1 はじめに

このファイルではpython3における変数の扱い方と型について見ていきます.

2 目次

3 変数について

変数とは

pythonに限らず, 多くのプログラミング言語では `変数` というものが存在します.
数学で利用する変数と同じような違うような…って感じです.

とりあえずソースプログラムを書いて覚えましょう.
以下のプログラムを見てください.

```
a = 10

b = 20


print(a + b) # a と b を足す

print(a * b) # a と b を掛ける
```

このソースプログラムにおける変数は `a` と `b` です. `=` を使うと左側の変数へ何かしらの値(数字とは限らない)をいれることができます.
ちなみにここでの `=` は= `代入演算子` といいます. 数学では `=` は等しいことを表しますが, プログラミングでは代入を意味します.

また, 変数へ値を再代入することもできます.

```
a = 10
print(a)


a = 20
print(a)
```

変数の参照先

ここの部分は少しむずかしい内容なので, 読まなくても大丈夫です.
読みたい方だけ読んでください.

実はC言語などと違い, 再代入すると, 参照先が変わります.
そして変数から変数へ代入すると参照先がコピーされます.

```
a = 10
print("value:{}, id:{}".format(a, id(a)))


a = 20
print("value:{}, id:{}".format(a, id(a)))


b = a
print("value:{}, id:{}".format(b, id(b)))
```

ちなみにC言語では

```
#include <stdio.h>


void main(){
        int a = 10;
        printf("value:%d, adress:%p\n", a, &a);


        a = 20;
```

```
        printf("value:%d, adress:%p\n", a, &a);

        int b = a;
        printf("value:%d, adress:%p\n", b, &b);
}
```

```
value:10, adress:0x7ffd31913620
value:20, adress:0x7ffd31913620
value:20, adress:0x7ffd31913624
```

変数の代入/再代入

先ほどやったとおりです.

変数から変数への代入

作成した変数を違う変数へ代入演算子 `=` を利用するとどうなるでしょうか.

```
a = 10
b = a
print(a, b)
```

変数 `a` の中の値がコピーされ, `b` へ入りました!
…と思いきや, 少し違います.
pythonでは変数から変数へコピーすると `参照先` がコピーされます.

```
a = 10
b = a
print(a, id(a))
print(b, id(b))
```

ただの変数のときは値がコピーされているという認識でも大丈夫ですが, 今後出てくる操作で間違いが起こらないように, `参照先がコピーされている` と覚えていた方が良いと思います.

あとで図を載せます

ただの変数の演算であれば, もとの変数を再代入するとそちらの変数の参照先が変わるため, あまり気に病む必要はありません.

```
a = 10
b = a
print(a, id(a))
print(b, id(b))

print()

a = 20
print(a, id(a))
print(b, id(b))
```

4 型

pythonでは型というものをあまり気にする必要がないのですが, 型を理解しておくと後々好都合なので説明しておきます.

以下のようなプログラムを実行すると

```
a = 20
b = "hoge"

print(a + b)
```

以下のようなエラーが出ると思います.

```
TypeError: unsupported operand type(s) for +: 'int' and 'str'
```

エラーの内容は「`+` では'int'と'str'という型同士の演算がサポートされてないよ」的なことです.
当たり前といえば当たり前ですが, `20` という数値と `"hoge"` という文字列は足し算できません.

このようにデータには `型` というものがあり, データ処理の整合性を確かめるのに役立ちます.

代表的な型

以下に代表的な型をあげていきます. 自分が知らないものや, 今のところ知らなくて良いものは省いています.

数値型

数値型にはさらに `int` 型, `float` 型, `complex` 型があります.
それぞれ整数, 浮動小数点数, 複素数が扱えます.

型名	内容	フルスペル
int	整数	integer
float	浮動小数点数	floating point number
complex	複素数	complex number

ブール値

2つの定数オブジェクト `False` と `True` からなる.
真理値を表すのに用いられます.

イテレータ型

あとで

シーケンス型

シーケンス型としては `list` , `tuple` , `range` があげられます.

テキストシーケンス型

文字列は `str` オブジェクトとして扱われます.
(Unicodeコードポイントのイミュータブルなシーケンス)

マッピング型

ハッシュ可能な値を任意のオブジェクトに対応付ける.
現在は辞書のみ.

集合型

ハッシュ可能なオブジェクトの重複なし, 順序なしコレクション.

- ミュータブル → set
- イミュータブル → frozenset

の2つがある.

module

あとで

クラスおよびクラスインスタンス

あとで

関数

あとで

メソッド

あとで

いろんな型を使ってみる

数値型

int

整数の型. 2/8/10/16進法で表すことができる.

```
dec = 10;        print(dec.__class__.__name__+" :", dec)    # 10進数
octal = 0o17;    print(octal.__class__.__name__+" :", octal)  #  8進数
```

```
binary = 0b101;  print(binary.__class__.__name__+" :", binary) #  2進数
hex = 0x1a;      print(hex.__class__.__name__+" :", hex)    # 16進数
```

Python2系では, intには表現の最小/最大値が存在したが, Python3では最小/最大は存在せず, どこまでも大きい数を表現できる.
(PCのメモリを全部使うより大きい数はもちろん無理.)

float

浮動小数点数という方式で数を表現する.
精度によってどこまで細かく数値を表せられるかが決まる.

```
a = 10.33;      print(a.__class__.__name__+" :", a)
b = 2.2e3;      print(b.__class__.__name__+" :", b) # 10のn乗を en と書ける
c = 1.;         print(c.__class__.__name__+" :", c)

# 表現限界
d = 1.9999999999999999; print(d.__class__.__name__+" :", d)
```

小数はこのように誤差が生まれることを前提にプログラムを組むのがベター.
例えば, 以下のように `aとb` が等しくないのに等しいと判断される場合がある.
(プログラムの内容はまだ習っていない部分が多くあるので, そういうこともあるんだなってくらいで大丈夫です.)

```
a = 1.99999999999999999
b = 2.0

print( a == b )
```

complex

Pythonではデフォで複素数を扱える.
虚数単位は数値の後ろに `j` をつけて表現する.

```
a = 1 + 1j;     print(a.__class__.__name__+" :", a)
b = 2.2 + 3.1j; print(b.__class__.__name__+" :", b)
```

また, 各複素数の実部と虚部は `num.real` , `num.imag` のように取り出せる.
取り出した各数字は `float` 型として扱われる.

```
a = 1 + 1j;
print(a.real.__class__.__name__+" :", a.real)
print(a.imag.__class__.__name__+" :", a.imag)

b = 2.2 + 3.1j;
print(b.real.__class__.__name__+" :", b.real)
print(b.imag.__class__.__name__+" :", b.imag)
```

文字列

Pythonでは `''` か `""` で囲うと文字列として扱われる.
この二つの記号間に機能的な差異はないが, 文中に `'` か `"` を使いたいときは, その逆の文字列を使用するとよい.

```
str1 = "hello, world!"; print(str1)
str2 = 'こんにちは, 世界!';print(str2)

str3 = "文中に「'」を使う.";    print(str3)
str4 = '文中に「"」を使う.';    print(str4)
```

文字列中で改行などを行いたい時の特殊な文字列も埋め込める.

```
str1 = "こんにちは\nよろしくお願いします"
print(str1)
```

なお, 複数行の改行であれば, ヒアドキュメントを利用するのがとても便利.
`""" ~ """` のように3つで囲む.

```
str1 = """おはよう
こんにちは

```

```
こんばんは"""
print(str1)
```

ブール値

演算結果の真偽などを示すのにもっぱら使われる型.
演算結果が正しい(真のとき)は `True` , 正しくないとき(偽のとき)は `False` となる.

```
print( 1 == 2 )
print( True == False )
print( True == True )


a = True
b = False
print(a & b)
```

(後々, 演算子の項目のところでより深く言及するので)
ここのプログラム中の演算はあまり気にしなくていいです.

その他

シーケンス型とマッピング型はまたあとで詳しく触れます.
関数, メソッド, クラスなどは後々それとなく言及します.
集合型もできれば説明したいけど, すごい特殊な用途でしか使ったことない…