

目次

第 1 章	Sympy とは何か？	3
1.1	Sympy とは？	6
1.2	Sympy でできること	7
1.3	本書の目標 ～Python を超高性能電卓として使いこなせ！～	14
第 2 章	Python の基本の基本	17
2.1	Python の環境構築 ～Google Colaboratory～	17
2.2	画面への文字表示	22
2.3	変数	23
2.4	データ構造（リスト、タプル、集合）	25

第 1 章

Sympy とは何か？

近頃、空前の機械学習ブームの到来などがキッカケとなって、基礎／応用数学を学ぶ人や、**Python** というプログラミング言語を使って計算、データ分析等を行う人が世の中で飛躍的に増加しました。

1.0.1 Python とは

Python^{*1}を使うと、従来のプログラミング言語では非常に面倒、あるいは難解だったような複雑な処理や計算が、とても簡単に直感的に行えます。筆者は 3 つくらいプログラミング言語を使えますが、Python の

「直感性」「面倒なことは考えなくて良いシンプルさ」はまさに特筆すべき利点と言えるでしょう。

1.0.2 数学を学ぶハードル (1) ～「突然の機械学習ブーム」の観点で～

そして、Python 需要と同時に高まったのが「**数学需要**」です。数学は、昔々から学校教育の中でずーっと扱われ続けていますが、「受験勉強のため

^{*1} 「ぱいそん」と読みます。

に仕方なく勉強するもの」というイメージが強く根付いていることが事実でした。しかし、近頃は突然の機械学習ブームを皮切りに「数学」という学問の重要性を誰もが認識するようになりました。だって、これだけ毎日毎日話題になっている「AI(人工知能)」を司る「機械学習」のアルゴリズムは、ほぼ完全に数学によって記述されているのですから。「AI」の原動力となる仕組みは、もっぱら数学だと言っても過言ではありません。

しかしながら、筆者がこの「AI ブーム」の最中、誰もが「数学」を学ぼうと試みる様子を見ながら強く感じたことは、「やはり数学の学習は鬼門なのだ」ということだったりします。というのも、無理はないのです。高校、あるいは大学を卒業して IT のお仕事をはじめて、結構な間数学とは遠い世界でお仕事をしていたところ、突然訪れてしまったこの「AI ブーム」の中で、世の中が「数学が重要だ！」「数学を勉強しないと！」なんてまたたく間に色めきだつのですから。戸惑うに決まっています。今まであんなに、「数学はお金にならない」「数学科は就職がないから工学部に進学したほうがいい」なんて、世間から完全に「食えない学問の代表格」扱いされまくっていたあの数学がですよ。

当然、数学なんてしばらく触っていない。そんな日々の中で、数学の知識もまるっと頭から抜けてしまった。でも、機械学習の本なんかを読んでみれば、「ここは活性化関数を微分して...」「ここでこのベクトルをこの行列で線形変換して...」「この分布の確率密度関数は...」なんて言葉が「知ってますでしょ？」みたいに列挙されています。ああ、一体どうすればいいんだろう...なんて時代に生まれてしまったんだ...なんてことを感じている方も多いかもしれません。

ここで一言皆さんにお伝えしておきます。そんな悩みを抱えている方には、「Sympy」*2を使うのがおすすめです (詳しくはまた後で)。

*2 「しむばい」と読みます。

1.0.3 数学を学ぶハードル (2) ～「学校での学び」の観点で～

中学校までの間に学ぶ「数学」と、高校で学ぶ「数学」、そして大学で学ぶ「数学」の間には、それぞれ結構な隔たりがあって、どれも独自の難しさと面白さを兼ね備えています。

その中でやはり最も皆さんにとって壁になりやすいのが、「式変形」*3です。中学校くらいまでの数学であれば、そんなに難しい式変形に出会うこともありませんが、高校、大学へと進んで行くと、そこで現れるのはいくつにでも連なる式変形の嵐。例えば、高校数学で「基本」と言われる操作、「二次関数の平方完成」ひとつとっても、式変形を丁寧に追いかければ以下のようになります*4。

$$\begin{aligned}y &= 2x^2 + 8x + 3 \\&= 2(x^2 + 4x) + 3 \\&= 2(x^2 + 4x + 4 - 4) + 3 \\&= 2(x^2 + 4x + 4) - 8 + 3 \\&= 2(x + 2)^2 - 5 \quad \square\end{aligned}$$

この式変形を手計算でゴリゴリと進める間、だんだんと頭の中がぼんやりとした不安で埋め尽くされてきます。「これ、計算合ってるんだろうか...?」そんな中で、数学を学び続けることが徐々にしんどくなってきて、結局のところ、「ああ、自分は数学には向いていないのかもしれない...」なんて、心がぼっきり折れてしまうなんてことも少なくありません。

ここで一言皆さんにお伝えしておきます。そんな悩みを抱えている方には、「SymPy」を使うのがおすすめです。

*3 数式をイコールで結びながら、違う形に変形すること。または、それを繰り返すこと。

*4 本書では、数式変形の終わりを □ 記号で表します。

1.1 Sympy とは？

こんなふうに、「数学」を学ぶことは、（もちろんその向こう側に楽しさはあれど）なかなかしんどい気持ちになる作業の連続だと感じてしまうことも少なくないものです。

こんな悩みと日々格闘されながら、それでも式変形とうんうん唸りながら向かい続ける方々を目の当たりにして、筆者もいろいろと考えました。どうかして、この負担をうまく軽くする方法はないものだろうか？ということ。そしてその結果、「現代人の超効率的数学勉強法」として、このやり方こそベストソリューションなのではないか！？という方法を見つけ出すに至りました。それが、

「面倒な式変形は Sympy にある程度投げてしまう」

というアプローチです。

Sympy とは、プログラミング言語「Python」上で動く、式変形などの数学的操作を、秒速かつ正確に行ってくれるモジュール^{*5}のひとつです。Python 上で Sympy を動かしてしまうことによって、我々が「面倒」だと思ふ式変形を（ほぼ）全自動で行うことができ、さらに、機械学習の専門書や学校の問題集の式変形の行間を埋めることも非常に容易になります。

ここで一つ断っておきたいこととして、「手計算のやり方なんて知らなくても数学を学ぶのに問題はない」ということを主張しているわけではありません。ただ、「毎回毎回、同じような面倒な式変形を手ですべて行う」必要は、この現代そんなにないのかもしれませんし、あるいは、「自分の手計算が正しいのかどうか」を確かめる術というのは、意外とそんなになかったりもするものです。こういう「数学を学ぶ上でぶち当たる面倒な問題」を、

^{*5} モジュールとは、簡単にいうと「プログラミング言語を構成する便利機能の集まり」のようなもののことを指します。

Sympy は「最新技術」の力である程度解決してくれるのです。

しかも、Sympy を使うことはとても簡単です。面倒な環境構築等もほとんど必要ありませんし、使い方も極めて明快で、誰もが簡単に、この Sympy の恩恵に預かることが出来ます。この「数学の重要性」が当たり前のように語られるになった世の中、こんなに便利なアプローチを使わない手はありません。

1.2 Sympy でできること

Sympy を使うと、以下のような数学的操作を簡単に行うことができます。ここでは詳細な説明を省き、簡単に実例を交えながらこれらの「便利機能」を俯瞰してみようことにしましょう。もちろん、数学的な理論の説明はまだしていないわけですから、皆さんは「おぉー、なんか知らないけどめっちゃ便利そうだな」くらいの気持ちで眺めていただければ問題ありません。

- 式の四則演算、展開、因数分解など、簡単な式の計算を行う。
- 初等関数 (多項式、指数関数、対数関数、三角関数など) を扱ったり、グラフを描画する。
- 方程式を解く。
- 微分、積分などの計算を行う。
- ベクトル、行列、行列式など、線形代数と呼ばれる分野の計算を行う。
- その他、複素数や組合せなどの計算を行う。
- 機械学習で必要になるような式変形の数々を行う。

1.2.1 簡単な式の計算

例題 1.2.1. 次の式を因数分解せよ。

$$x^3 + 8x^2 + 3xy + 24y$$

(i) 手計算による解法

次数が小さい y について、降べきの順に整理すると、

$$\begin{aligned} 3xy + 24y + x^3 + 8x^2 \\ = 3y(x + 8) + x^2(x + 8) \end{aligned}$$

共通因数 $(x + 8)$ でくくると、

$$(x + 8)(x^2 + 3y) \quad \square$$

(ii) SymPy による解法

```
[1] from sympy import *

x, y = symbols("x y")
factor(x**3+8*x**2+3*x*y+24*y)
```

☞ $(x + 8) * (x**2 + 3*y)$

特筆すべき点は、手計算による解法では、「解法」を人間が頭で考えて、そのとおりの変形を行って答えにたどり着いているのに対し、SymPy による解法では、「 x, y という文字を使いますよ」「この式を因数分解してください」と、「指示」を与えるだけで正しい答えが得られているところです。この2

つの解法は、「人間がウンウン唸りながら考える」というプロセスの有無という意味で、天と地の差くらいの面倒さの違いがあります。

1.2.2 初等関数を扱う

例題 1.2.2. 次の関数 $f(x)$ のグラフを描画せよ。

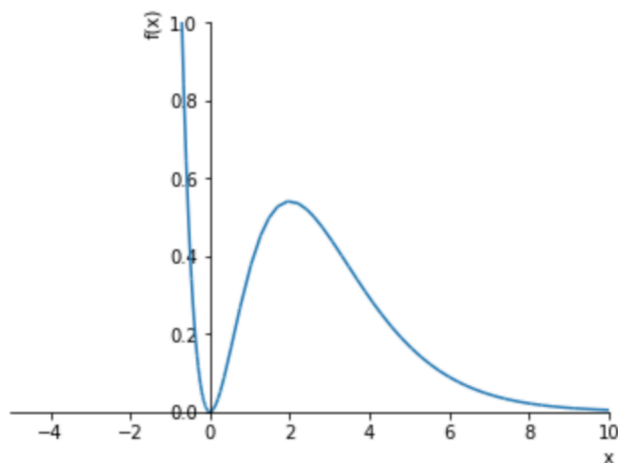
$$f(x) = x^2 e^{-x}$$

(i) 手計算による解法

関数 $f(x)$ の 1 階導関数、2 階導関数を計算し、関数の増減、凹凸を調べ、増減表を作成してそれをもとにグラフを描く。

(ii) Sympy による解法

```
[1] from sympy import *  
  
x = symbols("x")  
plot(x**2*exp(-x), (x,-5,10))
```



<sympy.plotting.plot.Plot at 0x7fd07518a160>

この例でも、手計算による「増減表」を用いた方法だと、かなりの大きな手間がかかります。しかし、Sympy を用いた方法は、「x という文字を使いますよ」「この関数のグラフ描いてくださいね」と指示を与えるだけで、非常に正確なグラフを描いてくれています。

1.2.3 方程式を解く

例題 1.2.3. 次の方程式の解を求めよ。

$$3x^3 + 2x^2 + x + 10 = 0$$

(i) 手計算による解法

多分これを手計算で解くには、以下の **3 次方程式の解の公式** ^{*6}を使うしかないでしょう (あるいは、センスのある人ならば**因数定理**で解くことも奇跡的にできるかもしれません)。

$$\begin{aligned}
 x &= \frac{\sqrt[3]{\sqrt{(-27a^2d + 9abc - 2b^3)^2 + 4(3ac - b^2)^3} - 27a^2d + 9abc - 2b^3}}{3\sqrt[3]{2}a} - \\
 &\quad \frac{\sqrt[3]{2}(3ac - b^2)}{3a} - \frac{b}{3a} \\
 x &= -\frac{1}{6\sqrt[3]{2}a}(1 - i\sqrt{3}) \\
 &\quad \frac{\sqrt[3]{\sqrt{(-27a^2d + 9abc - 2b^3)^2 + 4(3ac - b^2)^3} - 27a^2d + 9abc - 2b^3} + \\
 &\quad \frac{(1 + i\sqrt{3})(3ac - b^2)}{3 \times 2^{2/3}a\sqrt[3]{\sqrt{(-27a^2d + 9abc - 2b^3)^2 + 4(3ac - b^2)^3} - 27a^2d + 9abc - 2b^3}} - \frac{b}{3a} \\
 x &= -\frac{1}{6\sqrt[3]{2}a}(1 + i\sqrt{3}) \\
 &\quad \frac{\sqrt[3]{\sqrt{(-27a^2d + 9abc - 2b^3)^2 + 4(3ac - b^2)^3} - 27a^2d + 9abc - 2b^3} + \\
 &\quad \frac{(1 - i\sqrt{3})(3ac - b^2)}{3 \times 2^{2/3}a\sqrt[3]{\sqrt{(-27a^2d + 9abc - 2b^3)^2 + 4(3ac - b^2)^3} - 27a^2d + 9abc - 2b^3}} - \frac{b}{3a}
 \end{aligned}$$

^{*6} フェラーリの公式、カルダノの公式などと呼ばれることもあります。長過ぎるのでかなり小さく記載しました。

(ii) Sympy による解法

```

▶ from sympy import *
x = symbols("x")
solve(3*x**3+2*x**2+x+10)
--INSERT--
☞ [-5/3, 1/2 - sqrt(7)*I/2, 1/2 + sqrt(7)*I/2]

```

あらあら、手計算ではまあそう簡単には解けないだろうなという方程式でも、解が $x = -\frac{5}{3}, \frac{1}{2} - \frac{\sqrt{7}}{2}i, \frac{1}{2} + \frac{\sqrt{7}}{2}i$ であることが一瞬でわかってしまいました。

1.2.4 微分や積分の計算

例題 1.2.4. 次の関数 $f(x)$ を微分せよ。

$$f(x) = e^{ax} \sin bx$$

(i) 手計算による解法

積の微分法則を用いて計算する。

$$\begin{aligned}
 \frac{df}{dx} &= (e^{ax})' \sin bx + e^{ax} (\sin bx)' \\
 &= ae^{ax} \sin bx + be^{ax} \cos bx
 \end{aligned}$$

(ii) Sympy による解法

```
[1] from sympy import *  
  
a,b,x = symbols("a b x")  
diff(exp(a*x)*sin(b*x), x)  
  
↳ a*exp(a*x)*sin(b*x) + b*exp(a*x)*cos(b*x)
```

微分の計算を行うには、関数の形に合わせた微分公式や微分公式を熟知した上で選択できなければなりません。それを「知らなくても良い」というわけでは決してありませんが、このくらいの微分計算なら、Sympy でやったほうが簡単で正確ですね。

1.2.5 ベクトル、行列、行列式に関する計算

例題 1.2.5. 次の行列 X の逆行列を求めよ。ただし、 a は定数である。

$$X = \begin{pmatrix} 1 & -a & 0 \\ 0 & 1 & -a \\ 0 & 0 & 1 \end{pmatrix}$$

(i) 手計算による解法

3 次逆行列の公式、または掃き出し法（ガウスの消去法）などを使って求めます（どちらも結構めんどくさいのでここに詳細を書く必要もないかと思います。気になる方は線形代数の教科書読んでみてください）。

(ii) Sympy による解法

```
[1] from sympy import *

a = symbols("a")
A = Matrix([[1,-a,0],
            [0,1,-a],
            [0,0,1]])

A**(-1)

↳ Matrix([
  [1, a, a**2],
  [0, 1,  a],
  [0, 0,  1]])
```

この結果のすごいところは、 x が文字 (a) が含まれる行列であるにも関わらず、何も言わずにさっと逆行列を計算してくれているところです。このような「文字を含む式変形などの計算」のことを「代数的計算」と呼び、それに対して「数字だけの計算」を「数值的計算」と呼びます。プログラミング言語というのは、本来「数值的計算」は得意ですが、「代数的計算」は不得手なのです。しかし、SymPy はいとも簡単にこのくらいの計算は正確に行ってくれます。

1.3 本書の目標 ～Python を超高性能電卓として使いこなせ！～

Sympy を使うと、様々な面倒な計算がいとも簡単に、しかも瞬速&正確に行なえることを何となく、感じて頂けましたか。Python 上で使えるモジュール（ライブラリ）は何も SymPy に限るわけではなく、他にも非常に優秀な機能を Python 上でゴリゴリ動かし、人間の手助けをしてもらうことが十分に可能です。

しかし、この SymPy というモジュールひとつだけをとっても、この威力は計り知れないと思っています。なんと言っても「代数的計算」を非常に気軽に行えること、そして、使い方がとても簡単で、それでいて高機能であることなど、利点を挙げれば正直キリがありません。他のモジュールなどは、

非常に便利ではあるが、使用方法が難解、あるいは高度な計算に特化していて数学入門の段階では使いにくいなど、やや我々にとっては不都合な性質を持っていることも多いのです。

本書では、この SymPy に焦点を絞って（他のモジュールは思い切って見なかったことにして）、様々な数学的操作がこの SymPy で驚くほど自在に行えることを体感し、皆さんが Python を「超高性能電卓」として使いこなし、数学の学習や実務での応用における非常に強力な武器として振りかざしていただける、という状態までたどり着くことを目標とします。

何度もいいますが、SymPy の使い方は驚くほど簡単です。そして強力です。そしてこの SymPy を使いこなして習得する「数学の力」は、我々の業務や学習、さらには世界の見え方まで変えてしまう強大な力を秘めています。ぜひ、皆様と一緒にそこまでたどり着けたら、筆者としては光栄に思います。

それでは、早速本論を始めていきましょうか。SymPy の世界へようこそ！

第 2 章

Python の基本の基本

Sympy は、Python 上で動くモジュールの一つです。すなわち、Python の基本的な使い方を知らなければ、Sympy もあまり自由に使いこなすことができません。この章では、「全く何も分からない」状態から、Python の「基本の基本」を身につけることを目標にしましょう。

2.1 Python の環境構築 ～Google Colaboratory～

まずは、お手元の PC で Python が動作する環境を整えなければお話は始まりません。というわけで、早速環境構築を... という前に、少しだけお話を。

実はちょっと昔ですと、お手元の PC の中に Python の環境を構築するために、なかなか面倒な手順が必要でした。様々なアプリケーションをダウンロードして、これとこれとこれをインストールして、いざ動かしてみるとあれ？なんかエラーでうまく動かない... こんな格闘を繰り返るのが、通過儀礼になっていた時代もあったのです。なんなら、Python を一度も触ることなく、この環境構築で挫折してしまう、なんて人も...

でもこんな、最初の一步で心が折れてしまうのってあまりにも悲しいじゃないですか。そんな悲痛な叫びを察知してくれたのか、現在は Google

のサービス「Google Colaboratory」を用いると、ものの数分で環境構築が誰でも簡単に、完璧に終わってしまいます。本書ではこの「Google Colaboratory」を使って、さくっと環境構築を済ませてしまうことにしましょう。

手順1. Google アカウントを取得する

まず、Google アカウントを取得してください（この手順はここでは解説しませんが、やりかたが分からない方は検索すればすぐにわかります）。

手順2. Google drive を開く

作った Google アカウントにログインし、Google drive を開きます。



図 2.1 Google drive を最初に開くとこんな感じ（何もファイルは入っていない）。

手順3. フォルダを作る

作ったプログラムを実際に格納し、動かすためのフォルダを作りましょう。右クリック → 新しいフォルダを選択します（ここでは、とりあえずフォルダ名は「test」にでもしておきましょう）。

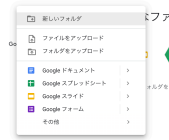


図 2.2 右クリック→新しいフォルダ。



図 2.3 フォルダ名はとりあえず「test」に。

手順 4. フォルダの中に入り、GoogleColaboratory をインストール（有効化）。

作ったフォルダ「test」の中に入りましょう。



図 2.4 ダブルクリックで入れます。

そして、GoogleColaboratory を使える状態にします。右クリック→その他→アプリの追加 をクリックします。

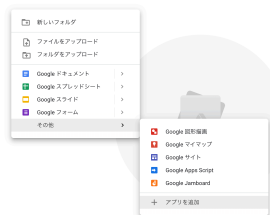


図 2.5 右クリック→その他→アプリの追加。

「GoogleColaboratory」を探して、インストールを行います。

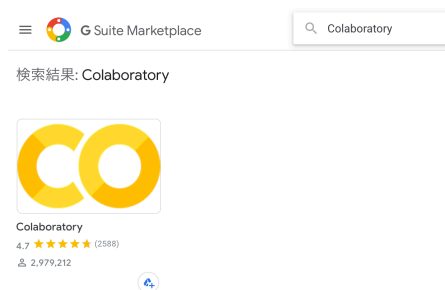


図 2.6 検索窓に「Colaboratory」と入れると出てくるはず。これをクリック。



図 2.7 インストールボタンを押すと、インストールがはじまる。

手順 5. .ipynb ファイルを作成。

GoogleColaboratory 上では、プログラムを.ipynb 形式で扱います。.ipynb ファイルを「test」フォルダの中に作ってみましょう。

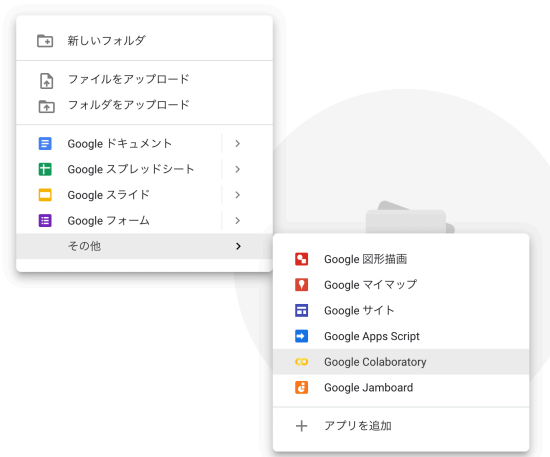


図 2.8 右クリック→「その他」に、「GoogleColaboratory」が追加されているので、クリックして.ipynb ファイルを作る。



図 2.9 こんな画面が表示されれば準備完了！

なんと、Python の環境構築はこれでほぼ終了です。あとは、フォルダ内に必要に応じて.ipynb ファイルを作成し、様々なプログラムを作成／実行して行きます。



図 2.10 「test」フォルダには、ちゃんと.ipynb ファイルが作成されています。

2.2 画面への文字表示

いよいよ Python のプログラムを作成／実行する準備が整ったところで、簡単なプログラムを動かしてみましょう。以下のソースコード^{*1}を「一言一句違わず」「半角文字で」入力し、再生ボタンをクリック^{*2}して実行します。

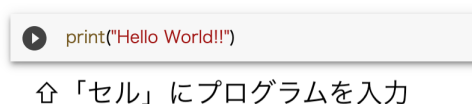


図 2.11 「セル」にプログラムを入力し、左の再生ボタンを押して実行。

以下のように「Hello World!!」と表示されれば、はじめての Python プログラムが動作したことになります。

このように、**print** という命令を使うと、画面に好きな文字列を表示することができます^{*3}。

^{*1} プログラムの処理内容を記述した命令の集まりのこと。

^{*2} Windows では「Ctrl+Enter」、Mac では「Command+Enter」を押しても OK（慣れるとこちらのほうが楽）。

^{*3} 文字列をダブルクォーテーションで囲むのを忘れずに！あと、表示したい文字列は全角

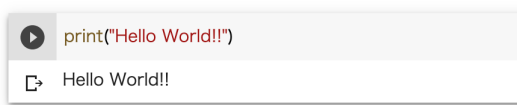


図 2.12 「Hello World!!」という文字列が表示されました。

画面に文字列を表示

```
print("表示したい文字列")
```

さて、本格的に Python のプログラムを動かす準備が整いました。というわけで、ここからは基本的な Python のプログラムの書き方を学んでいきましょう。Python は、「プログラムが非常に簡単に書ける」言語として有名で、やりたい処理を直感的に、サクサクと記述して動かすことができます。

2.3 変数

Python をはじめとする様々なプログラミング言語では、**変数 (variable)** という「数値や文字列をしまっておく箱」に様々な値を格納しておいて、その値を必要なときに利用することによっていろいろな計算や処理を効率的に行います。先程の.ipynb ファイルの新しいセルで、`x` という変数を宣言^{*4}し、`1` という値を代入^{*5}してみましょう。また、その次の行で、`x` にちゃんと `1` が代入されていることを確認するために、`print` を使って `x` を表示してみましょう^{*6}。

文字でも OK ですが、それ以外のソースコードはすべて半角で書きましょう。

*4 変数を新しく作ること。

*5 変数に値をしまうこと。

*6 変数の「中身」を表示したいときは、ダブルクォーテーションでは囲まないことに注意。

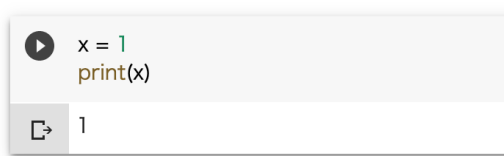


図 2.13 「1」が表示されているので、正しく x に 1 が代入されていることがわかります。

変数には、1 のような整数だけではなく、3.14 のような**実数** (小数点以下がある数) や、"こんにちは" のような**文字列** も代入しておくことが出来ます。また、変数の名前 (変数名) も自由に設定できます^{*7}。

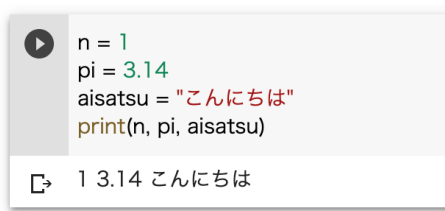


図 2.14 変数 n, pi, aisatsu を宣言し、それぞれに 1, 3,14, "こんにちは" を代入して表示してみました。

複数の変数などを並べて表示するときは、このようにカンマ (,) で区切れば表示できます。

^{*7} 「自由に」とはいつても、何でもかんでも OK というわけではなくある程度のルールがあります。とりあえず、「アルファベットの大文字小文字」と「数字」が使えて、「アンダースコア (_)」も使える、また、「1 文字目を数字で始めてはいけない」「同じ変数名を複数回使うことはできない」「Python ですでに使われている名前 (予約語) はつけられない (`print` など) 」というルールだけを知っておけば問題は生じないでしょう。ちなみに、一応全角文字も使えますが、使わないほうがいろいろと無難です。

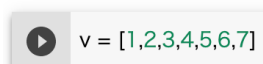
変数を宣言し、値を代入

変数名 = 代入する値

2.4 データ構造（リスト、タプル、集合）

たくさんの数値をまとめて利用したいときに、変数を1つずつ頑張ってたくさん宣言して、それぞれに値をちまちまと代入して利用するのは、なかなか手間がかかって効率が良くありません。「複数の値を一気に効率よく扱う」ためには、**データ構造**を利用します。

データ構造とは、複数の値（数値や文字列）を「ひとまとまりに扱う」ための方法のことをいいます。最も基本的なデータ構造は**リスト**です。リストとは、複数の数値を並べてひとまとまりに宣言したものです*8。以下のコードを実行してみましょう。



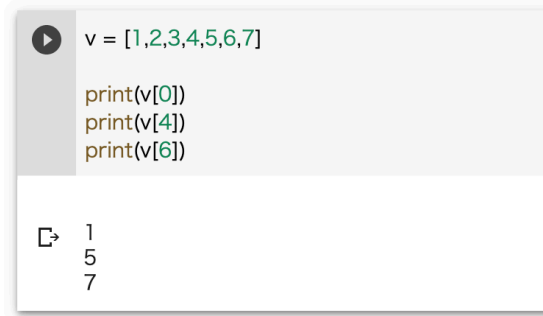
```
v = [1,2,3,4,5,6,7]
```

図 2.15 リスト v を宣言するソースコード。

このコードでは、v というリストに 1,2,3,4,5,6,7 という 7 つの数値を順に格納してあります。このようにリストに格納された値のことを、リストの**要素**と呼びます。リストの要素には、0 番目から順に番号（インデックス）づけされており、以下のように v の要素を読み出して利用することができます。

リストの要素の値を上書きすることもできます。例えば、先程の v で、v[3] の値を 4 から 100 に書き換えたいときは、以下のような代入を行えば OK です。

*8 他の言語なんかだと、「配列」というものが現れることが多いですが、リストと配列はだいたい同じようなものです。



```
v = [1,2,3,4,5,6,7]

print(v[0])
print(v[4])
print(v[6])
```

1
5
7

図 2.16 リスト `v` の要素 `v[0]`, `v[4]`, `v[6]` を `print` 文で画面に表示しています。



```
v = [1,2,3,4,5,6,7]

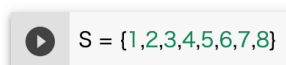
print("書き換え前", v[3])
v[3] = 100
print("書き換え後", v[3])
```

書き換え前 4
書き換え後 100

図 2.17 リスト `v` の要素 `v[3]` を 100 に書き換えました（`print` 文で都度、値を確認しています）。

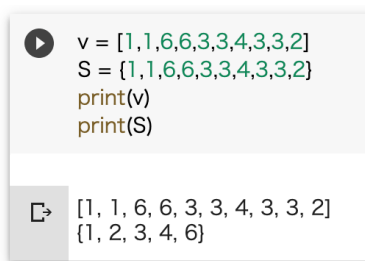
ちなみに、リストは `v=[1,2,3,4,5,6,7]` というふうに宣言をしましたが、`v=(1,2,3,4,5,6,7)` というふうに宣言をした場合、**タプル**というデータ構造になります。タプルはリストとほぼ同じように扱えますが、要素の書き換え（代入）ができません。

次に、以下のように新しいセルで `S` を宣言してみましょう。
このように宣言した `S` を**集合 (set)** と呼びます。集合とリストは何が違うのか？以下のソースコードで比較をしてみましょう。



```
S = {1,2,3,4,5,6,7,8}
```

図 2.18 S を宣言。



```
v = [1,1,6,6,3,3,4,3,3,2]
S = {1,1,6,6,3,3,4,3,3,2}
print(v)
print(S)
```

```
[1, 1, 6, 6, 3, 3, 4, 3, 3, 2]
{1, 2, 3, 4, 6}
```

図 2.19 リスト v と集合 S を比較。

実行結果を見ると、以下の違いに気づきます。

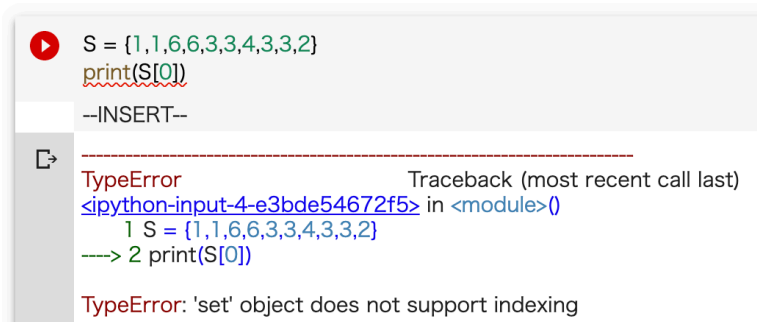
- リスト v には重複する要素が残っているのに対し、集合 S では、要素の重複が無視されている。
- リスト v は、要素の順番が保持されているのに対し、集合 S は要素の順番が変わっている。

このように、集合というデータ構造は、要素の重複と順序を無視して数値を保持することが分かります。ですので、例えば以下のような要素の読み出しは出来ません *9。

このように、あくまで「含まれるか含まれないか」だけを問題とするデータ構造である「集合」は、高校数学で習う集合まさにそのものです。

リスト、タプル、集合についての細かい操作（例えば、要素の追加や削除

*9 S は「要素の順番」がないので、「0 番目」を呼び出すことがそもそもできない。



```
S = {1,1,6,6,3,3,4,3,3,2}
print(S[0])
--INSERT--
```

↳

```
-----
TypeError                                Traceback (most recent call last)
<ipython-input-4-e3bde54672f5> in <module>()
      1 S = {1,1,6,6,3,3,4,3,3,2}
----> 2 print(S[0])

TypeError: 'set' object does not support indexing
```

図 2.20 S[0] を読みだそうとするとエラーになる。

など) については、他の Python 入門書に解説を譲ることとします。