Python の文法など: 最低限 $+\alpha$
Sage の使用を視野に入れた Python 入門.

沼田 泰英[*1][*2]

2014-12-19 14:22:07

[*1] numata at stat.t.u-tokyo.ac.jp
[*2] nu+y at math.shinshu-u.ac.jp

西山絢太君へ.

■この資料について　数式処理システム Sage を使用することを視野に入れ Python のプログラミングに必要な事項について概説します. 大雑把に言うと, Sage は Python に数学に関するライブラリを被せたものであるので, Python の文法などを知ることは Sage を使う上で手助けになるかもしれません. Sage の使用を目標としているため, すべての事項について説明しているわけではありません. 例えば, クラスを定義する方法などについては触れていません. 必要に応じて別の文献を参照するなどしてください.

Python の一部の機能は Sage によって上書きされているため, Sage での動作は本来の Python での動作と異なる場合があります. 動作が異なる場合に本来の Python の動作を記述する際には, 'pure Python では' と断りをいれています.

Acknowledgment. これは, 2012 年 5 月 26 日から 27 日の日程で, 九州大学で開催された Sage Days 39 のために書いた原稿を加筆したものです. 草稿に目を通していただいきアドバイスをして下さった稚内北星学園大学の片山雄大氏, 大阪大学の西山絢太氏, 福岡大学の濱田龍義氏, 国立情報学研究所の松井鉄史氏, 九州大学の横山俊一氏に感謝します. (五十音順, 所属は当時)

目次

第 1 章

基本的なこと

‘とりあえずプログラミングをする’ のに必要な事項について, ここでは説明をします. より綺麗な, もしくは, より効率的なプログラミングをするために必要な事項については, 後ろの章または他の文献を参照してください.

1.1 まずは使ってみる

とりあえず, Python を使ってみることにします. プログラミングの入門では, ‘まず最初に “Hello world.” という文字列を出力しなければいけない’ という掟がありますので, そこから始めます. Python には `print` という出力を行うための ‘命令’ があり, “Hello world.” という文字列を出力するプログラムは, 例えば次のようになります.

```
print 'Hello world.'
```

このたった 1 行のプログラムを Python で実行するための方法は, 大雑把に分けると 2 つあります. まずひとつ目は, Python のインタプリタを立ち上げ, そこで対話的に実行する方法です. [*1] ターミナルから `python` というコマンドを実行しましょう. すると, いくつかの情報が表示された後, 次のような入力を待ち受ける状態になると思います.

```
>>>
```

[*1] Python のインタプリタをインストールする方法は他の文献に譲ります. 例えば MacOS や標準的な Linux といった OS を使っているのであれば, デフォルトでインストールされており, 特に設定や追加のインストールをすることなく, ここで紹介する方法で使うことが出来ると思います. 他の OS については自分でインストールし, 設定を行う必要があるかもしれません. 必要に応じて自分の設定に読み替えてください.

ここから, 先程の `print 'Hello world.'` を入力し Enter キーを押すと, 入力した行が実行され `Hello world.` と表示され, 再び入力待ちとなると思います.

```
>>> print 'Hello world.'
Hello world.
>>>
```

ここから抜けるには, `exit()` を実行するか, Ctrl を押しながら d を押します.

もうひとつの方法は, テキストファイルとしてプログラムを保存し, それを Python で読み込むという方法です. まずテキストエディタを使い,

```
print 'Hello world.'
```

という 1 文を書いたファイルを作成し, 例えば `test001.py` という名前で保存します. そして `python test001.py` というコマンドをターミナルから実行します. すると `Hello world.` が表示されると思います. この場合は, プログラムが終了したらそれを呼び出した Python も自動的に終了します.

どちらの方法で実行しても構いません. どんなプログラムを実行したのか記録が残り後々使いまわせるという点でファイルを用意し実行した方が便利かもしれませんし, ファイルをわざわざ作成しなくても良いという点でインタプリタで対話的に実行した方が便利かもしれません. 自分に合った方法で実行してください.

とりあえず実行方法は説明したので, 徐々に詳しいことを説明していきたいと思います. `print` を使えば数を表示することもできます. 例えば, 次のプログラムを実行すると `0` と `2` を出力します.

```
print 0
print 1+1
```

実行結果の例

```
0
2
```

`print` を使うと出力後に改行をします. 改行をさせたくない時は最後に ‘`,`’ を置きます.

```
print 0,
print 1+1
```

実行結果の例

```
0 2
```

1.2 変数への代入

変数に値を代入するには次のようにします.

```
a = 3
print a
```

実行結果の例

```
3
```

現在の a の値に 5 を足して得られる値を a に代入するには次のようにします.

```
a = a + 5
print a
```

実行結果の例

```
8
```

変数の名前は大文字と小文字を区別します. 基本的には a–z, A–Z, で始まらないといけません. 冒頭以外には, a–z, A–Z, の他に 0–9, _ なども使えます. また, システムで既に定義されている文字列は使えません. 例えば, 次のものは使わない方が無難です.

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

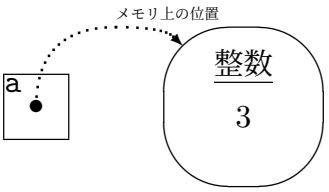
また, Sage では起動時にいくつかの数学定数などを自動的に変数に代入しているので, それらを上書きしない方が良いです. (詳しくは A 章を参照)

注意 1.2.1. 基本的には予約後と重複しなければどの様な名前をつけてもエラーは出ないのですが, 名前の付け方の標準的な作法に従っていると, プログラムを読み理解するのが容易になることが多いです. 名前の付け方の作法については第 2.14 節を参照して下さい.

a=3 と書くと a という変数に 3 という値が代入されるという説明をしましたが, この時どのようなことが起こっているのかもう少し詳しく説明をしたいと思います.

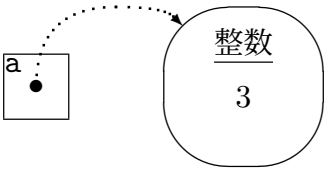
a=3 と書くと, まず 3 という式が評価され, 整数の 3 を表すものが作られメモリのどこかに確保されます (図 1.1(a)). このようにメモリのどこかに確保された具体的に何かを表しているものをオブジェクトと呼びます. 整数の 3 を表すオブジェクトは作られメモリのどこかに確保されたあと, そのオブジェクトがどこにあるのかという情報が a という変数

(a) まずオブジェクトが作られる

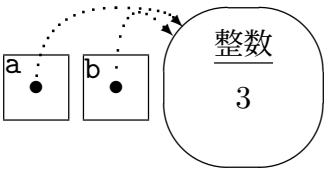


(b) 場所が変数に記録される

図 1.1　a=3 のときの挙動



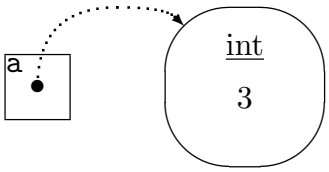
(a) a=3



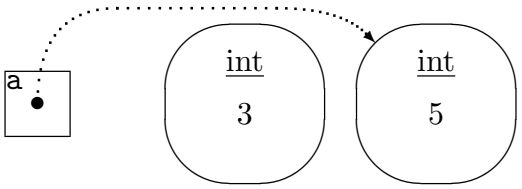
(b) b=a

図 1.2　a=3; b=a のときの挙動

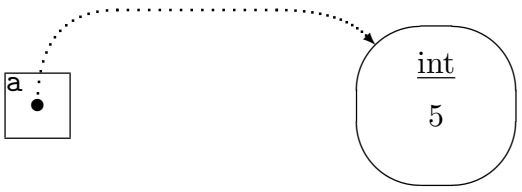
に記録されます (図 1.1(b)). a=3 と書くと a という変数に 3 という値が代入されるという説明をしましたが, これは実はあまり正確ではなく, より正確には, a という変数には整数の 3 というオブジェクトがメモリ上のどこにあるかという情報が記録されており, a という変数には整数の 3 そのものが記録されているわけではありません. 例えば, print a と書いた場合, まず a という変数に書いてあるメモリ上の位置を確認し, その後実際にその位置に書いてあるオブジェクトを読み取り, その内容を表示するということが行われます.



(a) `a=3`



(b) `a=5`



(c) 'そのうち' 3 は消される

図 1.3 `a=3; a=5` のときの挙動

さらに `b=a` と書いた時に何が起こるかについて, 考えてみましょう. まず `a=3` と書いた段階で 3 を表すオブジェクトが作られ, その場所が `a` には記録されています. 図 1.2(a) の様な状況です. この時に, `b=a` と書くと `a` に記録されている場所が, `b` に記録され図 1.2(b) の様な状況になります. 場所の情報が書き加えられるだけで, 新たに 3 を表すオブジェクトが作られたりはしません.

今度は, `a=3` としたあと, `a=5` とすることを考えましょう. まず最初に, `3` という式が評価され, 整数の 3 を表すオブジェクトが作られメモリのどこかに確保され, それがどこにあるのかという情報が `a` という変数に記録されます (図 1.3(a)). `a=5` とすると, `5` という式が評価され, 整数の 5 を表すオブジェクトが作られメモリのどこかに確保され, それがどこにあるのかという情報が `a` という変数に記録されます (図 1.3(b)). 最初に作られた整数の 3 を表すオブジェクトが上書きされるわけではありませんので, `a=5` とした後も, 整数の 3 を表すオブジェクトはメモリ上にあります. 整数の 3 を表すオブジェクトはメモリ上にありますが, どの変数からも参照されていません. このようにもう使われることのなくなったオブジェクトは, 自動的に適当なタイミングでメモリから消去されます (図

1.3(c)). このような仕組みを**ガベージコレクション**と呼びます. 自動で削除されますので, ユーザがオブジェクトをメモリから消去する事を考えることは必要ありません.

1.3　メソッドの定義

プログラムを作成する時には, 同じような処理を何度も行うことがあります. その処理を行う '命令' を定義して使いまわすと, プログラムがすっきりします. また, 何か修正をする時も '命令' を定義している部分を修正するだけで済むため, 負担が減ります. 定義する '命令' のことを '関数' と呼ぶこともあります. 本稿でも関数と呼んでも良いのですが, Sage では symbolic に定義された (数学的な意味での) 函数を扱うこともあり, その仕組みは Python の '命令' とは異なる[*2]ため, 誤解を防ぐために Python の '命令' のことは関数とは呼ばずにメソッドと呼ぶことにします.

メソッドを定義するには `def` を用います. 例えば, `21` と `34` を出力するメソッドを定義するには次のように書きます.

```
def print_21_34():
  print 21,
  print 34
```

対話的に Python を使っている時にはメソッドを定義した後に空行が必要です.

定義したメソッドを使うには次のように呼び出します.

```
print_21_34()
```

実行結果の例

```
21 34
```

メソッドの名前として使用できる文字列は変数の時と同じです. Python はメソッドの定義部分をインデントで判断しているので注意しましょう. 例えば次のように書くと, `print 55` の 1 文のみがメソッドの定義として判断されてしまい, '`print_55_89()` というメソッドを定義した後 `print 89` を実行する' というプログラムだと認識されます.

```
def print_55_89():
```

[*2] 引数に対して値を返すような Python のメソッドであれば, 数学的な函数と一見大差無いように見えます. 確かに, 値を代入すればそれに対応する値が返ってくるという点では同じですが, 数学的な函数は Python の '命令' よりも多くの情報を持っています. 例えば, 数学的な函数に対しては微分や積分といった数学的な操作をすることが出来ますが, Python のメソッドに対しては出来ません.

```
  print 55
print 89
```

したがって, 次を実行しても 55 しか表示されません.

```
print_55_89()
```

実行結果の例

```
55
```

インデントに使うスペースの数はいくつでも良いですが, 揃っていなければなりません.

注意 1.3.1. 大雑把に言うと, l 行目が: で終わっていた時, l 行目よりも $l+1$ 行目の方が深くインデントされる必要があります. さらに l 行目と同じかもしくはそれよりも浅いインデントが現れるまで $l+1$ 行目と同じかそれより深くインデントされる必要があります. l 行目と同じかそれよりも浅いインデントが現れるまでの範囲をブロックとしてひとかたまりとして扱います. なお, l 行目が:で終わっていないのであれば, l 行目よりも $l+1$ 行目の方を深くインデントすることはできません.

引数をとるメソッドを定義することもできます. 例えば,

```
def print_plus_144(a):
  print a+144
```

と定義しておいて, 次を実行すると, 377 $(= 233 + 144)$ が表示されます.

```
print_plus_144(233)
```

実行結果の例

```
377
```

複数の引数をとるメソッドも定義できます. 例えば,

```
def print_sum(a,b):
  print a+b
```

と定義しておいて, 次を実行すると 1597 $(= 610 + 987)$ が表示されます.

```
print_sum(610,987)
```

```

```

実行結果の例

```
1597
```

メソッドから値を返すには return を使います. 例えば

```
def plus_1597(a):
  return a+1597
```

と定義しておいて, 次を実行すると plus_1597(2584) を実行した結果として, 4181 $(= 1597 + 2584)$ が a に代入されていることが分かります.

```
a=plus_1597(2584)
print a
```

実行結果の例

```
4181
```

return が実行されると, その時点でそのメソッドから抜けます. 例えば

```
def print_some():
  print 6765,
  print 10946,
  return 17711
  print 28657,
  return 46368
```

と定義して, 次を実行してみましょう.

```
a=print_some()
```

実行結果の例

```
6765 10946
```

6765 と 10946 は表示された後 return 17711 が実行されます. その時点でメソッドの実行を打ち切るため, 28657 は表示されません. また次を実行することで, a に 17711 が代入されていることを確かめられます.

```
print a
```

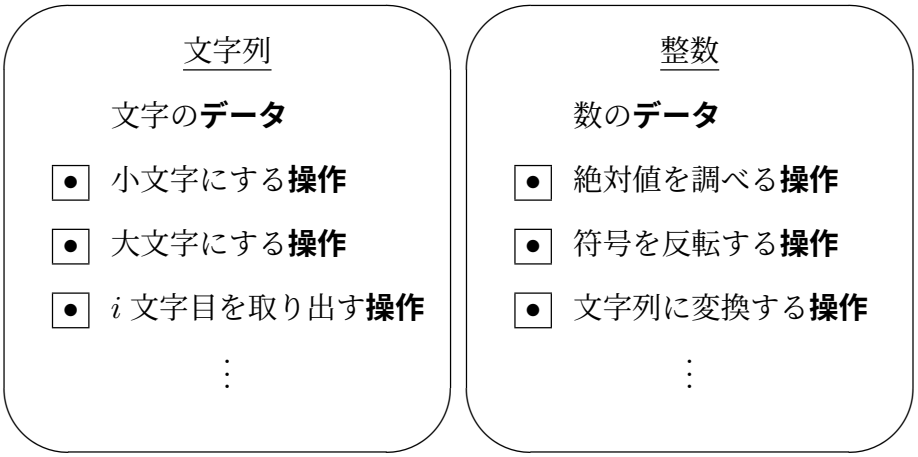


図 1.4　オブジェクトは要素の情報以外にも知っていることがある

実行結果の例

```
17711
```

return の後ろに何も書かなかった場合は None が返されます.

メソッドを定義する方法については, 以上の通りです. 以下では, オブジェクトと関連した少し細かい話をします. メソッドは, 大雑把に言うと, データに対する操作です. データに対する操作と操作されるデータとは密接に関連しており対にして考えるべきです. 例えば, a が偶数か奇数かということを調べる操作は, a が整数であれば意味を持ちますが, a が例えば小数であったり文字列であったりした場合には意味を持ちません. 偶数か奇数かということを調べる操作は整数に固有の操作です. Python では, あるオブジェクトに固有の操作を, そのオブジェクトと関連させて定義する仕組みがあります. 例えば, a="abc" と書くと, まず"abc"という式が評価され, 文字列 z を表すオブジェクトが作られメモリのどこかに確保されたあと, そのオブジェクトがどこにあるのかという情報が a という変数に記録されます. メモリに確保されているオブジェクトには, 自分自身が何者であるか (文字列であるという情報), そのオブジェクトに固有のデータ (abc という情報), そのオブジェクトが持っているメソッド, などが格納されています. 文字列のオブジェクトであれば, 例えば, 'その文字列に含まれるアルファベットを全て大文字に変換した文字列を返すメソッド' が upper という名前で格納されています. a が指すオブジェクトが持っている upper() というメソッドを使うには, a.upper() というように書きます.

```
a='abc'
a.upper()
```

実行結果の例

```
ABC
```

dir(a) を実行することで変数 a に代入されているオブジェクトの持つメソッドを調べることが出来ます.

1.4　Python で扱える基本的なデータ形式

Python で扱えるデータ形式について誤解を恐れず概説します.

1.4.1　数値型

Python では, 整数 (int), 小数 (浮動小数点型, float) などが使えます. これらは四則演算などができます.

```
a=7
b=3
print a + b
print a - b
print a * b
print a / b
print a // b
print a % b
print a ** b
```

整数の割り算/は注意が必要です. (Python 2 では丸められた整数が返って来ます. Python 3 では, 小数が返って来ます. Sage では有理数が返って来ます.) //は商 (を切り下げたもの) が返って来ます. Pure Python では**は冪を表します. (A.4 節も参照のこと.)

```
a=7.0
b=3
print a + b
print a - b
print a * b
print a / b
print a // b
print a % b
print a ** b
```

小数と整数の割り算では/は商が返って来ます. //は商を切り下げたものが返って来ます.

1.4.2 文字列

Python では文字列は, "と"で挟むか, ’ と’ で挟むことで表します.

```
a=’string’
print a
```

実行結果の例

```
string
```

```
b="string"
print b
```

実行結果の例

```
string
```

この時, 途中で改行するとエラーとなります. 改行を含む文字列を表したい時には\n を入れます. もし\n を使わずに, 改行を含む文字列を表したい時には, Python では文字列は, """と"""で挟むか, ’’’ と’’’ で挟むことで表します.

```
a=’AAA\nBBB\nCCC’
print a
```

実行結果の例

```
AAA
BBB
CCC
```

```
b=’’’AAA
BBB
CCC’’’
print b
```

実行結果の例

```
AAA
BBB
CCC
```

```
c="AAA\nBBB\nCCC"
print c
```

実行結果の例

```
AAA
BBB
CCC
```

```
d="""AAA
BBB
CCC"""
print d
```

実行結果の例

```
AAA
BBB
CCC
```

\の直後の改行は無視されますので, それを使うと見やすくなることもあります.

```
a='''\
AAA
BBB
CCC\
'''
print a
```

実行結果の例

```
AAA
BBB
CCC
```

```
b="""\
AAA
BBB
CCC\
"""
print b
```

実行結果の例

```
AAA
BBB
CCC
```

\を使い, 改行文字や, \, ", ' などを表すことが出来ます. 基本的に C 言語と同じですが, 主なものを以下に挙げておきます.

- \改行 ⇝ 無視
- \\ ⇝ \
- \" ⇝ "
- \' ⇝ '

- \n ⇝ 行送り
- \t ⇝ タブ

文字列同士を足す (+) とつなげた文字列が得られます.

```
a='abc'
b='ABC'
c=a+b

print c
```

実行結果の例

```
abcABC
```

数値型を文字列に変換するには次のようにします.

```
a=196418
b=str(a)

print b
```

実行結果の例

```
196418
```

逆に文字列を数値に変換するには次のようにします.

```
a='317811'
b=int(a)
c=float(a)

print a,
print b,
print c
```

実行結果の例

```
317811 317811 317811.0
```

1.4.3 論理型

True と False があります. これらには, and or not の操作をすることができます.

```
a=True
b=False
```

```
print a and b,
print a or b,
print not b
```

実行結果の例

```
False True True
```

1.4.4 コレクション

データをいくつか集めたものを**コレクション**といいます. Python ではコレクションの種類がいくつかあります.

リスト (list)

Python での一番基本となるコレクションはリストというものです. [と] で挟んだ中に, ‘,’ で区切り, 列挙をして表すことができます. 例えば, 次のように書きます.

```
a = [ 400 , 301 , 202 , 103 ]
print a
```

実行結果の例

```
[400, 301, 202, 103]
```

各要素には次のようにアクセスすることができ, 他の言語での配列などと同様に扱うことができます.

```
print a[0],
print a[1],
print a[2],
print a[3]
```

実行結果の例

```
400 301 202 103
```

添字は 0 から始まります. 負の整数を引数に与えると, 後ろから数えたものにアクセスすることができます.

```
print a[-4],
print a[-3],
print a[-2],
print a[-1]
```

実行結果の例

```
400 301 202 103
```

要素の総数よりも大きい値を与えるとエラーとなります.

```
a[5]
```

実行結果の例

```
IndexError: list index out of range
```

要素の総数は, 次のように len を使うことで調べることができます.

```
b=len(a)

print b
```

実行結果の例

```
4
```

各要素に代入し内容を部分的に変更することができます.

```
a = [ 400 , 301 , 202 , 103 ]
print a
a[1]=3524578
print a
```

実行結果の例

```
[400, 301, 202, 103]
[400, 3524578, 202, 103]
```

この操作では a に代入されているリストが書き変わっていることに気を付けてください.

最後に要素を付け加えることもできます.

```
a = [ 400 , 301 , 202 , 103 ]
print a
a.append(5702887)
print a
```

実行結果の例

```
[400, 301, 202, 103]
[400, 301, 202, 103, 5702887]
```

この操作では a に代入されているリストが書き変わっていることに気を付けてください.

また, もとあった順と逆順にしたり, 小さい順に並び替えたりなどということもできます.

```
a = [ 80400 , 10301 , 30202 , 10103 ]
a.reverse()
print a
a.sort()
print a
```

実行結果の例

```
[10103, 30202, 10301, 80400]
[10103, 10301, 30202, 80400]
```

この操作では a に代入されているリストが書き変わっていることに気を付けてください.

リスト同士の足し算 (+) は, 2 つをつなぎあわせた新しいリストになります.

```
a=[300,201,102]
b=[40,31,23,14]
c=a+b
print c
print a
print b
```

実行結果の例

```
[300, 201, 102, 40, 31, 23, 14]
[300, 201, 102]
[40, 31, 23, 14]
```

この操作では a に代入されているリストは書き変わっていません.

スライスといって, リストの一部分を切り出してくることもできます. リスト a に対し, a[n:m] と書くと, [a[n],a[$n+1$],...,a[$m-1$]] というリストが返って来ます.

```
a=[500, 401, 302, 203, 104]
b=a[1:4]
print b
print a
```

実行結果の例

```
[401, 302, 203]
[500, 401, 302, 203, 104]
```

この操作では a に代入されているリストは書き変わっていません. また, 最初のインデックスを省略したり (0 を指定したことになる), 最後のインデックスを省略したり (-1 を指定したことになる), インデックスを負の整数を使って指定することもできます.

```
a=[500, 401, 302, 203, 104]
print a[1:-1]
print a[1:]
print a[0:4]
print a[:4]
```

実行結果の例

```
[401, 302, 203]
[401, 302, 203, 104]
[500, 401, 302, 203]
[500, 401, 302, 203]
```

a がリスト, n が整数ならば,

- a[:n] は最初の n 個の要素からなる新しいリスト,
- a[-n:] は最後の n 個の要素からなる新しいリスト,
- a[n:] は最初の n 個の要素を取り除いた新しいリスト,
- a[:-n] は最後の n 個の要素を取り除いた新しいリスト

となります.

注意 1.4.1. a=[0,2] を実行した際には, まず [0,2] が評価されオブジェクトが作られます. この場合, 'リストに整数 0, 2 が記録されている' と思うよりは, '整数 0, 2 を表すオブジェクトがそれぞれ作られ, そのオブジェクトがどこにいるかがリストには記録されている' と思った方が実際に起こっている事に近いです (図 1.5(a)). 例えば, a=[0,2] の後に a=[1,3] を実行した場合は, 新しいリストが作られた後, a に代入されます (図 1.5(b)). この場合, 最初に作られたリスト自身は書き換えられていません. 一方, a=[0,2] の後に a[0]=1 を実行した場合は, 1 を表すオブジェクトが作られ, 今までのリストが書き換えられることになります (図 1.6(b)). さらに a[1]=2 を実行した場合は, 2 を表すオブジェクトが作られ, リストが書き換えられることになります (図 1.6(c)). 変更が加わる所が異なるので注意が必要です. リストの場合は一度オブジェクトが作られた後も, オブジェクトの内容を書き換えることができます. この様なオブジェクトは**変更可能な** (**可変性**/*mutable*) オブジェクトと呼ばれます.

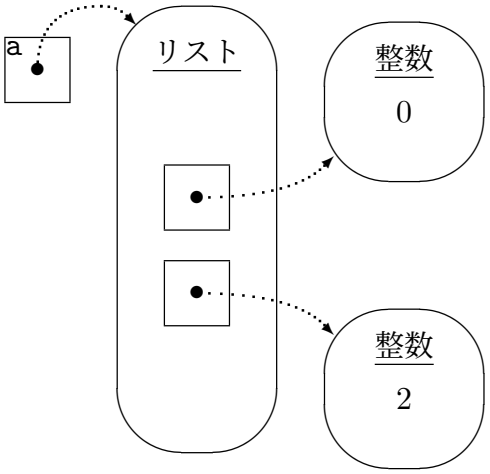
一度オブジェクトが作られた後はオブジェクトの内容を書き換えることができない様なものもあります (例えば次で紹介するタプルなど). この様なオブジェクトは**変更不可能な** (**不変性**/*immutable*) オブジェクトと呼ばれます.

タプル (tuple)

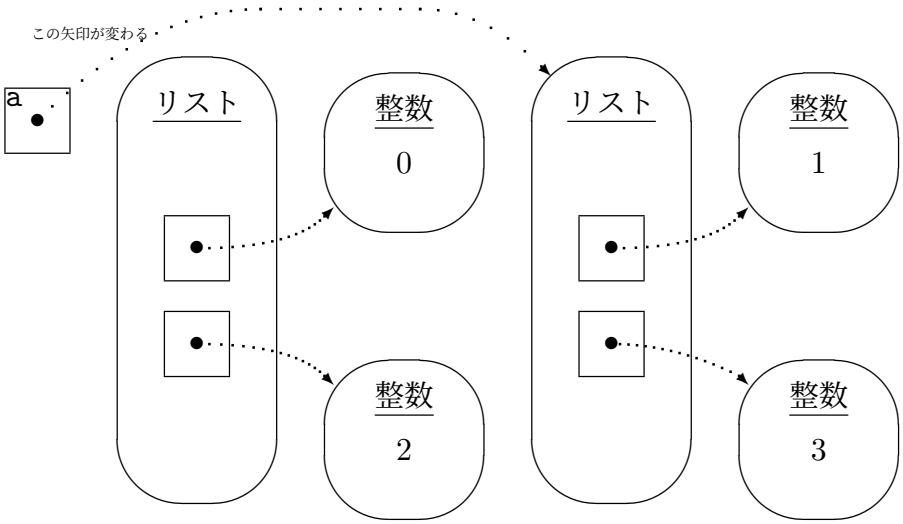
リストによく似たタプルというものがあります.

(と) で挟んだ中に, ',' で区切り, 列挙をして表すことができます. 例えば次のようにします.

```
a=(300, 201, 102)
print a
```



(a) a=[0,2]



(b) a=[1,3]

図 1.5　a=[0,2]; a=[1,3] のときの挙動

実行結果の例

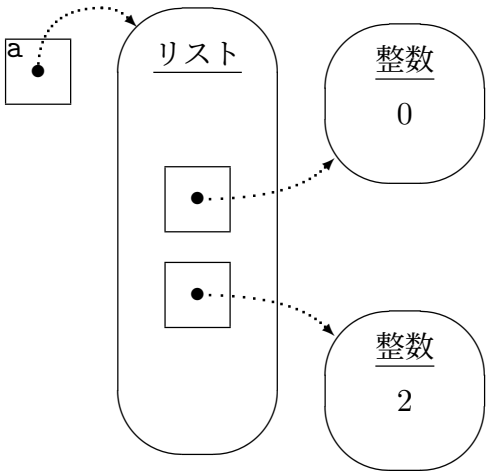
```
(300, 201, 102)
```

タプルでもリストの時のように各要素にアクセスすることができます.

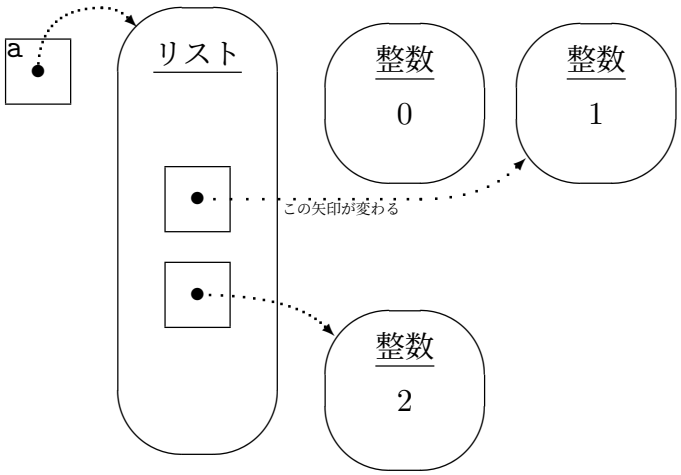
```
a=(300, 201, 102)
print a[1]
```

実行結果の例

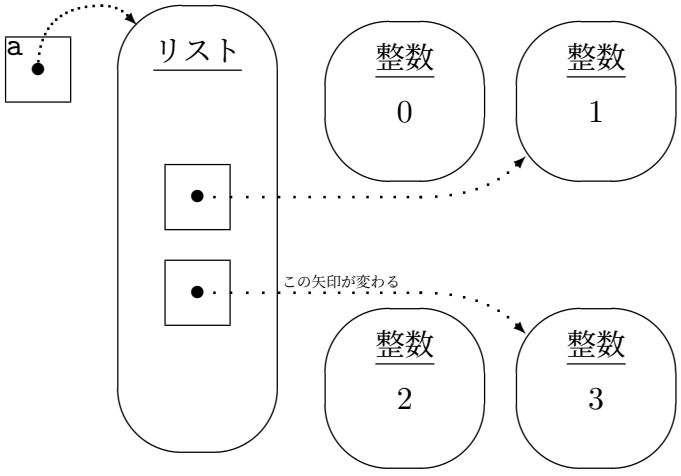
```
201
```



(a) `a=[0,2]`



(b) `a[0]=1`



(c) `a[1]=3`

図 1.6 `a=[0,2]; a[0]=1; a[1]=3` のときの挙動

ただし, リストとは異なり, 各要素に代入して内容を部分的に変更することはできません.

```
a=(300, 201, 102)
a[1]=3524578
```

実行結果の例

```
TypeError: 'tuple' object does not support item assignment
```

また, append のようにタプル自身の内容を書き換えるような操作もできません. リストのように書き換える操作ができるコレクションを変更可能な (mutable) コレクションといい, タプルのように書き換える操作ができないコレクションを変更不可能な (immutable) コレクションといいます.

タプル自身の内容を書き換える操作はできませんが, そうではない操作はできます.

```
a=(1,2,3,4,5)
b=(6,7,8)
c=a+b
print c
print a[:2]
a=(8,9)
print a
print b
print c
```

実行結果の例

```
(1, 2, 3, 4, 5, 6, 7, 8)
(1, 2)
(8, 9)
(6, 7, 8)
(1, 2, 3, 4, 5, 6, 7, 8)
```

タプルでは, 次のように各要素の値をそれぞれ別の変数に一気に代入することができます.

```
a=(1,2,3,4,5)
(a1,a2,a3,a4,a5)=a

print a1,
print a2,
print a3,
print a4,
print a5
```

実行結果の例

```
1 2 3 4 5
```

ただし, 変数の個数が長さに合っていない時はエラーとなります.

タプルからリストにを作ったり, 逆にリストからタプルを作ったりもできます.

```
a=(1,2,3)
b=list(a)
print a,
print b
```

実行結果の例

```
(1, 2, 3) [1, 2, 3]
```

```
a=[1,2,3]
b=tuple(a)
print a,
print b
```

実行結果の例

```
[1, 2, 3] (1, 2, 3)
```

辞書 (dictionary)

リストやタプルはインデックスとして整数をとりました. 辞書はインデックスとして整数以外のものまで使うことができるものです. 他のプログラミング言語では, ハッシュ, 連想配列などと呼ばれることもあります. 辞書ではインデックスのことを**キー**と呼びます. {と}で挟んだ中に, **キー**:**値** というものをカンマ区切りで列挙します.

```
a={'x':12, '3':24 , 3:13}
print a['x']
```

実行結果の例

```
12
```

値にないものを指定するとエラーとなります. keys や values を用いることで, キーのリストと値のリストを得られます.

```
a={'x':12, '3':24 , 3:13}
b=a.keys()
c=a.values()
print a
print b
print c
```

実行結果の例

```
{'x': 12, '3': 24, 3: 13}
['x', '3', 3]
[12, 24, 13]
```

辞書のキーとしては, 色々なものが使えます. ただし, immutable でなければいけないという制限があります. したがって, リストをキーとして採用することはできませんが, タプルならキーとして使うことができます.

要素にアクセスし値を代入することができます. 代入の際にもともと存在していなかったキーを指定すると, 自動的に加えてくれます.

```
a={'a':1,'b':2}
print a
a['a']=3
print a
a['c']=4
print a
```

実行結果の例

```
{'a': 1, 'b': 2}
{'a': 3, 'b': 2}
{'a': 3, 'c': 4, 'b': 2}
```

存在していないキーを使ってアクセスしようとすると, (代入でない時には) エラーとなります.

```
a['d']
```

実行結果の例

```
KeyError: 'd'
```

キーと値をペアにしたタプルたちからなるリストを使って, 次のように定義することも出来ます.

```
a=[('a',1), ('b',2)]
b=dict(a)
print b
```

実行結果の例

```
{'a': 1, 'b': 2}
```

集合 (set/frozenset)

数学での集合に相当するものがあります. リストでは, 要素の中に同じものが複数あっても良いですし, どのような順番に並んでいるかという情報も持っていました. set や frozenset では, 順番を気にしません. さらに各要素は unique です. ただし, 要素として使えるのは immutable なもののみです.

```
a=[1,4,3,2,2]
b=set(a)
print b

b.add(5)
print b

b.add(5)
print b

c=frozenset(a)
print c
```

実行結果の例

```
set([1, 2, 3, 4])
set([1, 2, 3, 4, 5])
set([1, 2, 3, 4, 5])
frozenset([1, 2, 3, 4])
```

set は mutable ですが, frozenset は immutable です.

1.4.5 メソッド

メソッドも変数に代入することができます. 例えば

```
def printA():
  print 'A'
```

として A を表示するメソッドを定義します. このメソッドは

```
printA()
```

実行結果の例

```
A
```

のように使います. このメソッドを別の変数に代入すると同じように使用することができます.

```
f=printA
f()
```

実行結果の例

```
A
```

例えば, メソッド f, g を引数にとり, $f(2)$ と $g(2)$ の大きい方の値を返すようなメソッドは次の様に定義します.

```
def max_at_two(f,g):
  a=f(2)
  b=g(2)
  if a>b:
    return a
  else:
    return b
```

このメソッドはおおよそ次のような写像に相当します:

$$\begin{array}{ccc} \{\text{ メソッド }\} \times \{\text{ メソッド }\} & \to & \mathbb{R} \\ (f, g) & \mapsto & \max\{f(1), g(1)\} \end{array}$$

例えば,

```
def square(x):
  return x*x

def half(x):
  return x/2
```

という関数を定義して, 次のように実行します.

```
print max_at_two(square,half)
```

実行結果の例

```
4
```

実は, `def` というのはメソッドを作りそれを変数に代入しているにすぎません.

1.4.6 Type

それぞれの値が一体何者なのかということは type というもので調べることができます.

```
a=[0,1,2]
b=(0,1,2)
c='0,1,2'

print type(a),
print type(b),
print type(c)
```

実行結果の例

```
<type 'list'> <type 'tuple'> <type 'str'>
```

1.5 条件分岐

ここでは, 条件によって実行する処理を分ける方法について説明します.

1.5.1 if/elif/else

条件分岐には if を使います. if に続く条件をチェックし条件が成立していたらそれに続くブロックを実行します. 例えば, 次を実行すると, a には 10 が代入されており 5 よりも大きいので, 10 > 5. と表示されます.

```
a=10
if a > 5:
  print a,
  print '> 5',
  print '.'
```

実行結果の例

```
10 > 5 .
```

条件が成立している時に実行される範囲は, インデントを使って指定します.

このままでは, 1 回しか使えないのでメソッドを定義することにします.

```
def check(a):
  if a > 5:
    print a,
```

```
    print '> 5',
    print '.'
```

と定義すると, check(15) では 15 > 5 . と表示され, check(0) では何も表示されないと思います. 例えば,

```
def check(a):
  if a > 5:
    print a,
    print '> 5',
  print '.'
```

と書くと条件が成立していた時に実行される範囲は, print '> 5.' までとなります. したがって, print '.' は条件によらず実行されるので, check(0) でも . だけは表示されるようになります.

条件が成立していない時に別の動作をしたい時には, else を使います.

```
def check(a):
  print a,
  if a > 5:
    print '> 5.'
  else:
    print '< 5 or = 5.'
```

例えば a が

- 5 より大きい場合;
- それ以外で, 0 より大きい場合;
- それ以外で, 0 と等しい場合;
- それ以外で, −5 以上の場合;
- それ以外

で, それぞれ別の動作をするようにするにはどうしたら良いでしょうか. 例えば

```
def check(a):
  print a,
  if a>5:
    print '>5.'
  else:
```

```
    if a>0:
      print ’ is in (0,5].’
    else:
      if a==0:
        print ’=0.’
      else:
        if a>=-5:
          print ’ is in [-5,0).’
        else:
          print ’<-5.’
```

のように入れ子にすることもできます. しかしながら, if の条件が満たされなかった時にさらに別な条件をチェックするには elif というのが使えるので, そちらを利用した方がシンプルに書けます.

```
def check(a):
  print a,
  if a>5:
    print ’>5.’
  elif a>0:
    print ’ is in (0,5].’
  elif a==0:
    print ’=0.’
  elif a>=-5:
    print ’ is in [-5,0).’
  else:
    print ’<-5.’
```

1.6 ループ

ここでは, 同じ処理を繰り返し行う方法について説明します.

1.6.1 while/else/break/continue

繰り返し同じことを行う方法に while を使ったものがあります. while は条件をチェックし, 条件が成立していたら, それに続くブロックを実行します. ブロックを実行したら, また条件をチェックし, 同じことをします. 例えば, 次を実行すると 1, 2, 3, 4, 5, END が表示されます.

```
a=0
```

```
while a<5:
  a=a+1
  print a,

print 'END'
```

実行結果の例

```
1 2 3 4 5 END
```

while 文でも else を使うことができます. while で条件をチェックし成立しなければ, else に続くブロックを実行します.

```
a=0

while a<5:
  a=a+1
  print a,
else:
  print a,
  print '>4',

print 'END'
```

実行結果の例

```
1 2 3 4 5 5 >4 END
```

while のブロックの中では continue と break というのが使えます. continue は, ブロックの実行をそこで止め, while の条件をチェックしにいき (ループを続け) ます. break はブロックの実行をそこで止め (条件をチェックすることなく) ループから抜けてしまいます. この時 (条件をチェックしないので) else ブロックは実行されません. 例えば, 次を実行してみましょう.

```
a=0

while a<5:
  a=a+1
  print a
  if a % 2 == 0:
    print '.',
    continue
  print 'is odd.'
else:
  print a
  print '>4.'

print 'END'
```

実行結果の例

```
1 is odd. 2 . 3 is odd. 4 . 5 is odd. 5 >4 END
```

a が偶数の時には continue で while の最初に戻り条件のチェックし直しとなるので, a が偶数の時には is odd は表示されません.

```
a=0

while a<5:
  a=a+1
  print a,
  if a % 3 == 0:
    break
  print '.',
else:
  print a,
  print '>4',

print 'END'
```

実行結果の例

```
1 . 2 . 3 END
```

では, a が 3 になった時に, break で while の外に飛ばされるので>4 は表示されません.

1.6.2 for/else/continue/break

繰り返し同じことを行う方法に for を使うものがあります. 例えば次のように書きます.

```
a=[0,1,2,3,4]
for ai in a:
  print ai,
```

実行結果の例

```
0 1 2 3 4
```

このプログラムでは, リスト a の要素一つ一つに対してそれに続くブロックを実行します. 各要素は ai という変数に代入されます. 繰り返しを書くたびにリストを書き下すのは面倒なので, range というリストを作るメソッドがあります. 例えば次のようとすると, 0 以上 4 以下の整数が出力されます.

```
for i in range(5):
  print i,
```

実行結果の例

```
0 1 2 3 4
```

range では開始を 0 以外にすることなどもできます.

```
for i in range(1,5):
  print i,
```

実行結果の例

```
1 2 3 4
```

さらに range では, 刻み幅を 1 以外にすることなどもできます. 刻み幅は第 3 引数として指定します. また刻み幅を指定するときは開始する値を指定しなければなりません.

```
for i in range(1,5,2):
  print i,
```

実行結果の例

```
1 3
```

刻み幅は負の値を指定することも出来ます.

```
for i in range(5,1,-1):
  print i,
```

実行結果の例

```
5 4 3 2
```

for でも else や, break, continue が使えます.

ここでは例としてリストを使いましたが他の様々なコレクションでも for を使うことができます.

第 2 章

知っておくと便利なこと

ここでは, Python でプログラムを書くにあたり知っているとより便利なことについて説明します.

2.1 リストの内包的記法

Python でリストを書く方法に内包的記法というものがあります. 例えば, 偶数を小さい方から 5 つ集めたリストを作ることを考えましょう. ひとつの方法に次のように, 空のリストに次々と付け加えていく方法が考えられます.

```
a=[]
for i in range(5):
  a.append(2*i)

print a
```

実行結果の例

```
[0, 2, 4, 6, 8]
```

同じことを次のように書くこともできます.

```
a = [ 2*i for i in range(5) ]
print a
```

実行結果の例

```
[0, 2, 4, 6, 8]
```

数学で集合を $a = \{2i | i \in \{0, \ldots, 4\}\}$ のように書くことがあるのと似ています. この書き方を**内包的記法**といいます.

さらにこの中で, 3 の倍数のみを集めるにはどうしたら良いでしょうか. 例えば次のように書くこともできます.

```
a=[]
for i in range(5):
  if i % 3 == 0:
    a.append(2*i)
print a
```

実行結果の例

```
[0, 6]
```

次のように書くことも出来ます.

```
a=[ 2*i for i in range(5) if i % 3 == 0 ]
print a
```

実行結果の例

```
[0, 6]
```

$\{\, 2i \mid i \in \{\, 0, \ldots, 4 \,\}, i \equiv 0 \pmod 3 \,\}$ という形に対応しているようにも見えます. また, 次のように書くことも出来ます.

```
a = [ 2*i for i in range(5) ]
b = [ ai for ai in a if ai % 3 == 0]
print b
```

実行結果の例

```
[0, 6]
```

入れ子になったループも同じように書くことができます. 例えば

$$\{\, (i,j) \mid i \in \{\, 0,1,2 \,\}, j \in \{\, 0,1 \,\} \,\}$$

に相当するリストは次のように作ることができます.

```
a=[]
for i in range(3):
  for j in range(2):
    a.append((i,j))
print a
```

実行結果の例

```
[(0, 0), (0, 1), (1, 0), (1, 1), (2, 0), (2, 1)]
```

これを内包的記法で書くと次のようになります.

```
a=[ (i,j) for i in range(3) for j in range(2)]
print a
```

実行結果の例

```
[(0, 0), (0, 1), (1, 0), (1, 1), (2, 0), (2, 1)]
```

これは次とは少し異なります.

```
a=[ [ (i,j)  for j in range(2) ] for i in range(3) ]
print a
```

実行結果の例

```
[[(0, 0), (0, 1)], [(1, 0), (1, 1)], [(2, 0), (2, 1)]]
```

2.2 リストの内包的記法とスライス

スライスといってリスト一部分を取り出す方法がある事は既に述べたとおりです. 整数 n, m とリスト a に対して, a[n:m] と [a[i] for i in range(n,m)] は等しくなります.

```
a = [500,401,302,203,104]
b = a[1:4]
c = [ a[i] for i in range(1,4)]

print b
print c
```

実行結果の例

```
[401, 302, 203]
[401, 302, 203]
```

また range の時と同様に刻み幅を指定することもできます. 整数 n, m, k とリスト a に対して, a[n:m:k] と [a[i] for i in range(n,m,k)] は等しくなります.

```
a = [500,401,302,203,104]
b = a[0:4:2]
c = [ a[i] for i in range(0,4,2)]

print b
print c
```

実行結果の例

```
[500, 302]
[500, 302]
```

2.3　yield

yield というものを紹介します. とりあえず次のメソッドを定義しましょう.

```
def xodd(n):
  i=1
  while(i<n):
    yield i
    i=i+2
```

このメソッドは次のように使うことができます.

```
for i in xodd(20):
  print i,
```

実行結果の例

```
1 3 5 7 9 11 13 15 17 19
```

20 より小さい奇数が表示されることと思います. def の中で return ではなく yield を使うと for 文に使うことができます. メソッドの中に yield があった場合一旦処理を中断し, 値を返します. 再び呼ばれた時には, 処理を中断したところから, 処理を再開します.

ただ単に同じ処理をしたいだけなら, 次のようにも書けます.

```
odd=[ 2*i+1 for i in  range(10)]
for i in odd:
  print i,
```

実行結果の例

```
1 3 5 7 9 11 13 15 17 19
```

しかしながら, このように書くと最初の行で, odd に $1, 3, \ldots, 19$ からなるリストが代入されます. この時点でメモリ内に 10 個の整数が確保されてしまいます. 今は 10 個しかないので大丈夫ですが, 数が大量になると, メモリが足りなくなる可能性があります. 一方 yield を使った方は, 確保しているのは基本的に i だけですので, メモリが節約できてい

ます.

また yield を使った方法では, 何に関する繰り返しなのかが明確になり, プログラムが分り易くなる可能性があります. 例えば, 次のように書いても同じではありますが, 20 より小さい奇数について print を実行しているということは, 最初のプログラムの方が理解しやすいと思います.

```
for i in range(10):
  print  2*i+1,
```

実行結果の例

```
1 3 5 7 9 11 13 15 17 19
```

再帰的に用いることで列挙が簡単にできる場合もあります. 例えば

$$\left\{ (a_1, \ldots, a_3) \in \mathbb{N}^3 \;\middle|\; \sum_i a_i = 2 \right\}$$

という集合に興味があるとします.

$$S_{n,m} = \left\{ (a_1, \ldots, a_n) \in \mathbb{N}^n \;\middle|\; \sum_i a_i = m \right\}$$

とおくと,

$$\begin{aligned} S_{n,m} &= \left\{ (a_1, \ldots, a_n) \in \mathbb{N}^n \;\middle|\; \begin{array}{c} a_n \le m, \\ (a_1, \ldots, a_{n-1}) \in S_{n-1,m-a_n} \end{array} \right\} \\ S_{1,m} &= \{\, (m) \,\} \end{aligned}$$

と書けますので, このことを使って, 次のように書くことができます.

```
def xs(n,m):
  if n==1:
    yield [m]
    return
  for an in range(m+1):
    for a in xs(n-1,m-an):
      yield  a+[an]
```

これを次のように実行すると欲しかった対象を得ることができます.

```
for a in xs(3,2):
  print a,
```

実行結果の例

```
[2, 0, 0] [1, 1, 0] [0, 2, 0] [1, 0, 1] [0, 1, 1] [0, 0, 2]
```

yield を使った場合にも, メソッドから抜けて処理を終了したい時には return を用います. ただし yield を使った場合には, return で値を返すことはできません. ですので, その直前で yield を使い値を返しています.

例えば, 次の例は ‘きちんと対応が付いているカッコ’ を出力するためのメソッドです.

```
def xparentheses(n):
  if n==0:
    yield ''
    return
  for i in range(n):
    for p1 in xParentheses(i):
      for p2 in xParentheses(n-1-i):
        yield '('+p1+')'+p2
```

これを使い次を実行すると ‘きちんと対応が付いている 3 組みのカッコ’ をすべて列挙します.

```
for p in xparentheses(3):
  print p,
```

実行結果の例

```
()()() ()(()) (())() (()()) ((()))
```

2.4　range v.s. xrange

例えば次のプログラムのように, range(100) とすると, Python 2 の場合は 0 から 99 までの整数をメモリ上に確保して長さ 100 のリストを作ってから for 文を始めます.

```
n=0
for i in range(100):
  n=n+i

print n
```

実行結果の例

```
4950
```

この場合, それぞれの整数は足し算の時にだけあればよく, 最初にすべて確保する必要はありません. 例えば, 次のように yield を使ってメソッドを定義し

```
def r(m):
  i=0
  while i<m:
    yield i
    i=i+1
```

次のように実行することもできます.

```
n=0
for i in r(100):
  n=n+i

print n
```

実行結果の例

```
4950
```

この場合, 最初にすべての整数をメモリに確保するわけではないので, 無駄がないように見えます. 実はこのように yield を使って書いたものと同等のものが既にあり, xrange という名前で使えます.

```
n=0
for i in xrange(100):
  n=n+i

print n
```

実行結果の例

```
4950
```

実は, xrange の方が便利だということで, Python 3 では大胆な変更が加えられました. Python 2 までの range に相当するものは廃止になり, Python 2 までの xrange に相当するものを Python 3 では range と呼ぶことになりました. また, Python 3 では xrange と呼ばれるものはありません.

注意 2.4.1. Python 2 と Python 3 とは, 基本的には同じではあるものの, 一部で互換性

のない変更が加えられました. (例えば, 整数同士の割り算 1/2 は Python 2 では 0 と丸めて整数にしたものが返って来ますが, Python 3 では 1.5 という小数になります. 丸めた値が必要なときは//を使うほうが良いです.) ですので, マニュアルを読む時は気をつけた方が良いこともあります. (ほとんどの場合は Python 2 と Python 3 とで互換があるので過度に気にする必要はないかもしれません.) 現状では Sage は Python 2 がベースになっていますが, 今後変更があるかもしれませんので, その時は注意が必要です.

2.5　— if — else —

例えば, a_i の値が, i が 3 の倍数なら 1, そうでないなら 0 となるようなリストを定義することを考えます. 内包的記法を使うのであれば, まず, i が 3 の倍数なら 1, そうでないなら 0 となるようなメソッドを定義します.

```
def zero_one(i):
  if i % 3 == 0:
    return 1
  else:
    return 0
```

そしてこのメソッドを用いて次のように書くことができます.

```
a=[ zero_one(i) for i in range(10)]
print a
```

実行結果の例

```
[1, 0, 0, 1, 0, 0, 1, 0, 0, 1]
```

これを別の書き方で書いてみます. `A if COND else B` という式は, `COND` が真なら `A` そうでなければ `B` となります. 例えば, 次は 1 となり

```
i=3
a=( 1 if i%3==0 else 0 )
print a
```

実行結果の例

```
1
```

次は 0 となります.

```
i=4
a=( 1 if i%3==0 else 0 )
print a
```

実行結果の例

```
0
```

見やすさのためにカッコをつけましたがなくても大丈夫です. この書き方を利用すると, 先程のものは次のように書けます.

```
a=[ 1 if i%3==0 else 0 for i in range(10)]
print a
```

実行結果の例

```
[1, 0, 0, 1, 0, 0, 1, 0, 0, 1]
```

短く書けますが, 内包的記法が複雑になると読みにくくなるので注意が必要です.

2.6 if — : —

次の例を考えましょう.

```
for i in range(10):
  if i % 3 == 0:
    continue
  print i,
```

実行結果の例

```
1 2 4 5 7 8
```

これは 3 の倍数の時は処理をスキップしそれ以外の時に表示するということをしています. この場合 if 文で実行されるのは continue の 1 行しかありません. このように if 文で実行すべきものが 1 行しかない場合には次のように:の後ろに処理を書くことができます.

```
for i in range(10):
  if i % 3 == 0: continue
  print i,
```

実行結果の例

```
1 2 4 5 7 8
```

このような書き方をしたほうが読みやすくなることもあります.

注意 2.6.1. この節扱った例は次の様に書くこともできます:

```
for i in range(10):
  if i % 3 == 0:
    continue
  else:
    print i,
```

実行結果の例

```
1 2 4 5 7 8
```

この場合も同様に次の様に書くことも出来なくもありませんが, 非常に読みにくいです:

```
for i in range(10):
  if i % 3 == 0: continue
  else:
    print i,
```

実行結果の例

```
1 2 4 5 7 8
```

プログラムにおいて, 読みにくい箇所があると, 誤解や勘違いの元となり, バグの潜在的なリスクになるので, この様な読みにくい書き方は避けるべきです. :の後ろに処理を書くのは, else がない時のみにすべきです.

2.7 pass

pass というものがあります. pass は何もしません. 例えば次のように書くと 1, 2, 3, 4 が表示されます.

```
def f():
  print 1,
  print 2,
  pass
  print 3,
  print 4
```

```
f()
```

実行結果の例

```
1 2 3 4
```

pass は何もしないわけですが, 例えば文法上では文が必要なのだけど何も実行したくないという時に使ったりします. 例えばの次のような形で使うことがあります.

```
def f(n):
  if n>0:
    pass
  else:
    print 'n<=0'
```

この場合は, if 文の条件を変更すれば pass を使わなくても書けます. しかし, 例えば, 後でプログラムを書く予定で試しに実行したい時などには便利かもしれません.

2.8 比較演算子

数値に対して使える比較演算子には次のものがあります.

```
== <> != < <= > >=
```

意味は大体想像がつくと思いますので省略します.

コレクションに対して使える比較演算子には次のものがあります.

```
== != in
```

同じコレクションでは, 任意の添字 i に対して a[i]==b[i] となる時に a==b は True となります. 次の例を見てみましょう.

```
a=[1,2,3]
b=[1,2,3]
c=a
d=a[:]

print a,
print b,
print c,
print d
```

実行結果の例

```
[1, 2, 3] [1, 2, 3] [1, 2, 3] [1, 2, 3]
```

この時, 次はすべて True になります.

```
print a==b,
print b==c,
print c==d
```

実行結果の例

```
True True True
```

このことは一見当たり前のように見えますが, 次のことに注意しましょう. 例えば, 次を実行すると a と c のみ [1,2,8] となるはずです.

```
a[2]=8
print a
print b
print c
print d
```

実行結果の例

```
[1, 2, 8]
[1, 2, 3]
[1, 2, 8]
[1, 2, 3]
```

a と c は本当に同じリストを指しています. ですので, a が指しているリストの内容を変更するということは, それは同時に c が指しているリストも変更するということになります. b は a と同じ内容の別のリストを新たに作っただけなので, 全く同じものを指しているわけではありません. また, d は a のコピーを指しているので, a と同じリストを指しているわけではありません. したがって, a が指しているリストを変更しても b や d が指しているリストは影響を受けません.

つまり, ==は**内容**が同じであれば True となります.

ちなみに, a が指しているリストの内容を変更するというのは, a に別なリストを代入するということとは違います.

```
a=[1,2,3]
b=a

a[0]=10
print a
print b

a=[4,5,6]
```

```
print a
print b
```

実行結果の例

```
[10, 2, 3]
[10, 2, 3]
[4, 5, 6]
[10, 2, 3]
```

次に in の話をします. 要素として含んでいるかを調べるのに使います.

```
a = [1,4,9,16]
print (2 in a),
print (4 in a)
```

実行結果の例

```
False True
```

例えば, 3, 6, 9, 12, 13 以外の 15 未満の非負整数を表示するには次のように書くことができます.

```
b=[3,6,9,12,13]
for i in range(15):
  if not i in b:
    print i,
```

実行結果の例

```
0 1 2 4 5 7 8 10 11 14
```

実は not in というのもあって, 次のようにも書けます.

```
b=[3,6,9,12,13]
for i in range(15):
  if i not in b:
    print i,
```

実行結果の例

```
0 1 2 4 5 7 8 10 11 14
```

内包的記法でも使えます.

```
b=[3,6,9,12,13,15,18]
a=[ i for i in range(20) if not i in b ]
print a
```

実行結果の例

```
[0, 1, 2, 4, 5, 7, 8, 10, 11, 14, 16, 17, 19]
```

辞書の時には, キーにあれば True となります.

```
a={1:10,2:20}
print 1 in a ,
print 10 in a
```

実行結果の例

```
True False
```

2.9　setdefault

辞書では, 存在しないキーでアクセスしようとするとエラーとなります.

```
a={1:10,2:20}
print a[3]
```

実行結果の例

```
KeyError: 3
```

ですので, アクセスする時には in を使って調べておくのが安全です. しかし setdefault という便利なものもあります. setdefault(KEY,VALUE) とすると, もし既にキー KEY が存在していればそのキーに対応する値を返します. もし KEY がキーとして存在しない時には, VALUE をそのキーに対応する値と設定した上で VALUE を返すというものです.

```
a={1:10,2:20}
print a.setdefault(1,-1)
print a.setdefault(2,-1)
print a.setdefault(3,-1)
print a
print a[1]
print a[2]
print a[3]
```

実行結果の例

```
10
20
-1
1: 10, 2: 20, 3: -1
10
20
-1
```

2.10 文字列に対する %

数値に対する % は割り算のあまりを求める演算でしたが, Python 2 では文字列に対して % を使うとフォーマット処理ができます. 次を実行すると b is 10. が表示されると思います.

```
a = 'b is %d.'
b = 10
c = a % b
print c
```

実行結果の例

```
b is 10.
```

'b is %d.' の %d の部分が b の値で置き換わっています. **文字列 % 値**とすると, 文字列の中にある %d のような書式を指定する部分を値で置き換えます. 書式の指定法は次のようなものがあります.

```
'%d' % 11
'%f' % 10.5
'%4d' % 11
'%04d' % 11
'%4f' % 10.5
```

複数の値で置き換えたい時には次のようにタプルを用います.

```
a = 10
b = 12
c = 'a is %d and b is %d'
d = c % (a,b)
print d
```

実行結果の例

```
a is 10 and b is 12
```

他にも辞書を用いて指定する方法などいくつかの方法が用意されています.

2.11　zip, enumerate

for 文を使う際に便利ないくつかのことを説明します.

zip

a というリストと b というリストに対し, 添字が等しいもの同士をペアにして処理をしたい時があります. 例えば, 添字が等しいもの同士の差 $a_i - b_i$ を表示することを考えましょう. 次のように書くことができます.

```
a=[0,1,2,3,4]
b=[10,11,12,13,14]
for i in range(len(a)):
  print a[i]-b[i],
```

実行結果の例

```
-10 -10 -10 -10 -10
```

この処理は zip を使って次のように書くことができます.

```
a=[0,1,2,3,4]
b=[10,11,12,13,14]
for (ai,bi) in zip(a,b):
  print ai-bi,
```

実行結果の例

```
-10 -10 -10 -10 -10
```

個数が揃っていない場合は, どちらかが尽きたら終わります.

この処理には添字 i は直接関係ないですから, i を使っていない分, この方がすっきりしていると言えます. また, zip を使う方法は yield を使って定義したメソッドに対しても使えますので, 汎用的です. 例えば, 次のメソッドは, Fibonacci 型の漸化式

$$a_{i+1} = a_i + a_{i-1}$$

で定義される数列を, 初期値 a_0, a_1 で計算し, m を超えるまで出力します.

```
def fibonacci(m,a0,a1):
  a=a0
  b=a1
  while a < m:
```

```
    yield a
    c=a
    a=b
    b=a+c
```

例えば次のように使います.

```
for ai in fibonacci(10,1,1):
  print ai,
```

実行結果の例

```
1 1 2 3 5 8
```

例えば, 初期値 (a_0, a_1) を $(2, 2)$ とした時と $(1, 1)$ とした時で各項の差がどうなっているか調べたいとしましょう. その時は次のように書けます.

```
for (ai,bi) in zip(fibonacci(100,2,2),fibonacci(100,1,1)):
  print ai-bi,
```

実行結果の例

```
1 1 2 3 5 8 13 21 34
```

enumerate

リスト a に対し, 添字とその添字に対応する値を使って処理をしたい時があります.

例えば, $a_i - i$ を計算することを考えましょう. いくつか方法があります. 例えば,

```
a=[10,8,8,3,2,2,1]
for i in range(len(a)):
  print a[i]-i,
```

実行結果の例

```
10 7 6 0 -2 -3 -5
```

と書くこともできますし,

```
a=[10,8,8,3,2,2,1]
i=0
for ai in a:
```

```
  print ai-i,
  i=i+1
```

実行結果の例

```
10 7 6 0 -2 -3 -5
```

と書くこともできます.

実はこれらの処理は enumerate を使って次のように書くこともできます.

```
a=[10,8,8,3,2,2,1]
for (i,ai) in enumerate(a):
  print ai-i,
```

実行結果の例

```
10 7 6 0 -2 -3 -5
```

2.12　モジュールの話

他の人が書いた便利なメソッドなどを取り込む方法があります. メソッドなどをその用途などによっていくつかまとめたものをモジュールと呼びます. あるモジュールを取り込むには, import を使います. 例えば, 日付や時間を扱うためのメソッドなどが詰まった datetime というモジュールを使うのであれば, 次のようにします.

```
import datetime
```

これを実行した後であれば, datetime モジュールの中で定義されているものたちは, datetime.time() などの様に, モジュール名 (と.) の後あとに続けて書くことで使えます.

また, 自分でモジュールを作ることもできます. 例えば次の様なプログラムを pochhammer.py というファイル名で保存します.

```
def get_raising_factorial(x,n):
  r=1
  for i in range(n):
    r=r*(x+i)
  return r

def get_falling_factorial(x,n):
  r=1
```

```
  for i in range(n):
    r=r*(x-i)
  return r
```

このとき, (同じフォルダにある) 別のプログラムからは, import pochhammer とすることで, これらの関数を取り込むことができます. 例えば次のようなプログラムを書くことができるようになります.

```
import pochhammer

print pochhammer.get_raising_factorial(3,2)
print pochhammer.get_falling_factorial(3,2)
```

import pochhammer が実行されると, 一度 pochhammer.py がひと通り実行され 2 つのメソッドが定義されます. この時, pochhammer.pyc というファイルが生成されます. よく使うメソッドなどはモジュールとして使えるようにしておくと, 再利用しやすくなります.

import pochhammer が実行されると, 一度 pochhammer.py が実行されます. もし仮に次のようなファイルを import すると, 最後の行の print 文も実行され, 3 が import されたときに表示されてしまします:

```
def get_raising_factorial(x,n):
  r=1
  for i in range(n):
    r=r*(x+i)
  return r

def get_falling_factorial(x,n):
  r=1
  for i in range(n):
    r=r*(x-i)
  return r

print get_falling_factorial(3,1)
```

import pochhammer で呼び出された場合と, python pochhammer.py で呼び出された場合で実行するかどうかを切り替えるには, __name__という変数に着目します. python から直接呼び出されているときには, __name__の値は__main__という文字列になっています. __name__の値で場合分けをし, 直接呼び出された時に実行することを記述します. 先ほどのプログラムであれば次のように書きます:

```
def get_raising_factorial(x,n):
  r=1
  for i in range(n):
    r=r*(x+i)
  return r

def get_falling_factorial(x,n):
  r=1
  for i in range(n):
    r=r*(x-i)
  return r


def main_function():
  print get_falling_factorial(3,1)

if __name__ == '__main__':
  main_function()
```

この様にかけば, 直接実行した時だけ, 3 が表示されます. 最後の if 文に続けて実行すべきことを直接書いても良いですが, 実行すべきことが長くなる時などは, 実行すべきことをまとめてメソッドにしてしまう方が, 読みやすくなるかもしれません.

2.13　コメントの話, マニュアル

コメントの方法について知っておくと良いことを述べます.

2.13.1　Docstring

Python では (文字列以外で) # があった場合そこから行末まで無視をします. ですので, 行の先頭に#を置いてその行にコメントを書くことができます.

```
# This is comment
print 2
```

また, インデントさえきちんとしていれば文字列からなる行があってもエラーにならないので, それを利用してコメントを書くこともあります.

```
def print23():
  """This is a comment. This function outputs 2 and 3."""
  print 2
  print 3
```

実はこのように文字列を使ってコメントを書くと良いこともあります. def の直後の行が文字列だった場合, その文字列はそのメソッドの使い方が書かれたコメントであると認識されます. この様なコメントは docstring とよばれます. 文字列を書く方法は色々ありましたが, docstring を書くときには, 通常"""と"""で挟みます. Sage を使用しているときに? を使うとマニュアルが表示されますが, 実は, このとき表示されるマニュアルは, このコメントをフォーマットすることで作成されたものです. ですので自分で定義したメソッドであっても?を使ってマニュアルを表示させることができます.

```
print23?
```

メソッドを定義するときは,

- 何を引数としてとるのか.
- 何が retuen されるのか.

ということ位は docstring としてメモを残す癖を付けましょう. 一言メモを残しておくだけで, 未来の自分という名前の他人にとっては, このプログラムを見た時の理解のしやすさが大きく変わります. プログラムを書いているときは面倒ですが, 一言メソッドの説明を書いておきましょう. また, 何か修正しなければいけないことがあるときなどは, FIXME などとかいて, 修正が必要なことと, 何を修正する必要があるのかということを書いておき, 修正せずそのまま使ってしまうことを防ぐと良いと思います.

2.13.2 Doctest

Docstring として, 何をするメソッドなのかについて簡単な説明をつけておくだけでも十分ではありますが, さらに, 簡単な実行例を載せておくと非常に理解がしやすいということはしばしばあります. Docstring に実行例を載せるときは, >>> の直後に実行するコマンドをまず書きます. そして, Python のインタプリタを立ち上げ対話的に実行した時の実行結果を次の行に書きます. 下降階乗 $x(x-1)(x-2)\cdots(x-n+1)$ を計算するメソッドを定義して, その使い方を説明するのであれば, 次のようにします.

```
def get_falling_factorial(x,n):
  """Returns x(x-1)..(x-n+1).

  >>> get_falling_factorial(10,0)
  1
```

```
  >>> get_falling_factorial(5,3)
  120
  """
  r=1
  for i in range(n):
    r=r*(x-i)
  return r
```

この様に例を書いてあると, 使い方を忘れてしまった時に非常に助かります. また, この様な形式にしたがって実行例を書いておくと, doctest というものを使って, 機械的にテストをすることができるので, 便利かもしれません. 例えば, 次のようなプログラムを用意し example.py というファイル名で保存したとします.

```
def get_falling_factorial(x,n):
  """Returns x(x-1)..(x-n+1).

  >>> get_falling_factorial(10,0)
  1
  >>> get_falling_factorial(5,3)
  120
  """
  r=1
  for i in range(n):
    r=r*(x-i)
  return r

def get_binomial_coef(x,n):
  """Returns xCn.

  >>> get_binomial_coef(5,2)
  10
  >>> get_binomial_coef(10,0)
  1
  >>> get_binomial_coef(2,3)
  0
  >>> get_binomial_coef(2,-1)
  0
  >>> get_binomial_coef(0.5,2)
  -0.125
  """
  if n < 0:
    return 0
  else:
    return get_falling_factorial(x,n)/get_falling_factorial(n,n)


if __name__ == '__main__':
```

```
    import doctest
    doctest.testmod()
```

この様なファイルを作り，python example.py を実行すると，2 つのメソッドが定義され，さらに，import doctest と doctest.testmod() が実行されます．doctest.testmod() は docstring から実行例を自動的に抜き出し, 実行し, 実行結果が一致するかをしらべます．もし実際に実行した結果と, 実行例として書いてあるものが一致するのであれば, 実行例が正しく動作していると判断します．もし全てが正しく動作した場合には, なにも表示されませんが, 何か正しく実行されなかった例があればそれが表示されます．もし, 何を実行しているの詳細を知りたければ python example.py -v を実行します．

この様な自動テストは複数のメソッドを定義した時に, 便利になります．例えば, 何かの都合で get_falling_factorial の定義を変更し, 引数の順番を入れ替えて次のようにしたとします．

```
def get_falling_factorial(n,x):
  r=1
  for i in range(n):
    r=r*(x-i)
  return r
```

この様な変更をくわえたら, それを使っているメソッドも修正しないといけません．今の場合であれば, get_binomial_coef は get_falling_factorial を使っていますから, これをきちんと修正しないといけません．修正をせずに, python example.py を実行すると, get_binomial_coef をテストした時に, 正しく実行されなかったと表示されます．自動テストをすれば, うっかり修正し忘れることを防げます．もし複数のメソッドを書いたり, 複数の人で協力してプログラムを作成するのであれば, 自動テストをして, 各メソッドごとにチェックをしておけば, 未然に色々なミスを防げます．

より詳しいことについては, 他の文献に譲ります．doctest というキーワードで調べると良いかもしれません．

2.14 名前の付け方について

ここでは, 変数やメソッドなどの名前の付け方について説明します．これらの名前は, 予約語と同じ名前でなければ基本的にはどの様なものでも良いのですが, 名前の付け方に

ルールがあると, プログラムを読みやすくなることがあります. さらに, 他の人が使っている標準的なルールに合わせておけば, 他の人にとって読みやすいプログラムになります. 名前の付け方のルールのことを, **命名規則**, **ネーミング**などと言う事もあります. 名前の付け方に対するルールについて, 基本的なものだけを述べておきます.

一般的に, 名前をつけるときには英語を用いますが, 複数の単語からなる名前をつける際にどのように一つの語にするかにいくつかの流儀があります. 主なものについて “Bacon lettuce tomato” を 1 つの語にすることを例に取って紹介します.

lower_case_with_under　全て小文字で書き, 単語間には_を置きます. (Lower) snake 形式と呼ばれる場合もあるようです.

例: `bacon_lettuce_tomato`.

UPPER_CASE_WITH_UNDER　全て大文字で書き, 単語間には_を置きます. Upper snake 形式と呼ばれる場合もあるようです.

例: `BACON_LETTUCE_TOMATO`.

CapitalizedWords　各単語の最初の文字のみ大文字で書き, 単語を詰めて書きます. Pascal 形式, upper camel 形式と呼ばれる場合もあるようです. もし単語の中に省略語がある時には, その語は全て大文字で書きます. 例えば, `HttpServer` ではなく `HTTPServer` とします.

例: `BaconLettuceTomato`.

mixedCase　最初の単語は全て小文字で書き, 2 つめ以降の単語は最初の文字のみ大文字で書きます. 単語を詰めて書きます. (Lower) camel 形式と呼ばれる場合もあるようです.

例: `baconLettuceTomato`.

lowercase　全て小文字で書き, 全てつなぎます.

例: `baconlettucetomato`.

UPPERCASE　全て大文字で書き, 全てつなぎます.

例: `BACONLETTUCETOMATO`.

ABBR　各単語の最初の文字を大文字でつなげて書きます.

例: `BLT`.

これらの方法を何の名前かによって使い分けることで, プログラムを読みやすくします. Python でプログラムを書くときにはは次のような作法に則るとよいでしょう:

メソッド　`lower_case_with_under`

変数　`lower_case_with_under`

定数　`UPPER_CASE_WITH_UNDER`

パッケージ, モジュール `lowercase`
クラス `CapitalizedWords`
例外 `CapitalizedWords`

この方法に従わなくてもエラーが出るわけではありませんが, 読みやすさを確保するために, この方法に従った方がよいです.

また, 例えば, 関数の引数に指定したいものの名前が, 予約語などと衝突してしまい, 別の名前を付けなければならないときがあります. この様な時に, 省略形を使ったりスペルをいじって解決しようとしてはいけません. 特に, `o` を `0` に変える, `l` を `1` に変えるといった, いわゆるギャル文字の様なことは絶対にやってはいけません.[*1] 後からプログラムを更新することになった際に, うっかり本来使うべきではない方を使って書き換えてしまうなどのミスを呼び, バグの潜在的な要因になりかねません. この様なときには, 次の方法で解決するようにしましょう:

- まず, 同義語を使って言い換えることができないか考えてみましょう. 言い換えることで別の名前にすることができるのであれば, そちらの方が良いことが多いです.
- 言い換えることができないのであれば, 末尾にに`_`を一つつけることで衝突を避けるようにしましょう. 例えば, 引数の名前として pass を使いたい時には, `pass_`とします.

[*1] このような書き換えを leet speek といったりします. Leet は, エリートと掛けて, いわゆる中二病と同じニュアンスをもって揶揄のために使われることもしばしばあります. あまり格好のいいものではなく, どちらかというと ‘イタイ’ 感じのものです.

第 3 章

ちょっと高度な話

ここでは, 少し込み入った話をします. ここで説明することは, 通常のプログラミンでは, あまり気にする必要はないかもしれません. しかしながら, 知っていると役立つこともあるかもしれません.

3.1 スコープの話, global

変数がいつ確保されるかということを説明します. Python では, 基本的には代入が起こった時に変数が確保されます. まだ確保されていない変数にアクセスをしようとするとエラーとなります. 例えば, いきなり次を実行するとエラーとなります.

```
print xxx1
```

実行結果の例

```
NameError: name 'xxx1' is not defined
```

次の場合はエラーにはなりません.

```
xxx2=1
print xxx2
```

実行結果の例

```
1
```

メソッドの中でも基本的には同じです.

```
def f():
  print xxx3
```

と定義はできますが, 次のようにメソッドを呼び出すと, エラーとなります.

```
f()
```

実行結果の例

```
NameError: global name 'xxx3' is not defined
```

次に, メソッド f の中で定義された変数はどのように振る舞うか説明します.

```
def f():
  xxx4=4
  print xxx4
```

とメソッドを定義して

```
f()
print xxx4
```

実行結果の例

```
NameError: name 'xxx4' is not defined
```

とすると, エラーとなります. つまりメソッド f の中で使われている変数 xxx4 は, 基本的にはそのメソッドの中で閉じており, 外側からには影響を与えません. ですので, 最後の行でエラーとなります. 例えば次のようにメソッドの外側で, 同じ名前の変数があった場合を考えます.

```
xxx5=10

def f():
  xxx5=5
  print xxx5
```

と定義し次を実行してみましょう.

```
f()
print xxx5
```

実行結果の例

```
5
10
```

この場合はメソッド f の中で代入があった時点で, f の中だけで使用できる変数が作られそこに 5 が代入されます. この変数は外側の変数とは別の変数ですので最後の print 文では, 10 が表示されます.

では, メソッドの中で代入が起こらない場合はどうでしょうか. 例えば次のようにメソッドを定義して

```
def f():
  print xxx6
```

次を実行した場合は, 実はエラーとならず 6 が表示されます.

```
xxx6=6
f()
```

実行結果の例

```
6
```

つまり代入が無い場合は, 外側の変数も含めて見つかった変数を呼び出します. では次の場合はどうなるでしょうか.

```
def f():
  print xxx7
  xxx7=77
```

と定義し次を実行するとエラーとなります.

```
xxx7=7
f()
```

実行結果の例

```
UnboundLocalError: local variable 'xxx7' referenced before assignment
```

まとめると, Python の変数の名前解決は次のようになっています.

メソッドの中のどこかで代入が起こっているもの local な変数 (つまりメソッドの中でだけ使われる変数) として扱われる.

メソッドの中で代入が起こっていないもの　global な変数 (メソッドのそとでも使われる変数) として扱われる.

どちらの場合でも変数にアクセスする前に, 代入などが行われていないとエラーとなります.

例えば

```
def f():
  xxx8=xxx8+1
  print xxx8
```

と定義し, 次を実行するとエラーとなります.

```
xxx8=8
f()
```

実行結果の例

```
UnboundLocalError: local variable 'xxx8' referenced before assignment
```

なぜエラーになるかというと, メソッド f のなかで xxx8 に代入する箇所が 1 ヶ所あります. ですので xxx8 は local 変数です. f が呼ばれる前に外側で xxx8 に代入されていますが, この global な xxx8 と local の xxx8 は異なる変数です. メソッド f の 1 行目では, “(local な)xxx8 を呼び出してその値と 1 とを足してから (local な)xxx8 に代入する” ということをしようとしていますが, 最初の xxx8 を呼び出す段階ではまだ一度も代入がされていませんので, エラーとなるわけです.

もし外側の変数の内容を利用したいだけなら, 一旦 local な変数に代入した上で使えば良く, 例えば,

```
def f():
  xxx10=xxx9
  xxx10=xxx10+1
  print xxx10
```

と定義し, 次を実行した場合はエラーとなりません.

```
xxx9=8
f()
```

実行結果の例

```
9
```

メソッドの中から外側の変数にアクセスしたり, 代入したりするにはどうしたら良いでしょうか. メソッドの外側の変数に直接代入するよりは, 計算結果を return で返す方が良いと思います. ただ, 次のように global というのをつければ, メソッドの中で代入がある変数であっても, global 変数であると認識させることができ, 外側の変数に代入することができます.

```
def f():
  global xxx12
  xxx12=5
```

と定義し, 次を実行すると, 12 と 5 が表示されます.

```
xxx12=12
print xxx12
f()
print xxx12
```

実行結果の例

```
12
5
```

外側の変数の値が書き変わっていることが分かります.

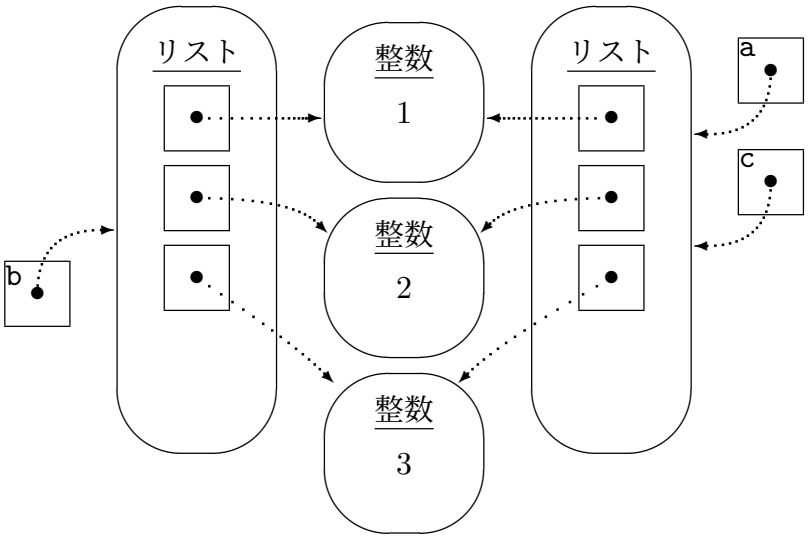
3.2 コピーの深さ

リストに対しスライス利用してコピーを得ることができました. 次を実行すると内部では, 図 3.1(a) の様になります.

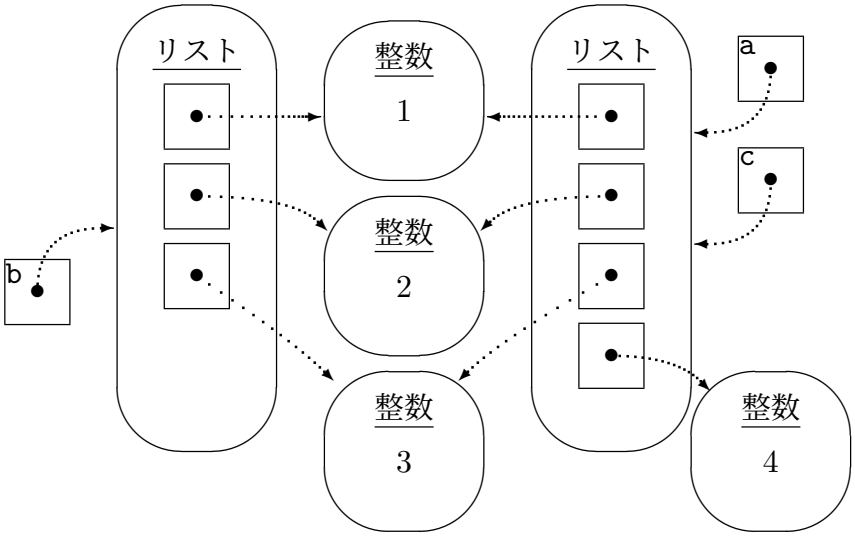
```
a=[1,2,3]
b=a[:]
c=a
print a
print b
print c
```

実行結果の例

```
[1, 2, 3]
[1, 2, 3]
[1, 2, 3]
```



(a) `a=[1,2,3]; b=a[:]; c=a`



(b) `a.append(4)`

図 3.1　`a=[1,2,3]; b=a[:]; c=a; a.append(4)`

コピーをしたものは, 同一のものではないので, リストに変更を加えても, 基本的には影響を受けません. 例えば, 次を実行すると図 3.1(b) の様になります.

```
a.append(4)
print a
print b
print c
```

実行結果の例

```
[1, 2, 3, 4]
[1, 2, 3]
[1, 2, 3, 4]
```

今は, リストについて見てみましたが, 次に, リストのリストについて考えてみます. リストの要素としてリストが含まれているような場合は, 少し気をつけて考える必要があります. まず, 先ほどと同じようにリストを作成し, スライスでコピーを行います.

```
a=[[10,11],2]
b=a[:]
print a
print b
```

実行結果の例

```
[[10, 11], 2]
[[10, 11], 2]
```

この後に, 次を実行すると b だけ変更されていることが分かります.

```
b[2]=9
print a
print b
```

実行結果の例

```
[[10, 11], 2]
[[10, 11], 9]
```

今の場合は, 最初の要素もリストですので, その要素に変更を直接加えてみたらどうなるでしょうか.

```
b[0][1]=17
print a
print b
```

実行結果の例

```
[[10, 17], 2]
[[10, 17], 9]
```

この時は, b へ変更を加えた所, それにつられて, a の内容まで変更されているように見えます. 実は, スライスでコピーされるのは一番外側のリストの部分だけで, 各要素までコピーされるわけではありません. (このようなコピーを**浅いコピー**といいます.) ですので, a[0] と b[0] はまったく同一のリストを指しています. したがって b[0] に直接変更を加えると, a[0] まで内容が変更されてしまいます.

今, 見たことを図で考えると次のようになります. リストのコピーについて考えましょう. 例えば, a=[[10,11],2] を行うと, 図 3.2(a) のようにオブジェクトが作られます. これに対して, b=a[:] とやると a が指しているリストの持っている情報を元にリストが作られます. リストには, 各オブジェクトどこにあるかという情報が記録されているわけですが, その情報がコピーされます. 各要素のコピーが作られるわけではありません. リストの要素にリストがあった場合も同様です. したがって図 3.2(b) の様になります. いまの状態で a==b を調べると True となります. もしここで b[1]=9 とすると b の指しているリストが書き換わります. 図 3.3(a) のような状態です. このときには a の指しているリストには影響がありません. ここでさらに, b[0][1]=17 とすることを考えましょう. このとき, b[0] は [10,11] というリストを指しており, そのリストが書き換えられます. つまり図 3.3(b) の様な状況になるのですが, a[0] も同じリストを指しています. したがって, この場合は結果的に a が指しているリストにも, 実質影響を与えることになります.

b[0] に別のリストを代入するのは, b[0] の指すリストに直接変更を加えるということとは違いますので, 次の場合は b には変更が起こりません.

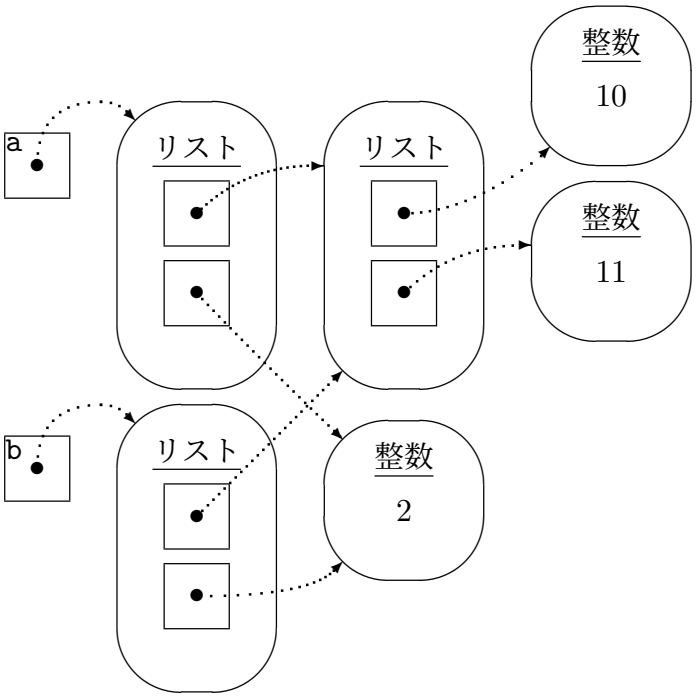
```
b[0]=[100,101]
print a
print b
```

実行結果の例

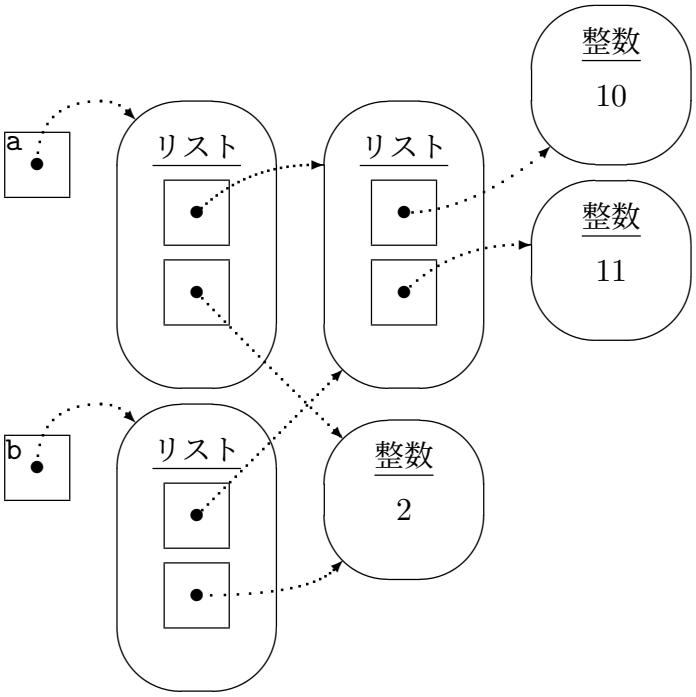
```
[[10, 17], 2]
[[100, 101], 9]
```

リストのような書き換え可能なオブジェクトの場合, 等しいオブジェクトといった場合, 内容が同じという意味で等しいオブジェクトと本当に同一なオブジェクトであるということの違いが重要になることがあります. 内容が等しいかどうかを調べるには a == b のようにします. もし, a と b の内容が等しければ, True となります. 一方, 同一のオブジェクトをさしているかを調べるには is を使います. もし, a と b がともに同じオブジェクトを指しているときには, a is b は True となります.

例えば, まず x=0 とし, y=2 とし, z=4 としておきましょう. このときに次を実行したらどうなるか考えてみましょう: a=[x,y] としたあと, b=a とし, c=[x,y] としましょう. このときは, まず a=[x,y] の右辺が評価され, 新しくリストが生成され, それがどこにあ

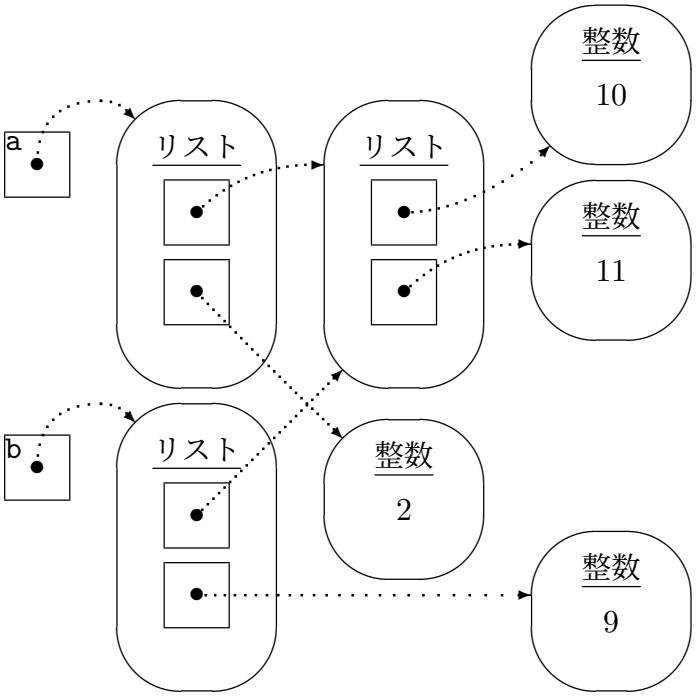


(a) `a=[[1,3],5]; b=a[:]`

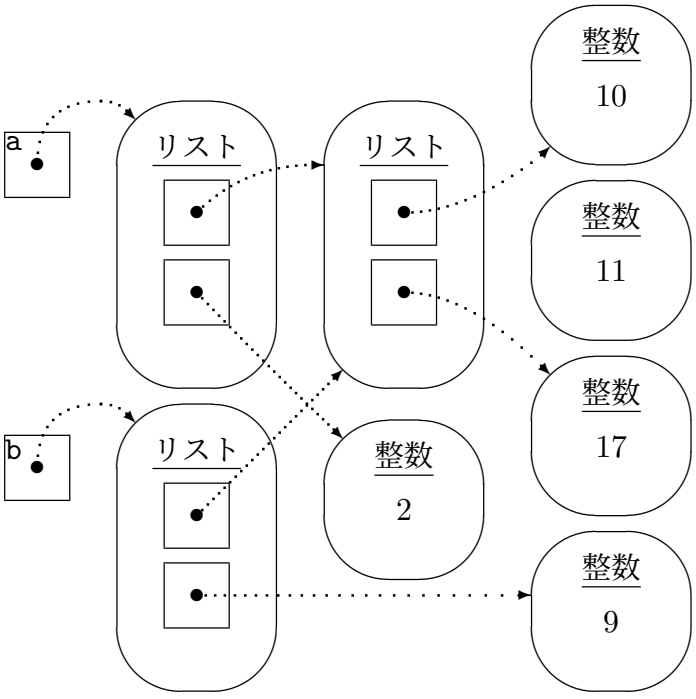


(b) `a=[[1,3],5]; b=a[:]`

図 3.2　`a=[[10,11],3];` のときの挙動



(a) `b[1]=5`



(b) `a[0][1]=17`

図 3.3　`b[1]=5; a[0][1]=17` のときの挙動

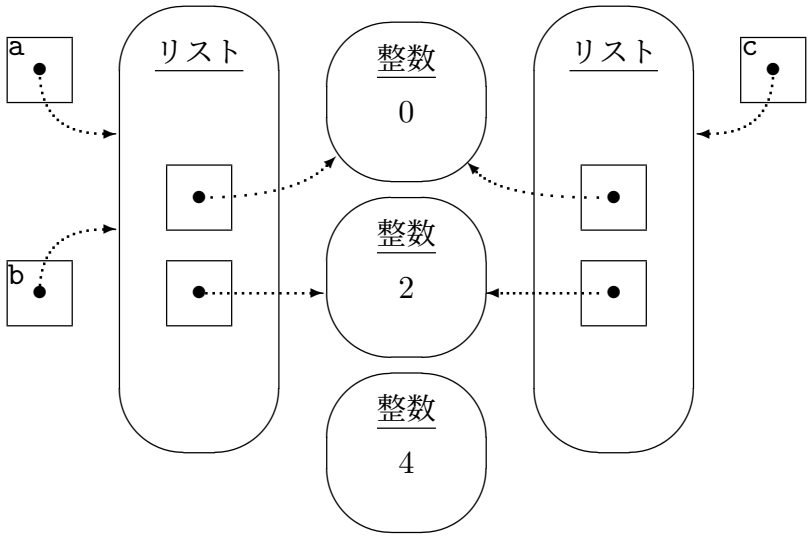
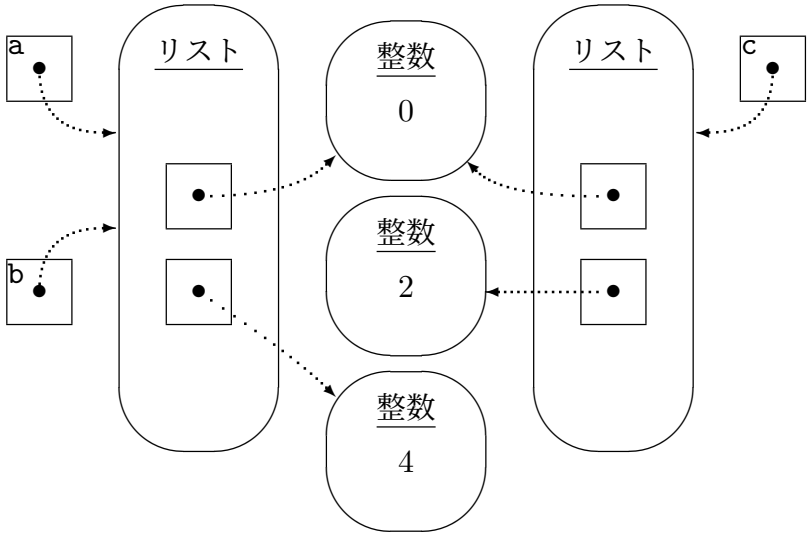


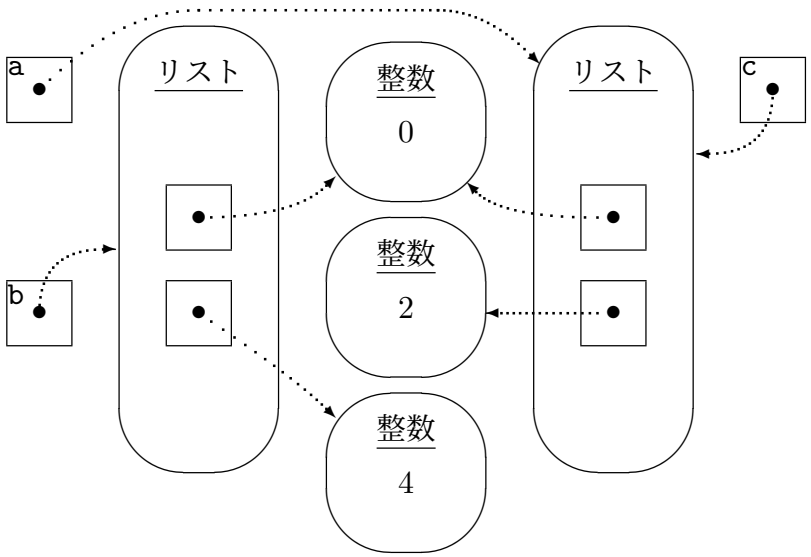
図 3.4 a=[x,y]; b=a; c=[x,y]

るかという情報が a に記録されます．その後, b=a が評価され a に記録されている情報が b に記録されます．その後, c=[x,y] が実行されます．このとき, まずの右辺が評価され, 新しくリストが生成され, それがどこにあるかという情報が c に記録されます．この時点で, a と b は同一のオブジェクトを指していますが, c は別のオブジェクトを指しています．今の状況を図示すうると図 3.4 のようになります．ただし, x, y, z は議論に置いて本質的ではないので省略しました．今の状況であれば, ==を使って調べると, a == b も a == c も b == c も True です．一方, is を使って調べると, a is b も True ですが, a is c も b is c も False です．このとき, a[1]=z とすると, a の指しているオブジェクトに変更が加わり, 図 3.5(a) のような状況になります．a==[0,4] も b==[0,4] も True となりますが, c==[0,4] は False です．また, さらに, a=c とすると, a に記録されている情報が, 新しく作られた c の指している場所に, 書き換わり図 3.5(b) のような状況になります．b には何も変化は起こりません．この場合, a is c は True であり, a is b と b is c は False となります．

メソッドに引数として渡される情報も, オブジェクトがどこにあるかという情報です．ですので, 書き換え可能なオブジェクトに書き換える操作をすると, オブジェクトに変更が加わるため注意が必要です．



(a) `a[1]=z`



(b) `a=c`

図 3.5　`a[0]=z; a=c` のときの挙動

3.3 Default 引数

Python では, デフォルト引数とかキーワード引数と呼ばれるものを持つメソッドを定義することができます. 例えば, 次のようなメソッドを定義しましょう.

```
def modb(a,b=2):
  return a % b
```

このメソッドは次のように実行すると a=8, b=3 の場合として実行します.

```
print modb(8,3)
```

実行結果の例

```
2
```

もし b の部分に何も書かず引数をひとつしか与えなかった場合には, b=2 として処理をすすめます. 例えば, 次のように実行すると a=8, b=2 の場合として実行します.

```
print modb(8)
```

実行結果の例

```
0
```

次のように b=3 の場合であることを明示して実行することもできます.

```
print modb(8,b=3)
```

実行結果の例

```
2
```

複数の引数がある時には, デフォルト値を指定する引数は後ろの方にまとめて書くという決まりがあります. 例えば次のような定義ではエラーとなってしまいます.

```
def printab(a=100,b):
  print (a,b)
```

実行結果の例

```
SyntaxError: non-default argument follows default argument
```

このルールさえ守っていれば, 例えば, 次のようなメソッドも定義することができます.

```
def printabcd(a,b,c=100,d=1000):
  print (a,b,c,d)
```

この場合, 次のように実行することができます.

```
printabcd(1,2)
printabcd(1,2,3)
printabcd(1,2,3,4)
```

実行結果の例

```
(1, 2, 100, 1000)
(1, 2, 3, 1000)
(1, 2, 3, 4)
```

引数が与えられていないところはデフォルトの値が使われています. 明示的に変数名を指定することで, c の値はデフォルトで d の値のみを指定するということもできます.

```
printabcd(1,2,c=3)
printabcd(1,2,d=4)
printabcd(1,2,c=3,d=4)
printabcd(1,2,d=4,c=3)
```

実行結果の例

```
(1, 2, 3, 1000)
(1, 2, 100, 4)
(1, 2, 3, 4)
(1, 2, 3, 4)
```

デフォルトの値として, 変更可能なコレクションを与えた時に, どのような振る舞いをするかということを説明します.

まず

```
def printa(a=1):
  a=a+10
  print a
```

というメソッドを定義して, 次を実行しましょう.

```
printa()
```

```
printa()
```

実行結果の例

```
11
11
```

この時, 11 と 11 が表示されます. a=a+10 で代入されるのは, local 変数でであり, デフォルト値を書き換えるわけではないので, 二回目の時も 11 が表示されています.

次に,

```
def printa(a=[1]):
  print a
```

というメソッドを定義しましょう. この時, 次を実行すると [1] と [1] が表示されると思います.

```
printa()
printa()
```

実行結果の例

```
[1]
[1]
```

では次のようなメソッドを定義しましょう.

```
def printa(a=[1]):
  a.append(5)
  print a
```

この時, 次を実行すると, [1,5] と [1,5,5] が表示されると思います.

```
printa()
printa()
```

実行結果の例

```
[1, 5]
[1, 5, 5]
```

実はデフォルトの値として使われるリストの評価はメソッドの定義の時に一度だけされ, その時に作成されたリストがデフォルトの値として使いまわされます. したがって, もし直接リストに変更を加えた場合, それ以降で使われるリストは変更が加えられたリストと

なります.

一方で

```
def printa(a=[1]):
  a=a+[5]
  print a
```

と定義して, 次を実行した場合は, [1,5] と [1,5] が表示されると思います.

```
printa()
printa()
```

実行結果の例

```
[1, 5]
[1, 5]
```

a=a+[5] で起こるのは, a のコピーに 5 という要素を付け加えたリストを a に代入するということです. デフォルト値として用意されているリストを直接書き換えるわけではないので, 二回目も同じ結果を得ます.

3.4　lambda

lambda というキーワードについて説明します. Python ではメソッドも変数に代入することができました. ですのでメソッドを引数にとるメソッドを定義することができます. 例えば, 次のようにメソッドを定義します.

```
def print1234(f):
  for i in range(4):
    print f(i),
```

これに対し, 引数としてメソッドを与えれば 0, 1, 2, 3 での値を表示することができます.

```
def sq(n):
  return n*n
```

と定義し, 次のように使うと, 0, 1, 4, 9 が表示されます.

```
print1234(sq)
```

実行結果の例

```
0 1 4 9
```

もちろん違うメソッドでも可能です.

```
def minus(n):
  return -n
```

と定義し, 次を実行すると, 0, −1, −2, −3 が表示されます.

```
print1234(minus)
```

実行結果の例

```
0 -1 -2 -3
```

print1234 というメソッドの中では f(i) というものを実行していますので, 例えば整数を引数に与えると f(i) が実行できず, エラーとなります. (カッコを付けて値を呼ぶことができるものを *callable* といっています.)

```
print1234(8)
```

実行結果の例

```
TypeError: 'int' object is not callable
```

さて, もしメソッドの引数として渡すだけであれば, わざわざ名前を付けて定義する必要はありません. このような時に lambda を使います. 例えば t に対し t^2 を返すメソッドを引数と与えたいなら, 次のように書くことができます..

```
print1234(lambda t: t*t)
```

実行結果の例

```
0 1 4 9
```

このように簡単なメソッドであれば, 名前をつけず無名のメソッドとして引数として渡す方が便利です. 無名のメソッドを扱う時に, lambda というキーワードを使います.

2 つ以上の引数をとるようなメソッドも作れます. 例えば, 次のメソッドを考えましょう.

```
def print23(f):
  print f(2,3)
```

このように f(2,3) の値を表示するメソッドに対しては, 普通はまず次のようにメソッドを定義します.

```
def mul(x,y):
  return x*y
```

その上で, 次のようにメソッドを定義して引数として与えます.

```
print23(mul)
```

実行結果の例

```
6
```

しかし, 次のように lambda を使って書くことも出来ます.

```
print23(lambda x,y: x*y)
```

実行結果の例

```
6
```

付録 A

Sage ならではの話

ここでは, pure Python ではなく Sage を使う際に気を付けておくべきことなどについて説明をします.

A.1 表示について

Sage Notebook ではセルに書いた最後のものが変数の場合には, 自動的に内容が表示されます.

```
a = 8
a
a = a + 13
a
```

実行結果の例

```
21
```

表示のされ方は, `print` の時とは少し異なります. 実際には `a.show()` もしくは `a._repr_()` といったメソッドが呼ばれています.

A.2 ZZ v.s. int

Sage は, 数学の “カテゴリ (圏)” を意識して作られています. それぞれの値は, どういう集合の元か, そして, 集合はどういう構造を持っているかということを知っています. 例えば, 1/2 は有理数体 `QQ` の元であり, `QQ` は体という構造を持っています. ある値がどの集合の元として認識されているかは, 次のように調べることができます.

```
a=1/2
```

```
print a.category()
```

実行結果の例

```
Category of elements of Rational Field
```

1 は有理数でもあり, 整数でもあります. デフォルトでは 1 と書いた時有理整数環 ZZ の元として認識されています.

```
a=1
print a.category()
```

実行結果の例

```
Category of elements of Integer Ring
```

割り算などをすると自動的に有理数体の元になるなどするので, 通常の計算は不自由することはありません.

```
a=1
a=a/2
print a.category()
```

実行結果の例

```
Category of elements of Rational Field
```

しかしながらある集合に属する値を, 別の集合での対応する値に読み替えたいことがあります. 例えば, 厳密計算ができる有理数体 QQ の元 1/2 を近似実数体 RR の元 0.5 に変換したい時には次のようにします.

```
a=1/2
print a.category()
print a

b=RR(a)
print b.category()
print b
```

実行結果の例

```
Category of elements of Rational Field
1/2
Category of elements of Real Field with 53 bits of precision
0.500000000000000
```

RR(a) とすると a の値に相当する RR の元が返ってきます.

例えば, 次を実行すると F3 には, 3 元体が代入されます.

```
F3=FiniteField(3)
```

これを使って, F3(6) とすると, 6 で代表される $\mathbb{F}_3$ の元が返ってきます.

```
a=6
print a.category()

b=F3(a)
print b.category()
print b
```

実行結果の例

```
Category of elements of Integer Ring
Category of elements of Finite Field of size 3
0
```

さて, 1 と書いた場合, ZZ の元として認識されました.

```
a=1
print a.category()
```

実行結果の例

```
Category of elements of Integer Ring
```

この 1 は Sage で定義されている ZZ の元の 1 であり pure Python の int としての 1 ではないため, Sage で定義される様々な機能の恩恵を受けることができます. Sage で計算する際, 通常の計算の範疇で出てくる整数は, ZZ の元です. しかしながら, len などの Python のメソッドを使って得られる整数は, pure Python の int であり, ZZ の元ではありません. 従って, その整数はカテゴリなどの情報を持っていません. 例えば次は実行できますが

```
a=[0,1]
b=len(a)
print type(b)
```

実行結果の例

```
<type 'int'>
```

次はエラーになります.

```
b.category()
```

実行結果の例

```
AttributeError: 'int' object has no attribute 'category'
```

このような pure Python の int が返ってくる場合は, ZZ を使って対応する ZZ の元を使う方が便利な場合があります.

```
a=[0,1]
b=ZZ(len(a))
print type(b)
b.category()
```

実行結果の例

```
<type 'sage.rings.integer.Integer'>
Category of elements of Integer Ring
```

A.3 不定元

Python の変数は, 代入により生成され, 呼び出した際には常に何らかの具体的な値が代入され割り当てられています. 例えば次の例では, a には, 3 が割り当てられており, a^2+1 は 10 という具体的な値です.

```
a=3
print a^2+1
```

実行結果の例

```
10
```

これは, 数学で使われるシンボリックな変数 (= 不定元) とは異なります. Sage では不定元を扱うこともできます. 不定元を使うには, 例えば次のように var('a') というのを使います.

```
var('a')
print a^2+1
```

実行結果の例

```
a^2 + 1
```

var('a') とすると, 変数 a に 'a' という名前の不定元が代入されます. 複数の不定元を定義するには, カンマで区切るか, スペースで区切ることで, まとめて定義することもできます.

```
var('a, b')
print a+b
var('c d')
print c+d
```

実行結果の例

```
a + b
c + d
```

不定元の名前に使う長さは 1 である必要はありません.

```
var('alpha')
print a+alpha
```

実行結果の例

```
a + alpha
```

var で不定元を定義すると, 定義された不定元が一個の場合はその不定元が, 定義された不定元が複数の場合は定義された不定元のタプルが, 返って来ます.

```
tt=var('t')
print tt
```

実行結果の例

```
t
```

```
aa=var('a1, a2')
print aa
```

実行結果の例

```
(a1, a2)
```

この事を利用して次のように複数の不定元を定義することも出来ます.

```
a=[ var('a_%d' % i) for i in xrange(10)]
print a
print a[0]
print a_0
```

実行結果の例

```
[a_0, a_1, a_2, a_3, a_4, a_5, a_6, a_7, a_8, a_9]
a_0
a_0
```

Sage を立ち上げると, 自動で `var('x')` が実行され, 変数 `x` に不定元 x が代入されています.

`var` を使う時にオプションを指定することもできます. 例えば, 不定元がどのカテゴリの元であるかを指定することもできます.

```
var('n', domain=ZZ)
```

実行結果の例

```
n
```

typeset を行う時に使う TEX コマンドを指定することもできます.

```
var('alpha', latex_name='\alpha')
var('sui', latex_name='\color{red}{s_{u,i}}')
(alpha,sui)
```

実行結果の例

```
(alpha, sui)
```

詳細は Help を見てください.

A.4 上書きされているいくつかのもの

数学を記述する上で便利 (直感的 or 伝統的) になるように, Sage ではいくつかの記号が上書きされています. Pure Python ではべき乗を表すのに**を使いましたが, Sage ではべき乗を表すのに^を使います. Pure Python では^はビット演算を表していましたので, 気を付ける必要があります.

```
print 2^3
```

実行結果の例

```
8
```

また, m から $n-1$ までの整数からなるリストを表すのに, Python では `range(m,n)` と書いていました. もちろんこの書き方も使用できますが, 同じことを, `(m..n-1)` と書くことができます. 最後の値の指定の仕方が異なりますので気を付ける必要があります.

```
for i in range(-1,2):
  print i,
```

実行結果の例

```
-1 0 1
```

```
for i in (-1 .. 1):
  print i,
```

実行結果の例

```
-1 0 1
```

A.5 数学定数

有名な定数が定義されており, Sage の起動時に変数に自動的に代入されています. 例えば次のようなものがあります.

- `e` $= e$,
- `pi` $= \pi$,
- `i` $= i$,
- `oo` $= \infty$,
- `log2` $= \log 2$,
- `NaN` $=$ Not a number, 例えば, $0/0$ など.

他にも次のようなものもあります.

`brun catalan euler_gamma golden_ratio khinchin mertens twinprime` ...

これらは, 厳密計算が可能なようにシンボルとして定義されています. もし近似値が必要な場合には, 対応する `RR` などの元を使うか, `n` というメソッドを呼び出します.

```
a=e^log2
print a

b=a.simplify()
print b

c=RR(a)
print c

d=a.n()
print d
```

実行結果の例

```
e^log2
2
2.00000000000000
2.00000000000000
```

また, 数学定数ではありませんが, x には不定元 x が代入されています.

A.6　load v.s. attach

プログラムを Python で実行する方法は冒頭で説明したとおりです. ここではプログラムを Sage で実行する方法を説明します.

ターミナルで sage というコマンドを実行します. すると, いくつかの出力が合った後, 入力待ちになります.

この状況からは, Python を対話的に使ったように使用することができます. また, 通常の対話的な Python にさらに機能が追加されており, tab キーによる補完などができるようになっています. さらに, ここからプログラムを書いたファイルを読み込むこともできます. ファイルを読み込む方法は 2 つあります. 具体例を使って説明をします.

ひとつ目の方法は, load を使ってファイルを読み込むという方法です. まず

```
def test1():
  print 'This is test.'
```

というメソッドの定義を書いたファイルを, example1.sage という名前で保存しましょう. ターミナルで Sage を立ち上げた後, 次を実行しましょう.

```
load('example1.sage')
```

すると, そのファイルが読み込まれ, test1 という名前のメソッドが定義されます.

```
test1()
```

実行結果の例

```
This is test.
```

と実行すると This is test. が表示されると思います. load はこのコマンドを使った時にファイルを読み込みます. ファイルを変更しても, もう一度 load を使わない限り test1 の定義は変更されません.

もうひとつの方法は attach を使う方法です. まず

```
def test2():
  print 'This is test.'
```

というメソッドの定義を書いたファイルを, example2.sage という名前で保存しましょう. ターミナルで Sage を立ち上げた後, 次を実行しましょう.

```
attach('example2.sage')
```

すると, そのファイルが読み込まれ, test2 という名前のメソッドが定義されます. 次を実行すると This is test. が表示されると思います.

```
test2()
```

実行結果の例

```
This is test.
```

attach を行った場合には, Sage はファイルの更新を監視しています. ファイルが更新された場合, 次にコマンドを実行する前に, ファイルを再度読み込みます. 例えば,

```
def test2():
  print 'This function is test2.'
```

という内容で example2.sage を保存します. そして, test2() を実行すると test2 の定義が変更され This function is test2. と出力されます.

A.7 実行速度

Sage で実行速度を調べる方法について説明します. Sage で, あるメソッドの実行にかかる時間を調べるには, time というものを使います. 例えば, 次のような, ただ単に 10000000 回 1 を足し続けるメソッドを定義しましょう.

```
def sum_test():
    m=10000000
    k=0
    for i in xrange(m):
        k=k+1
    print k
```

通常であれば, このメソッドを次のように実行します.

```
sum_test()
```

実行結果の例

```
10000000
```

この実行の際に time と書くことで実行時間を測ることできます. 具体的には次の様に実行します.

```
time sum_test()
```

実行結果の例

```
10000000
Time: CPU 0.92 s, Wall: 0.92 s
```

するとメソッドを実行した後にその実行にかかった時間を表示します. 仮想マシンを使っている時にはこの時間は不正確であるかもしれませんが, 一応の目安にはなると思います.

また, Sage には timeit というメソッドが用意されています. このメソッドは, 数回実行してかかった時間が最短のものを表示します. 実行するコマンドを文字列として与えます.

```
timeit('sum_test()')
```

実行結果の例

```
5 loops, best of 3: 940 ms per loop
```

A.8 Memoization

Fibonacci 数列を計算する事を例にとって考えましょう. Fibonacci 数列は, $a_0 = a_1 = 1$, $a_n = a_{n-2} + a_{n-1}$ と定義されますので, 例えば次のように定義すると Fibonacci 数列 a_n を計算することができます.

```
def fib(n):
    if n<2:
        return 1
    else:
        return fib(n-2)+fib(n-1)
```

ただしこの方法では時間がかかってしまいます.

```
time fib(31)
```

実行結果の例

```
2178309
Time: CPU 9.74 s, Wall: 10.70 s
```

この様に計算に時間のかかるメソッドを何度も呼び出すと時間のロスにつながります. 一度計算した値はどこかに引数と値の表をメモしておいて, もしまったく同じ引数であれば, 再度計算せずにそのメモにある値を使いまわした方が, 速度に関しては速くなります. (ただし, 速度を速くするためのコストとして記憶領域を使っていることには注意が必要です.) この様な技法を *memoization* と言います.

Sage には memoization を簡単に行う方法が用意されています. `cached_function` というメソッドの引数に memoization したいメソッドを渡すと, memoization されたメソッドが返ってきます.

```
memoized_fib = cached_function(fib)
```

このように定義し `memoized_fib(n)` とすると, もし n での値がキャッシュにあればその値を返します. もし n での値がキャッシュになければ `fin(n)` を計算し, キャッシュに保存した上で値を返します. ですので一回目は時間がかかるかもしれませんが, 二回め以降は時間はかかりません.

```
time memoized_fib(31)
```

実行結果の例

```
2178309
Time: CPU 10.78 s, Wall: 12.38 s
```

```
time memoized_fib(31)
```

実行結果の例

```
2178309
Time: CPU 0.05 s, Wall: 0.26 s
```

なお, キャッシュに値が無い場合は, `fib(n)` が呼ばれます. `fib(n)` の中では `fib(n-2)` と `fib(n-1)` が呼ばれています. これは `memoized_fib(n)` では無いので, 途中経過の値はキャッシュには保存されません. ですので `memoized_fib(30)` を呼ぶと, やはり最初は時間がかかります.

```
time memoized_fib(30)
```

実行結果の例

```
1346269
Time: CPU 6.88 s, Wall: 9.60 s
```

```
time memoized_fib(30)
```

実行結果の例

```
1346269
Time: CPU 0.14 s, Wall: 0.35 s
```

fib(n) の中で memoized_fib(n-2) と memoized_fib(n-1) を呼ぶように変更すると, 途中の計算でもキャッシュを使用できるので, 非常にはやくなります.

```
def fib(n):
    if n<2:
        return 1
    else:
        return memoized_fib(n-2)+memoized_fib(n-1)

memoized_fib=cached_function(fib)
```

```
time memoized_fib(40)
```

実行結果の例

```
165580141
Time: CPU 0.28 s, Wall: 0.47 s
```

途中の fib というメソッドを定義する必要が無い場合は, 次の様にシンプルに書くこともできます.

```
@cached_function
def memoized_fib(n):
    if n<2:
        return 1
    else:
        return memoized_fib(n-2)+memoized_fib(n-1)
```

```
time memoized_fib(31)
```

実行結果の例

```
2178309
Time: CPU 0.00 s, Wall: 0.00 s
```

たとえば, 計算を途中で強制的に中断した場合にキャッシュとして正しくない値が記録されてしまうことがあります. このときは, `clear_cache()` でキャッシュを消すことができます.

```
memoized_fib.clear_cache()
```

Sage の中ではキャッシュは辞書 (dictionary) として記録されています. `get_cache()` でその辞書を確認できます. 辞書にキャッシュを保存するため, 引数は immutable なものしかとれないということに気をつける必要があるかもしれません.

```
memoized_fib(31)
memoized_fib.get_cache()
```

実行結果の例

```
((12,), ()): 233, ((20,), ()): 10946, ((17,), ()): 2584, ((5,), ()): 8,
 ((0,), ()): 1, ((30,), ()): 1346269, ((25,), ()): 121393, ((31,), ()):
 2178309, ((13,), ()): 377, ((10,), ()): 89, ((18,), ()): 4181, ((3,),
 ()): 3, ((28,), ()): 514229, ((11,), ()): 144, ((8,), ()): 34, ((6,),
 ()): 13, ((7,), ()): 21, ((21,), ()): 17711, ((16,), ()): 1597, ((29,),
 ()): 832040, ((26,), ()): 196418, ((4,), ()): 5, ((19,), ()): 6765,
 ((1,), ()): 1, ((27,), ()): 317811, ((14,), ()): 610, ((9,), ()): 55,
 ((15,), ()): 987, ((24,), ()): 75025, ((22,), ()): 28657, ((23,), ()):
 46368, ((2,), ()): 2
```

```
memoized_fib.clear_cache()
memoized_fib.get_cache()
```

実行結果の例

```

```

付録 B

Cython

ここではより速いプログラムを簡単に書くための方法として, Cython について簡単に説明をします. 詳しい使い方, 本格的に使うための方法については, 別の文献に譲ります.

B.1 Cython の簡単な説明

Cython の使い方を説明する前に, Cython とはどういうものなのかということを大雑把に説明します.

`python` はいわゆるインタプリタです. 書いたプログラムを実行時に逐次解釈しながら実行しています. 一方で, C 言語で書かれたプログラムは, まずコンパイルという作業でプログラムを翻訳し直接実行可能なファイルを作成してから, 作成されたファイルをコマンドとして実行します. コンパイルという手間はかかりますが, 直接実行可能なファイルを実行する方が, 逐次解釈をするよりも高速になります.

また, Python の変数には型の指定がありません. 例えば, `a` という変数に整数を代入しても文字列を代入してもエラーとなることはありません. この仕様は, 型を気にせずプログラムを書くことができるため手軽にプログラムを書ける反面, 実行の際にプログラムの内部でどの型の値が変数に代入されているかをチェックしなければならないことを意味しています. 例えば単純に `a+b` と書いてあっても, 整数の同士の時, 整数と小数の時, 文字列同士の時など場合によって実行すべきことが異なるため, 単に + を実行するときでも型のチェックをすることになります. もし変数に型が指定されていれば, 実行の際に ‘決め打ち’ で実行できるので型のチェックをせずにすみます.

Python は手軽にプログラムを書ける反面, 実行速度の面でハンデを負っています. Cython は, Python の手軽さを残したまま, C 言語のように高速なプログラムを書くためのツールです.

具体的には, Cython というのは Python とほぼ同じ文法をもつプログラミング言語で

す. Python とほぼ同じようにプログラムを書くことが出来ます. それに加え C 言語のメソッドを直接呼び出すことも出来ます.

Cython を使う手順を大雑把に説明します. まず, Cython の文法に従ってプログラムを書きファイル (`*.pyx`) として保存します. これを `cython` で処理すると, C 言語で書かれたプログラムのファイルが出来ます. このファイルを C 言語のコンパイラでコンパイルすると, Python の拡張モジュールが生成されます. この生成された拡張モジュールは Python のプログラムから呼び出し使用することが出来ます. このモジュールで定義されているメソッドを呼び出し実行するプログラムを Python で書き, そのプログラムを実行します.

Cython で書かれた部分については, Cython および C 言語のコンパイラによって計算機が直接実行可能な形式に変換されますので, 普通に Python で書いた時よりも実行速度は高速になることが期待されます.

Cython を使う手順を大雑把に説明しましたが, 実は Sage Notebook から使う方が簡単に使えますので, Sage Notebook での使用方法を説明をしたいと思います. まず Sage Notebook のセルの一行目に `%cython` と書きます. そして, そのセルに Cython のプログラムを記述します. そのセルを実行する (Shift を押しながら Enter を押す, または Evaluate をクリックする) と, コンパイルなどが行われそのセルで定義されているメソッドが実行可能な状況になります. `%cython` と書かれたセルを実行した時には, Cython で生成された C 言語のファイルが実行結果として現れます. すべて自動でやってくれますので, 実行結果として現れるファイルを自分で作業する必要は特になく, 無視しても構いません. この状態で, すでに定義したメソッドなどは使える状態になっています. 大雑把な流れについては説明しましたが, 今ひとつぴんとこないと思いますので, 次節ではもう少し具体的に実例を交えながら説明をしていこうと思います.

B.2 Cython を使った高速化の例

実行速度を早くしたい時には, 通常はアルゴリズム改良するのが得策です. しかしながら, Python のインタプリタが原因で遅くなっている場合には, アルゴリズムを変えずとも Cython を使うだけで, 速くくなることがあります. この事について, 節では具体例で眺めようと思います. Sage Notebook を実際に使いながら Cython の説明を簡単に行います. 詳細は他の文献に譲ります.

B.2.1 例 1

まずは普通の Python としてプログラムを組んでみましょう. 次をセルに書いて, 実行しましょう.

```
def test1(m):
    n=0
    for i in range(m):
        n=n+1
    return n
```

これで, 与えられた m に対し m 回 1 を足した数を返すというメソッドが定義されました. まずはこのメソッドの実行速度を計測しましょう. 例えば $m = 5{,}000{,}000$ の時にどれくらいの時間がかかるかを調べるには, 次を実行します.

```
timeit('test1(5000000)')
```

実行速度は計算機によって変わってくると思いますが, 例えば筆者のマシンでは `5 loops, best of 3: 580 ms per loop` と表示され, 580 ミリ秒で実行が終わったのが最速だったということがわかります.

さて, このメソッドを Cython を使って書き直すことで, 高速化してみましょう. 例えば, 次を別のセルに書いて, 実行してみましょう.

```
%cython
def test2(m):
    n=0
    for i in range(m):
        n=n+1
    return n
```

冒頭に `%cython` と一行加わっただけで他は変わっていません. このセルを実行すると, 自動的に Cython で処理され `test2` というメソッドが使用できるようになります. 定義されたメソッドは他のものと同じように使えます.

```
timeit('test2(5000000)')
```

とすると筆者のマシンでは 218 ミリ秒という結果になりました. 通常の Python で書かれたプログラムをほぼ手を入れず Cython にかけるだけでも, コンパイルされたものを実行する分だけ実行速度が上がります. 特に `for` 文がある時には飛躍的に早くなる可能性があります.

次に, 変数の型を指定することで高速化することを考えましょう. 変数の型を指定する時には `cdef` というものを使います. `cdef` の後に `int` や `double` といった C 言語での型を書きその後に変数を指定しすることで, 変数の型を宣言します. 例えば, 次のように書きます.

```
%cython
def test3(m):
    cdef int n
    cdef int i
    n=0
    for i in range(m):
        n=n+1
    return n
```

このように定義し

```
timeit('test3(5000000)')
```

を実行すると, 筆者のマシンでは 371 ナノ秒となりました. また変数の宣言は ‘,’ で並べてまとめて書くことも出来ます.

```
%cython
def test4(m):
    cdef int n, i
    n=0
    for i in range(m):
        n=n+1
    return n
```

またすべての変数の型を宣言する必要はなく, 一部だけ指定することも可能です.

```
%cython
def test5(m):
    cdef int n
    n=0
    for i in range(m):
        n=n+1
```

```
    return n
```

プログラムを読みにくくすることにつながるので, 闇雲にすべての変数の型を指定することは推奨されていません. 実際, 高速化のためにはすべての変数の型を指定する必要はなく, ボトルネックとなる変数にだけ型を指定するだけでも劇的に速くなることがあります. for に使われている変数や単純な四則演算が何度も行われる変数などの型を指定することで大抵は高速化できるようです.

型を指定するときには, int などで使える値の範囲を気にしなければなりません. もしとりうる値が大きくなるのであれば, int の代わりに, long int や long long int などを使う必要があるかもしれません.

メソッドの引数への型の指定は, cdef などをつけずにそのまま int などを指定するだけで行えます. 型の指定を行わなくても構いません.

```
%cython
def test6(int m):
    n=0
    for i in range(m):
        n=n+1
    return n
```

メソッドの返り値の型を, C 言語のように宣言することが出来ます. この場合も cdef を使います. ただし, メソッドの返り値の型を宣言した場合, 他のセルからそのメソッドを呼び出すことはできなくなりますので注意が必要です. もし他のセルから呼び出す必要があるのであれば, 返り値を宣言したメソッドを呼び出すためのメソッドを書いておけば一応解決します. 次のように定義すると test7(5000000) としても, エラーがでます. しかしながら, test8(5000000) は実行可能です.

```
%cython
cdef int test7(int m):
    cdef int n,i
    n=0
    for i in range(m):
        n=n+1
    return n

def test8(int m):
    return test7(m)
```

筆者のマシンでは 392 ナノ秒でした.

B.2.2 例 2

メソッドの返り値を宣言しておいたほうが良い例を挙げておきます. $a_i = (-1)^i$ という数列の和 $\sum_{i=0}^{m} a_i$ を計算することを考えましょう. 例えば, 次のように書くことが出来ます.

```
def a1(m):
    if m%2==0:
        return 1
    else:
        return -1

def sum1(m):
    n=0
    for i in range(m):
        n=n+a1(i)
    return n
```

このように定義し `timeit('sum1(5000000)')` とすると, 筆者のマシンでは 3.62 秒ということでした. これをまず, ただ `%cython` を書くことで高速化してみます.

```
%cython
def a2(m):
    if m%2==0:
        return 1
    else:
        return -1

def sum2(m):
    n=0
    for i in range(m):
        n=n+a2(i)
    return n
```

こうするだけで筆者のマシンでは 822 ミリ秒になりました. 次に, 使われている変数の型を指定することで高速化をします.

```
%cython
def a3(m):
    if m%2==0:
        return 1
```

```
    else:
        return -1

def sum3(m):
    cdef int n,i
    n=0
    for i in range(m):
        n=n+a3(i)
    return n
```

これで筆者のマシンでは 735 ミリ秒になりました. 一応は速くなっているのですが, 前の例に比べてそれほど速くなっていない印象を受けます. そこで, 引数の型の宣言をしてみます.

```
%cython
def a4(int m):
    if m%2==0:
        return 1
    else:
        return -1

def sum4(m):
    cdef int n,i
    n=0
    for i in range(m):
        n=n+a4(i)
    return n
```

これで筆者のマシンでは 476 ミリ秒になりました. この変更でなぜ高速になったかについて考えてみましょう. 1 つ前のものをみると, sum3 というメソッドの for ループの中から, a3 というメソッドが呼ばれます. いま i は cdef int と型が宣言されており, C 言語の int となっています. 一方で a3 の定義を見ると, 引数の型は宣言されていませんので, メソッドを呼び出すときに, i の値は一旦 Python の int に変換され, それからメソッド a3 が呼び出されます. メソッドの引数の方が宣言されていないばっかりに, 一旦型が変換されるという手間が入ってしまいます. 一方で a4 では定義の際に引数の型が宣言されていますので, 変換されることはないため高速になったのでした.

プログラムをよく見ると, a4 の返り値は型が宣言されていませんが n は型が宣言されています. したがって n=n+a を計算する際に, a の値の変換が起こることになります. もし n の型宣言をやめて型の変換が起こらないようにすると, 若干ですが速くなります.

```
%cython
```

```
def a5(int m):
    if m%2==0:
        return 1
    else:
        return -1

def sum5(m):
    cdef int i
    n=0
    for i in range(m):
        n=n+a5(i)
    return n
```

は筆者のマシンでは, 本当に少しだけ速くなり, 441 ミリ秒でした. しかしながら, もし型の変換が起こらないようにするのであれば, n の型宣言をやめるのではなく, メソッドの返り値の型を宣言するという方針も考えられます.

```
%cython
cdef int a6(int m):
    if m%2==0:
        return 1
    else:
        return -1

def sum6(m):
    cdef int n, i
    n=0
    for i in range(m):
        n=n+a6(i)
    return n
```

この方が劇的に速くなります. 筆者のマシンでは 38.4 ミリ秒となりました.

型が宣言されている変数とされていない変数を行き来するところがボトルネックになる可能性があるということはわかりました. 先ほどとは逆に, メソッドの返り値は型宣言されているが, 変数 n のほうは型宣言されていない場合を考えてみましょう.

```
%cython
cdef int a7(int m):
    if m%2==0:
        return 1
    else:
        return -1

def sum7(m):
    cdef int i
```

```
    n=0
    for i in range(m):
        n=n+a7(i)
    return n
```

この場合, 筆者のマシンでは 109 秒となりました. メソッドの定義の型宣言がされていないときよりも速くなっていることがわかります. このことをまとめると, ‘大雑把には, %cython と書いたセルの中で使うメソッドの型宣言はしておいたほうが無難’ ということになると思います.

注意 B.2.1. %cython と書いたセルの中でメソッドを定義するときにその定義の中からメソッドを呼び出すことがあると思います. 最初の例のように, sum1 の定義の中から a1 を呼び出している場合, a1 の定義はそのセルを実行した時点の定義が採用されます. 後から別のセルで a1 を別に定義しなおしたとしても, sum1 で使われる a1 は古い定義の a1 のままであることに注意してください.

注意 B.2.2. 他のセルの中で自分で定義したメソッドは, %cython と書いたセルからは基本的には呼び出せません.