれば公園に行くが、それ以外ならば家にこもる」、「パン屋さんでパンを購入して、イートインならば消費税 10%、持ち帰りならば消費税 8% にする」などのようなことを表現したりする際に利用する。これらのことをプログラミング言語らしく（擬似コード）表現すると右のようになる 。

```
weather = 'sunny'


if weather is 'sunny',
    go_to_park()

otherwise,
    stay_home()
```

```
where_to_eat = 'takeout'
amount = 100

if where_to_eat is 'eatin',
    tax = 0.10

otherwise,
    tax = 0.08

amount = amount * (1 + tax)
```

if 構文

Python の条件構文は if、elif、else などの単語を使う。Python の条件構文は必ず if から始まる。条件構文の 1 行目には、if とともに条件を書く。条件判定後の処理は、2 行目以降に書く。ただし、条件判定後の処理は、条件構文の一部であることを明示するために、行の先頭にインデントを入れる。

もし持ち帰りならば ▶

もし店内で食事するならば ▶

```
a = 180
b = 120
s = 0


takeout = True

eatin = False


if takeout is True :
    s = (a + b) * 1.08


if eatin is True:
    s = (a + b) * 1.10


s
# 324.0
```

if 構文

```
a = 180
b = 120
s = 0
takeout = True

if takeout is True :
    s = (a + b) * 1.08

print(s)
```

判定条件

条件構文の開始 ▶

インデント

条件構文の終了 ▶

条件構文ブロック

インデントの数で、条件構文ブロックが終了しているかどうかを判定している。このブロックのことをスコープと呼ぶこともある。

インデント

インデントは、タブまたはスペースキーで入力する。

- タブ 1 個分
- スペース 4 個分
- スペース 2 個分

英語キーボード

日本語キーボード

if 構文

```
a = 180
b = 120
s = 0
takeout = True

if takeout is True :
    s = (a + b) * 1.08

print(s)
```

判定条件

条件構文の開始 ▶

インデント

条件構文の終了 ▶

真判定の場合は、「is　True」を省略するのが正しい書き方。

```
a = 180
b = 120
s = 0
takeout = True

if takeout:
    s = (a + b) * 1.08

print(s)
```

論理演算子

条件構文は、与えられた条件が真（True）であるかどうかを判定して、分岐処理を行う構文である。条件を、次のような演算子を使って演算した結果とすることが多い。

論理演算子	処理
a == b	a と b が等しいならば True
a != b	a と b が等しくなければ True
a > b	a が b よりも大きければ True
a >= b	a が b 以上ならば True
a < b	a が b よりも小さければ True
a <= b	a が b 以下ならば True

```
a = 4
b = 3
c = 0
d = 0



if a > 3:
    c = 1
c
# 1



if b > 10:
    d = 1
d
# 0
```

問題 S1-1

次のプログラムを実行した時に、最後に出力される金額 n はいくらになるのかを答えよ。

```
apple = 200
melon = 100
takeout = True

s = apple + melon
n = 0

if takeout:
    n = s * 1.08


n
```

変数の使い回し

代入演算子の左右に同じ変数名を使用できる。代入演算子は、まず右側の式の計算を行い、次にその計算結果を左の変数に代入する、という働きを持つ。代入演算子の働きに順序があるため、その左右に同じ変数を使用してもエラーが生じない。

```
s = 1200
takeout = True
n = 0
m = 0

if s > 1000:
    n = s * 0.90


if takeout:
    m = n * 1.08



m
# 1166.4
```

```
s = 1200
takeout = True



if s > 1000:
    s = s * 0.90


if takeout:
    s = s * 1.08



s
# 1166.4
```

問題 S1-2

次のプログラムを実行した時に、最後に出力される金額 n はいくらになるのかを答えよ。

```
apple = 200
melon = 100
takeout = True

n = apple * 4 + melon * 2


if n > 1000:
    n = n * 0.95


 if takeout:
    n = n * 1.08


n
```

```
apple = 200
melon = 100
takeout = False

n = apple * 4 + melon * 2


if n => 1000:
    n = n * 0.95


if takeout:
    n = n * 1.08


n
```

問題 S1-3

イートインスペースのあるパン屋さんで、アップルパイを 1 つとメロンパンを 1 つ、購入して持ち帰る。商品購入時に、可能であればクーポン券を使う。このとき、合計金額を計算せよ。

ただし、

- アップルパイは 1 つ 180 円、メロンパンは 1 つ 120 円である。
- クーポン券は、合計金額が 1000 円を超えた場合のみに使用でき、その際、5% OFF される。
- 店内で食べる場合の消費税率を 10% で、持ち帰りの場合の消費税率を 8% とする。

```
apple = 180
melon = 120
takeout = True
eatin = False
```

入れ子構造

複数の条件に対して条件判定を行うとき、条件構文を二つ重ねた入れ子構造にすることで実現できる。例えば、「毎月 20 日に 1000 円以上の買い物をした時に 10% 値引きする」といった処理は、まず、会計日が 20 日かどうかを判定して、会計日が 20 日であれば、次に購入金額が 1000 円以上かどうかを判定する。これを if 構文で書くと右のようになる 。

```
apple = 200
melon = 100

date = 20


n = apple * 4 + melon* 2


if date == 20:
    if n >= 1000:
        n = n * 0.90
```

条件構文（if）の中の条件構文（if）の中の処理であるため、インデントは 2 個分必要。

```
n
# 900.0
```

入れ子構造

一つの if 構文の下に、複数の if 構文を入れても大丈夫。例えば、毎月 20 日の買い物で、1000 円以上ならば 10% OFF、1000 円未満ならば 5% OFF のような処理を行う場合は、右のように 2 つの if 構文を続けて書くことで実現できる。

```
apple = 200
melon = 100

date = 20


n = apple * 4 + melon* 2


if date == 20:
    if n >= 1000:
        n = n * 0.90
    if n < 1000:
        n = n * 0.95


n
# 900.0
```

20 日かつ 1000 円以上の買い物の場合に 10% OFF。

20 日かつ 1000 円未満の買い物の場合に 5% OFF。

論理演算

複数の条件を同時に判断するときは、入れ子構造の条件構文を利用するほか、複数の条件を論理演算した結果を条件構文の判定条件として使うこともできる。論理演算でよく使われる演算として、論理積と論理和である。論理積は、英語の AND に相当するものである。論理和は、英語の OR に相当するものである。

論理演算子	演算
and	論理積
&	論理積
or	論理和
|	論理和
^	排他的論理和

```
apple = 200
melon = 100
date = 20

n = apple * 4 + melon* 2

if date == 20:
    if n >= 1000:
        n = n * 0.90

n
# 900.0

m = apple * 4 + melon* 2

if (date == 20) and (m >= 1000):
    m = m * 0.90

m
# 900.0
```

if 構文の入れ子構造

二つの判定条件を論理演算している

問題 S1-4

下に示した 2 種類の割引処理の違いを、論理演算子に注意しながら、説明してみてください。それぞれがどのような条件で割引されるのか。

```
coupon = True

s = 1100




if (coupon)  and  (s >= 1000) :

    s = s * 0.95

s
```

```
coupon = True

s = 1100




if (coupon)  or  (s >= 1000) :

    s = s * 0.95

s
```

問題 S1-5

赤色で書かれたオブジェクトが保持している値を答えよ。

```
a = 4
b = 3
c = 0

if (a > 0) and (b > 0):
    c = 1

c
```

```
a = 1
b = 4
c = 0

if (a > 0) or (b < 2):
    c = 1

c
```

if-else 構文

これまでに見てきた条件構文は「もし〜ならば・・・をする」のように、ある条件を満たせば、特定の 1 つの処理を行うものであった。これに対して、ある条件を満たしたときに処理 A を、満たさなかった時に処理 B を行いたい場合は、if 構文を続けて 2 つ書けば実現できる。

```
a = 180
b = 120
s = 0


takeout = True




if takeout:
    s = (a + b) * 1.08


if not takeout:
    s = (a + b) * 1.10

s
# 324
```

if-else 構文

ある条件の真偽判定に基づいて処理を分けたい場合は、条件構文 if を 2 つ書くことで対応できる。しかし、if 構文を 2 つ続けて書くことで、同じ条件を 2 回判断していることになる。理論的にも、処理的にも煩雑である。このような場合は、if と else を用いて条件構文を作成すると、理論的にも処理的にもシンプルになる。

```
a = 180
b = 120
s = 0


takeout = True



if takeout:
    s = (a + b) * 1.08


if not takeout:
    s = (a + b) * 1.10


s
# 324.0
```



```
a = 180
b = 120
s = 0


takeout = True



if takeout:
    s = (a + b) * 1.08


else:
    s = (a + b) * 1.10


s
# 324.0
```

if-else 構文

ある条件の真偽判定に基づいて処理を分けたい場合は、条件構文 if を 2 つ書くことで対応できる。しかし、if 構文を 2 つ続けて書くことで、同じ条件を 2 回判断していることになる。理論的にも、処理的にも煩雑である。このような場合は、if と else を用いて条件構文を作成すると、理論的にも処理的にもシンプルになる。

```
a = 180
b = 120
s = 0


takeout = True



if takeout:
    s = (a + b) * 1.08

else:
    s = (a + b) * 1.10

s
# 324.0
```

条件判定 ▶

条件成立時に実行

条件不成立時に実行

問題 S1-6

イートインスペースのあるパン屋さんで、アップルパイを 1 つとメロンパンを 1 つ、食パンを 2 袋購入して持ち帰った。このとき、合計金額を計算せよ。なお、商品購入時に、可能であればクーポン券を使用する。

ただし、

- アップルパイは 1 つ 180 円、メロンパンは 1 つ 120 円、食パンは 1 袋 400 円である。
- クーポン券は、合計金額が 1000 円を超えた場合のみに、使用でき、その際、5% OFF される。
- 店内で食べる場合の消費税率を 10% で、持ち帰りの場合の消費税率を 8% とする。
- 店内で食べるか持ち帰るかの判断を if-else 文で判断し、また、クーポンが使えるかどうかの判断を if 文で判断せよ。

```
apple = 180
melon = 120
plain = 400

s = apple + melon + plain * 2
```

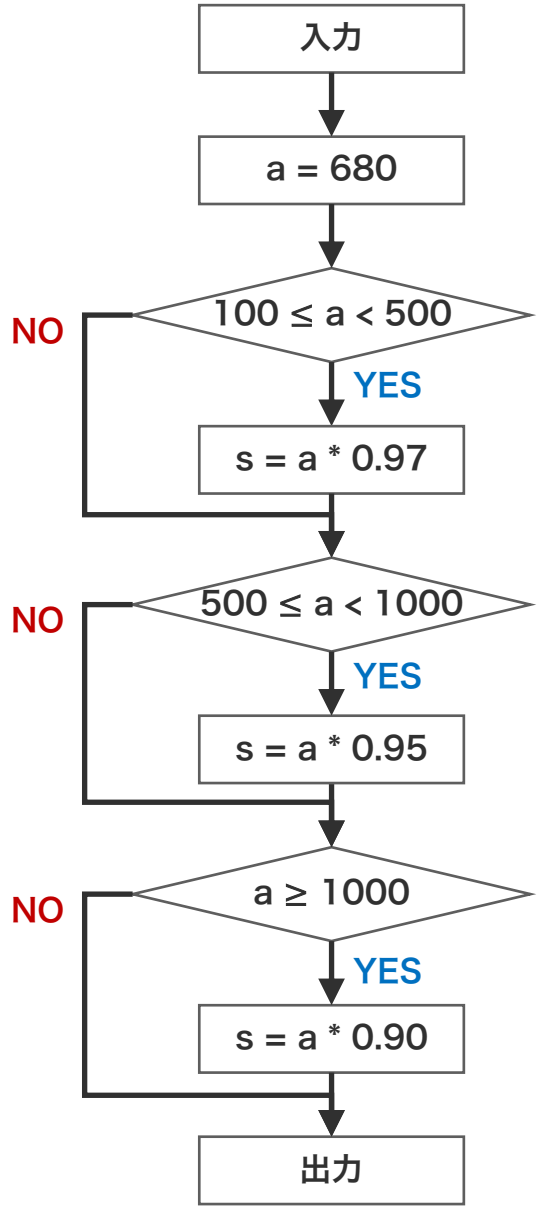
if-elif-else 構文

1 つの変数に対して複数の閾値を設けて条件判定したい場合は、複数の if 構文を使って書くことができる。この場合、ロジックが複雑になることに注意する必要がある。

小計	割引
100 円以上 500 円未満	3%
500 円以上 1000 円未満	5%
1000 円以上	10%

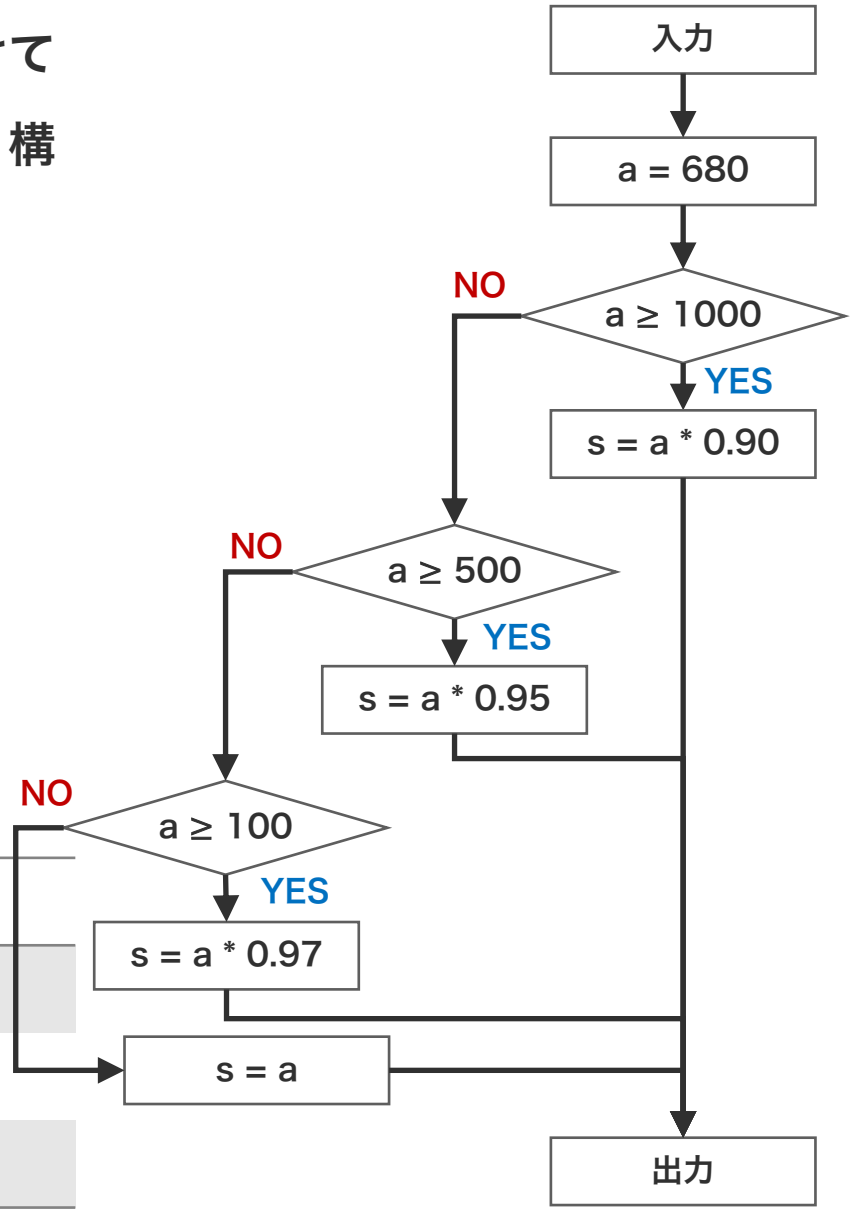
```
a = 680
s = 0


if 100 <= a and a < 500:
    s = a * 0.97

if 500 <= a and a < 1000:
    s = a * 0.95

if a >= 1000:
    s = a * 0.90



s
# 646.0
```

このプログラムでは 100 円未満の場合 s はゼロのままであることに注意！



if-elif-else 構文

1 つの変数に対して複数の閾値を設けて条件判定したい場合は、if-elif-else 構文を利用すると便利な場合がある。



小計	割引
100 円以上 500 円未満	3%
500 円以上 1000 円未満	5%
1000 円以上	10%

```
a = 680
s = 0


if a >= 1000:
    s = a * 0.90

elif a >= 500:
    s = a * 0.95

elif a >= 100:
    s = a * 0.97

else:
    s = a


s
# 646.0
```

if-elif-else 構文

1 つの変数に対して複数の閾値を設けて条件判定したい場合は、if-elif-else 構文を利用すると便利な場合がある。

小計	割引
100 円以上 500 円未満	3%
500 円以上 1000 円未満	5%
1000 円以上	10%

```
a = 680
s = 0


if 100 <= a and a < 500:
    s = a * 0.97

if 500 <= a and a < 1000:
    s = a * 0.95

if a >= 1000:
    s = a * 0.90



s
# 646.0
```

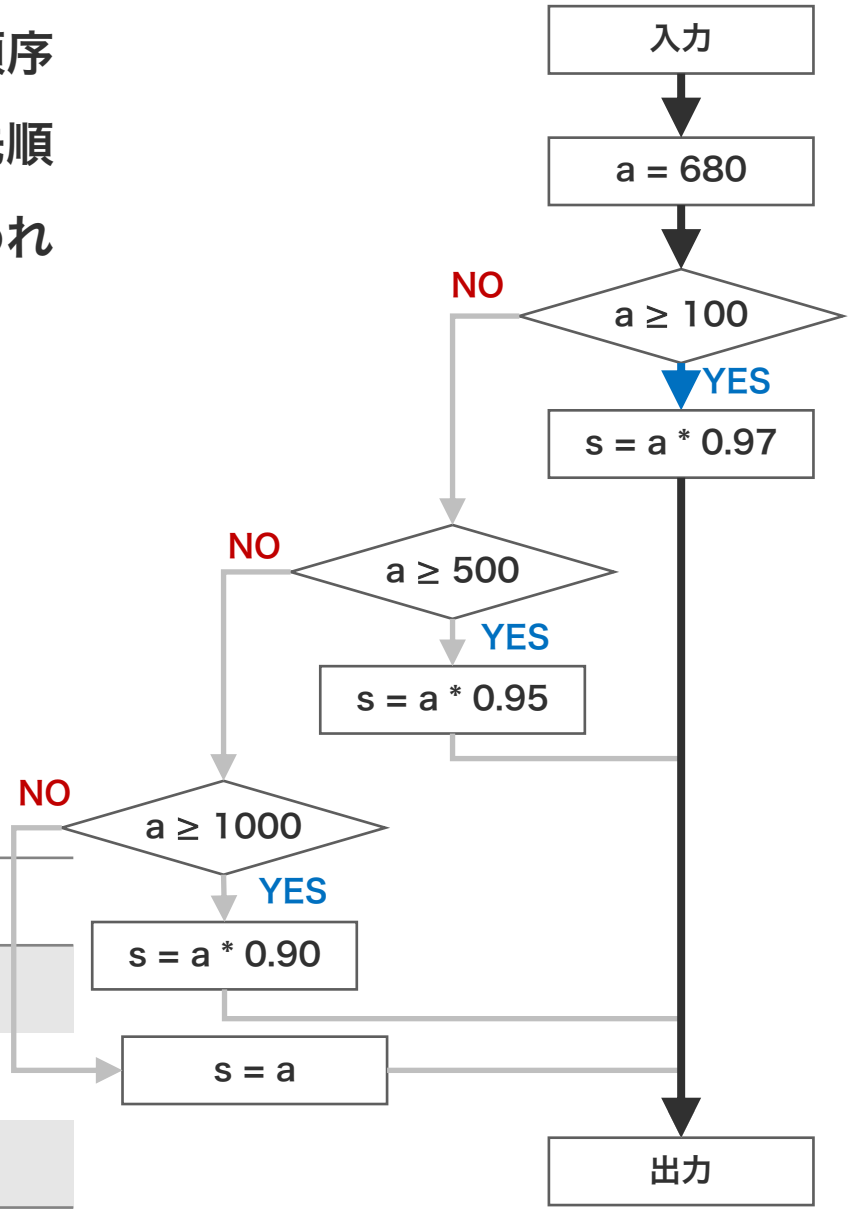
このプログラムでは 100 円未満の場合 s はゼロのままであることに注意！

```
a = 680
s = 0


if a >= 1000:
    s = a * 0.90

elif a >= 500:
    s = a * 0.95

elif a >= 100:
    s = a * 0.97

else:
    s = a



s
# 646.0
```

上記条件のいずれも満たさないとき、s に a の値を代入する。

if-elif-else 構文

条件が複数あるとき、各条件の優先順序に注意を払う必要がある。条件の優先順序を間違えると、想定外の処理が行われることがある。



小計	割引
100 円以上 500 円未満	3%
500 円以上 1000 円未満	5%
1000 円以上	10%

```
a = 680
s = 0


if a >= 100:
    s = a * 0.97

elif a >= 500:
    s = a * 0.95

elif a >= 1000:
    s = a * 0.90

else:
    s = a


s
# 659.6
```

基本文法

- ○ 予約語
- ○ 条件構文
- ○ 繰り返し構文

繰り返し構文

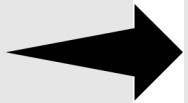
リスト a の各要素の合計を計算して、変数 n に代入する手順を考えよ。ただし、一度に 2 数値の足し算しかできないものとする。

```
a = [2, 3, 1, 5, 10]
n = 0




n
# 21
```

繰り返し構文

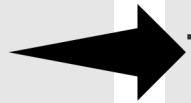
リスト a の各要素の合計を計算して、変数 n に代入する手順を考えよ。ただし、一度に 2 数値の足し算しかできないものとする。

```
a = [2, 3, 1, 5, 10]
n = 0



n = n + 2
n = n + 3
n = n + 1
n = n + 5
n = n + 10




n
# 21
```



```
a = [2, 3, 1, 5, 10]
n = 0



n = n + a[0]
n = n + a[1]
n = n + a[2]
n = n + a[3]
n = n + a[4]




n
# 21
```

i = 0, 1, ..., 4 のとき、
n = n + a[i]
の繰り返しである。



```
a = [2, 3, 1, 5, 10]
n = 0

for i in range(5):
    n = n + a[i]




n
# 21
```

i を 0, 1, ..., 4 に変化させながら、以下の処理を行う構文。

for 構文

繰り返し構文には while 構文と for 構文の 2 種類がある。このうち、for 構文は「n 回繰り返す」命令文である。for 構文を使用するとき、繰り返し回数 n をはじめに指定する必要がある。

```
a = [2, 3, 1, 5, 10]
n = 0

for i in range(5):
    n = n + a[i]

n
# 21



m = 0

for i in range(len(a)):
    m = m + a[i]

m
# 21
```

i = 0, 1, 2, 3, 4 として計 5 回繰り返せ。

for 構文

繰り返し回数　繰り返す範囲

繰り返し構文の開始 ▶

インデント

繰り返し構文の終了 ▶

```
a = [2, 3, 1, 5, 10]
n = 0

for i in range(5) :
    n = n + a[i]



n
# 21
```

繰り返し構文ブロック

インデントの数で、構文ブロックが終了しているかどうかを判定している。

for 構文

```
a = [2, 3, 1, 5, 10]
n = 0

for x in a :
    n = n + x


n
# 21
```

一時変数　繰り返す対象

繰り返し構文の開始 ▶

インデント

繰り返し構文の終了 ▶

リストの各要素に対して繰り返し処理を行う場合、繰り返す対象としてリスト名をそのまま与えることが一般的である。

繰り返し構文ブロック

インデントの数で、構文ブロックが終了しているかどうかを判定している。

問題 S2-1

for 構文を使用して、リスト a の平均値を求めよ。

（len 関数および sum 関数を使わないこと）

```
a = [1, 9, 2, 8, 3]
```

問題 S2-2

for 構文を使用して、リスト a の中の最大値を変数 M に代入するプログラムを書け。

```
a = [1, 9, 2, 0, 4, 7]
```

問題 S2-3

for 構文を使用して、リスト a の要素のうち奇数要素の個数を求めて変数 k に代入せよ。

```
a = [1, 9, 2, 0, 4, 7]
k = 0
```

while 構文

繰り返し構文には while 構文と for 構文の二種類がある。このうち、while 構文は、「与えた条件を満たす限り同じ処理を繰り返す」命令文である。例えば、「リスト a の末端に達するまで、a の各要素を n に足す」処理に利用できる。for 構文で書ける処理は while 構文でも書ける。右は、リスト a の各要素の合計を計算する処理を、for 構文および while 構文で書いた例を示している。

```
a = [2, 3, 1, 5, 10]
n = 0

for i in range(5):
    n = n + a[i]

n
# 21
```

▶

```
m = 0

i = 0
while i < 5:
    m = m + a[i]
    i = i + 1

m
# 21
```

while 構文

```
a = [2, 3, 1, 5, 10]
n = 0

i = 0
while i < 5 :
    n = n + a[i]
    i = i + 1


n
# 21
```

判定条件

繰り返し構文の開始 ▶

インデント

繰り返し構文の終了 ▶

繰り返し構文ブロック

インデントの数で、構文ブロックが終了しているかどうかを判定している。

問題 S2-4

while 構文を使用して、リスト a の平均値を求めよ。

（len 関数および sum 関数を使わないこと）

```
a = [1, 9, 2, 8, 3]
```

問題 S2-5

while 構文を使用して、リスト a の中の最大値を M に代入するプログラムを書け。

```
a = [1, 9, 2, 0, 4, 7]
```

for 構文とwhile 構文

for 構文

- 予め繰り返し回数の上限を設定する。
- 繰り返し回数が決まっている作業に利用する。
  - 前日の売り上げ処理。
  - 試験の自動採点処理。

while 構文

- 繰り返し回数の上限はない。
- 繰り返し回数が不明な作業に利用する。
  - レジで「小計」ボタンが押されるまで、スキャンされた商品の金額を足していく。
  - 貯金がなくなるまで、買い物を続ける。
  - 1 時間間隔で、郵便ポストの中に手紙類が入っているかどうかを確認する。