# オブジェクト指向プログラミング

第３回

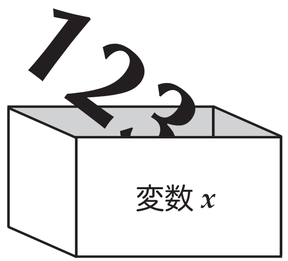
箕原辰夫

# 文とブロック

- 文
  - 式
- ブロック
  - 複数の文をまとめる
  - Pythonでは、空白あるいはTabによるインデントがあっているとブロックと見做される

# 変数

- 値を一時的に入れておく箱と思えば良い。




- 参照する前に、値を入れておかなければならない。
  - 入っていないと、文法チェック時か実行時にエラーが出る。

# 変数の名前と代入

- 変数の名前
  - 変数の名前に使えるのは、半角の英数字、およびアンダーバー（ _ ）
  - 名前の先頭の文字は英字でなければならない
  - 変数名は、半角の小文字の英字でお願いします

  例：

  $x$　$y$　$z$　*value67*

  複合語の場合：　*orange_juice*　　*orangeJuice*　　*orangejuice*　　*OrangeJuice*
- 代入（代入文）
  - 変数名 = 式

  例：　　$x = 10$

# プログラム上に現れる英字

- a … 変数名
- 'a' … a一文字から成る文字列
- "a" …a一文字から成る文字列
- 0xa … 16進数のa

- abc1234 … 変数名
- "abc1234" あるいは 'abc1234' … 文字列

- 以下は予約語なので、変数名としては使えない

  and as assert async await break class continue def del elif else except finally for from global if is import in lambda nonlocal not or pass raise return try yield while with

# 変数の参照

- 参照
  - 変数が保持する値に置き換えられる。
  - 変数が保持するオブジェクトが参照される。

- 自己参照代入
  - 元の値を利用して、新しい値が代入される。

$x = x + 1$

$x = -x$

$x = x - 20$

# 代入演算子としての=

- 右辺と左辺は同一ではない
- 等しいという意味ではなく、右辺の式の評価値を左辺の変数にAssignするもの。

左辺 = 右辺

✳この意味は、「左辺の変数　←　右辺の式の評価値」

- なお、代入演算子を複数記述することができる（右結合性）

$x = y = z = 0$ → ( x := ( y := ( z := 0 ) ) )
代入文　　代入式

# 式に何が書けるか

- 式の定義
  - 定数
  - 変数名
  - 式 + 式　←加算 add
  - 式 – 式　←減算 subtract
  - 式 * 式　←乗算 multiply
  - 式 / 式　←実数除算 true divide
  - 式 // 式　←整数除算 floor divide
  - 式 % 式　←剰余 modulo
  - 式 ** 式　←べき乗 power
  - 変数名 := 式　←代入式 assign expression
  - ( 式 )　←優先順位を上げる

# 式の構文

- ４５＊（３４＋２３）／（ｙ−５）を解釈する
- 式＊（式　+　式）／（式−式）
- 式＊（式）　／　（式）
- 式＊式　／　式
- 式　／　式
- 式

- ×　45x + 65y　は構文規則に合わないのでエラー

# 代入演算子 :=

- Python 3.8からの導入で、式の中に演算子として代入演算子を指定することができる
- = は、代入文を形成するので、左辺に代入する変数、右辺に式を記述するが、:=は式の中に代入式を記述することができる
- 例：

  print( n := 10,  n * 32  )　→　「10 320」が表示される

- :=は、優先度が低いので、場合によっては、()をつけて優先度を高くする必要がある
- また、関数呼出しの実引数の部分以外では「(変数名 := 式)」という形で、丸括弧をつけてあげないと文法エラーになる
- 例：

  w = 78 * ( u := 52 ) + 93
  ( n := 12, m := 13 )  # これはタプル

# 式の評価

- 評価（Evaluation）とは
  - 単一の値になるまで計算すること
- 式の書式に合っているか
  - 書式に合っていないと文法エラー

- 式の評価の優先順位
  - 優先順位がある
  - べき乗の演算子（**）の優先度は高い
  - 単項の±は次に優先される
  - 乗除算の演算子（* / // % )の方が優先される
  - 加減算の演算子（+ - )が優先度低い
  - 代入演算子（:=）は、加減算の演算子より優先度が低い
  - （）で囲むと優先度を高くする

# 結合性

- 同じ優先順位の演算子は、左から評価されていく（加減乗除などの場合）
  - 左結合性（Left associative）と呼ぶ

56 * 34 / 28 * 60 / 30 % 89

→( ( ( ( 56 * 34 ) / 28 ) * 60 ) / 30 ) % 89

83 + 45 - 23 + 38

→( ( 83 + 45 ) - 23 ) + 38

# 整数演算

- 整数除算は、小数点以下が切り捨てられる（実数でも可）
  - 5//2 → 2
  - 1//8 → 0　分母の方が大きいと0になる
- どこに整数除算があるか重要
  - 5 // 2 * 2 → 4
  - 5 * 2 // 2 → 5
- 剰余算は、余りを計算する（実数でも可）
  - 365 % 20 → 5
  - $x$ % $n$ → 0 ～ $n$-1の数しか出てこない

    計算結果が0になるときは、割り切れるということ

# 整数剰余・整数除算

- x // n * n
  - xと等しいか、xを超えない最大の数で、nで割り切れる数が求まる
  - 例： 10 // 3 * 3 → 9

- n // m
  - n < mの場合は、0になる
  - 例： 3 // 4 → 0

- n % m
  - n < mの場合は、nになる
  - 例： 3 % 4 → 3

# 基数と整数剰余・整数除算

- 各桁に分解できる
  - 3456 % 10 = 6
  - 3456 // 10 % 10 = 5
  - 3456 // 10 // 10 % 10 = 4
  - 3456 // 10 // 10 // 10 % 10 = 3
- n進数の各桁にも分解できる
  - 67 % 7 = 4
  - 67 // 7 % 7 = 2
  - 67 // 7 // 7 % 7 = 1

# 整数除算の計算方法

- x % n → x – ( x // n * n )
  - 例：

    35 % 8 → 35 - ( 35//8 * 8 )
    → 35 - (4*8) → 35 - 32 → 3

- x // n → ( x – x % n ) // n
  - 例：

    45 // 13 → ( 45 - 45 % 13 ) // 13
    → (45 - 6) // 13 → 39 // 13 → 3

- x // n * n ≠ xのときがある
  - 9 // 3 * 3 → 9
  - 10 // 3 * 3 → 9
  - 11 // 3 * 3 → 9
  - 12 // 3 * 3 → 12
  - 13 // 3 * 3 → 12
  - 14 // 3 * 3 → 12
  - 15 // 3 * 3 → 15
  - 16 // 3 * 3 → 15
  - 17 // 3 * 3 → 15

# 実数における整数除算・剰余の計算

- 整数除算（Floor division）は、実数でも使うことができる。ただし、計算結果は実数になる
  - 例： 4.7 // 2.1 → 2.0
  - 7.85 // 1 → 7.0
  - 56 // 0.856 → 65.0

- 実数での剰余には誤差が入るので注意すること
  - 例：4 % 0.7 → 0.5000000000000002
  - 1.2 % 0.4 → 0.3999999999999999

# 負の数を伴う整数除算・剰余

- どちらかに（あるいは両方）負の数がはいる場合は、もともと整数除算は、床関数（floor関数：⌊x⌋…xと等しいか、xよりも小さいなかでの最大の整数）で計算されるので、マイナス方向に引っ張られた値になる。
- 例：
  - 10 // -3　⇒　10/-3 = -3.333..., floor(-3.3...) = -4
  - -10 // -3　⇒　-10/-3 = 3.333..., floor(3.3...) = 3
  - -10 // 4　⇒　-10/4 = -2.5, floor(-2.5) = -3

- 負の値が入った場合の剰余の計算方法は、除数で整数除算を行なった結果と除数を乗算して、被除数からの差分が計算される
  - m % -n　⇒　m - (m // -n) * -n
- 例：
  - 12 % -7　⇒　12 // -7 = -2, 12 - (- 2 * -7) = 12 - 14 = -2
  - -4 % 12　⇒　-4 // 12 = -1, -4 - (-1 * 12) = -4 + 12 = 8
  - -8 % -5　⇒　-8 // -5 = 1, -8 - (1 * -5) * 1 = -8 + 5 = -3

- 参考：

  https://stackoverflow.com/questions/3883004/the-modulo-operation-on-negative-numbers-in-python

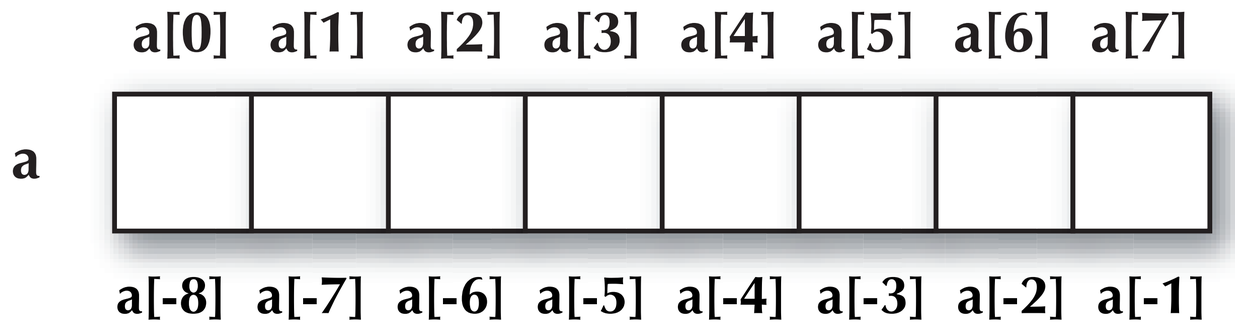
# PythonのIDLE

- インタープリタ（>>>が出ている）の画面では、実行履歴を使って表示することができる
  - control + p ： 1 つ前の履歴を出す (alt+p)
  - control + n ： 履歴の 1 つ後に進む (alt+n)

- IDLEのその他の編集キー
  - 左右の矢印キー：挿入カーソルを移動
  - control + a ： 挿入カーソルを最初に移動(alt+a)
  - control + e ： 挿入カーソルを最後に移動(alt+e)
  - control + k ： 挿入カーソルから後を削除(alt+k)
  - control + d ： 挿入カーソルの直後 1 文字を削除

# リスト・タプル・文字列の要素へのアクセス

- リスト・タプル・文字列では、各データを参照するための部屋番号がついている。これをインデックス（指標・添え字）と呼ぶ。




- インデックスは、「0〜サイズ-1」の範囲の整数に限られる。あるいは、後ろからアクセスする場合は、マイナスの値で「-1〜-サイズ」の範囲になる。範囲外だと実行時エラー（例外）が発生する。

# リスト・タプル・文字列のインデックス式

- リスト・タプル・文字列の長さを求める
  - len( s )　# sに代入されている要素の個数あるいは文字列の場合は長さを返す
- リスト・タプルから要素、あるいは文字列の中から 1 文字を取り出す
  - 書式：対象[ インデックス ]
  - インデックスは整数式で、範囲は0〜文字列・リスト・タプルの長さ-1
  - インデックスにマイナスを使うと最後から数える
  - マイナスの場合の範囲は、-1〜-文字列の長さ・-リストの長さ・-タプルの長さ
- 記述例：
  - s[ 0 ]　　s[ 17 ]　　s[ 7 ]　　s[ len( s ) - 2 ] ⇒ s[ -2 ]
  - s[ -1 ]　　s[ -6 ]　　s[ - len( s ) ] ⇒ s[ 0 ]

# リスト・タプル・文字列のスライスの記法

- 書式：対象[ 最初 : 終わりの次 ]
  - 部分的なリスト・タプル・文字列が新たに生成される
  - 例：alist[ 4: 7 ]

    aTuple[ -5: -1 ]
- 対象[ : 終わりの次 ]
  - 最初の要素の位置は、0と仮定される
  - 例：alist[ : 7 ] ⇒ alist[ 0:7 ]
- 対象[ 最初: ]
  - 指定された最初の位置から、最後までになる
  - 例：alist[ 4: ] ⇒ alist[ 4: len(alist) ]
- 対象[ -インデックス: ]
  - 最後の要素の次から、順次インデックスの値が引かれていく
  - 例：alist[ -5 ],  alist[ -5: ]

# 文字列のスライス式

- 文字列の中から範囲を指定して一定の範囲の文字列を取り出す
- スライスの記法は、以下の通り、インデックスを用いる
  - 文字列[ 始め:終わった次 ]
  - 文字列[ 始め:終わった次:間隔 ]
- 例：
  - s[ 0: 5 ]　# 最初の 5 文字
  - s[ -5: -1 ]　# 最後の 5 文字目から 4 文字分
  - s[ 7: 12: 2 ]　# 1 文字飛ばしつつ
- スライスの場合省略が可能
  - s[ : 5 ]　# 0からと仮定される
  - s[ -5 : ]　# 最後まで
  - s[ : ] # 初めから最後まで
  - s[ : : 2 ] # 初めから最後までで 1 つ飛ばし
  - s[ : : -1 ] # 逆順になる
  - s[ : : -2 ] # 最後から始めまで 1 つ飛ばし

# 文字列のインデックス式・スライス式を使った例

- s = "いろはにほへとちりぬるを"
  - s[ 0 ] ⇒ "い"
  - s[ -1 ] ⇒ "を"
  - s[ 1 ] ⇒ "ろ"
  - s[ 5 ] ⇒ "へ"
  - s[ -5 ] ⇒ "ち"
  - s[ 3:7 ] ⇒ "にほへと"
  - s[ 0:3 ] ⇒ "いろは"
  - s[ :5 ] ⇒ "いろはにほ"
  - s[ -3 : ] ⇒ "ぬるを"
  - s[ -5: -3 ] ⇒ "ちり"
  - s[ : : 2 ] ⇒ "いはほとりる"
  - s[ 1: : 2 ] ⇒ "ろにへちぬを"
  - s[ :: -2 ] ⇒ "をぬちへにろ"

# リストのインデックス式

- インデックスの式を用いて、要素を 1 つ取り出すことができる
- 文字列のインデックスと同じで、範囲は、

  正の整数の場合：0〜要素の個数-1

  負の整数の場合：-1 〜 - 要素の個数
- len( リスト )で要素の個数を返す
- 例
  - numlist[ 0 ]  # 最初の要素
  - numlist[ -1 ] # 最後の要素
  - numlist[ len(numlist) -1 ] # 最後の要素

# リストのスライス式の例

- スライスで複数の要素を取り出すことができる
- 結果は、新たなリストとして返されることに注意
- 例：
  - numlist[ 2:3 ] # 1個の要素の場合でもリストとして返す
  - numlist[ 5:9 ] # ５番目から９番目の前まで
  - numlist[ : 5 ] # 最初の５個の要素
  - numlist[ -5: ] # 最後の５個の要素
  - numlist[ 2:7:2 ] # ２番目から６番目まで、１つ飛ばし
  - numlist[ ::-1 ] # 逆順に取り出す
  - numlist[ ::-2 ] # １つ飛ばしで逆順
  - numlist[ 1:-1:-2 ] は動かないので、以下のようにする

    numlist[ -2:0:-2 ]
    numlist[ 1:-1][::-2 ]

# リストのインデックス式・スライス式の例

- nlist = [ 12, 23, 39, 86, 45, 66, 49, 52 ]
  - nlist[ 0 ]　⇒　12
  - nlist[ -1 ]　⇒　52
  - nlist[ len(nlist)-2 ]　⇒　49
  - nlist[ -len(nlist) ]　⇒　12
  - nlist[ 3 ]　⇒　86
  - nlist[ -3 ]　⇒　66
  - nlist[ 0:3 ]　⇒　[ 12, 23, 39 ]
  - nlist[ :2 ]　⇒　[ 12, 23 ]
  - nlist[ -4: -2 ]　⇒　[ 45, 66 ]
  - nlist[ -3:  ]　⇒　[ 66, 49, 52 ]
  - nlist[ -2: len( nlist ) ]　⇒　[ 49, 52 ]
  - nlist[ : : -1 ]　⇒　[52, 49, 66, 45, 86, 39, 23, 12]
  - nlist[ : : 2 ]　⇒　[12, 39, 45, 49]
  - nlist[ 0:-1][::-2]　⇒　[49, 45, 39, 12]

# タプルでのインデックス式・スライス式の例

- tup = ( 123, "Adam Smith", 45.6, True, 3j )
  - tup[ 0 ] → 123
  - tup[ 2 ] → 45.6
  - tup[ -1 ] → 3j
  - tup[ -2 ] → True
  - tup[ -len( tup ) ] → 123
  - tup[ len( tup )-3 ] → 45.6
  - tup[ 0:1 ] → (123,)
  - tup[ 2:4 ] → (45.6, True)
  - tup[ :2 ] → (123, "Adam Smith")
  - tup[ -2: ] → (True, 3j)
  - tup[ ::2 ] → (123, 45.6, 3j)
  - tup[ ::-1 ][ 1::2 ] → (True, 'Adam Smith')

# リスト・タプル・文字列への演算（加算・整数との乗算）

- 対象 + 対象（対象の型は同じでなければならない）
  - 対象同士が連結されたものが新たに生成される
  - 例：

    [ 1, 2 ] + [ 3, 4 ] ⇒ [ 1, 2, 3, 4 ]
  - (1, "ぶぶづけ" ) + ( "いかが", True )

    ⇒ (1, 'ぶぶづけ', 'いかが', True)
  - "たらこ" + "いくら" ⇒ 'たらこいくら'
- 対象 * 整数　あるいは　整数 * 対象
  - その対象が、整数回分繰り返されて、繋がったものが新たに生成される
  - 整数が0以下だと空の対象が作成される
  - 例：["a","b" ] * 4

    ⇒ [ "a","b", "a","b", "a","b", "a","b" ]
  - ("いか", 23.4) * 3

    ⇒ ('いか', 23.4, 'いか', 23.4, 'いか', 23.4)
  - "うらら" * 2 + "うらうらら"

    ⇒ 'うららうららうらうらら' # 山本リンダ「狙いうち」より

# リスト・タプル・文字列・集合・辞書での要素の包含判定

- 値 **in** 対象
  - その値が、対象の要素として含まれていればTrue
  - 例：12 **in** [ 23, 12, 45 ] ⇒ True
  - "ぶらら" **in** ( 4.5, "ぺろろ", "とぷぷ", 23 )
    ⇒ False
  - "ぱらいそ" **in** "みんな、ぱらいそさ、いくだ" ⇒ True
  - "A" **in** { "A", "C", "F" } ⇒ True
- 値 **not in** 対象
  - その値が、リストの要素として含まれていなければTrue
  - 2.3 **not in** [ 1.2, 2.2, 3.2, 4.2 ]
    ⇒ True
  - 4.5 **not in** ( 4.5j, 4.5, "4.5", 45 )
    ⇒ False
  - "ぽぽ" **not in** "ここなるる"
    ⇒ True
  - "one" **not in** { "one" : 1, "two" : 2 }
    ⇒ False

# リスト・タプル・文字列の演算の注意点

- リスト・タプル・文字列同士の足し算をすると、別のリスト・タプル・文字列が新規に生成される
  - "2345" + "6789" ⇒ "23456789" # 整数の加算にならないことに注意
  - "千里の道も" + "一歩から" ⇒ "千里の道も一歩から"
  - y = [ 12, 34, 45 ] ; y + [ 56, 67 ] ⇒ [ 12, 34, 45, 56, 67 ] # yの値は変化しない

- リスト・タプル・文字列と整数値の掛け算をすると、複数回コピーされたリスト・タプル・文字列が新規に生成される
  - uri = "うり" * 2
  - uri = uri + "が" + uri + "にいく" + uri + "の声" ⇒ "うりうりがうりうりにいくうりうりの声"
    # 新規に計算された文字列でuriが上書きされる

# 文字列のフォーマット

- Pythonには、4種類のフォーマット方式がある

  - C/C++のprintf関数と互換を持たせたフォーマット演算（%演算子でフォーマットする）
  - Python独自のフォーマット
    - format組込み関数を用いたもの
    - 文字列オブジェクトのformatメソッドを用いたもの
    - フォーマット済み文字列リテラル（Python 3.6より）

# 文字列のフォーマット演算

- C/C++言語のprintf関数との互換性を考えて、Pythonには文字列のフォーマット演算子がある
  - "フォーマット文字列" % ( 引数... )
- フォーマット文字列には、次のような文字が使える
  - %d　%nd　%0nd　　整数用　nは数字（nは最低何桁で表示するか）
  - %x　%nx　　整数用16進数　nは数字（nは最低何桁で表示するか）
  - %f　%n.mf　　実数用　n,mは数字（mは小数部の桁数上限）
  - %e　%n.me　　実数用、指数表記（mは正規化された後の小数部の桁数上限）
  - %s　%n.ms　　文字列用　n,mは数字（mは先頭から表示される文字数）
  - -をつける　　左寄せになる
  - +をつける　　符号がつく（数のみ）

# フォーマット文字列一覧

- %d…10進数として表示する
- %o…8進数として表示する
- %x…16進数として表示する
- %X…16進数として表示する（A-Fを大文字で）
- %f…実数として表示する
- %e…指数形式の実数として表示する
- %c… 1 文字として表示する
- %s…文字列として表示する

# フォーマットの桁指定例

- %6d…6文字分は最低限確保される。足りなければ、左側に空白が詰められる
- %06d…6文字分は最低限確保される。足りなければ、左側に0が詰められる
- %+d…必ず符号が含まれる（符号で上記の指定の 1 文字分は消費される）
- %-8s…8文字分は確保される。結果は左揃えになる（通常は右揃え）
- %#o…かならず0oで始まる8進数として表示する
- %#x…かならず0xで始まる16進数として表示する
- %.2f …小数部は、小数第3位で四捨五入されて2桁になる（指定がなければ、6桁）
- %10.3f…全体で最低10桁（小数点含む）、小数部は小数第4位で四捨五入されて3桁になる
- %+7.2e…全体で最低7桁（符号・小数点を含む）、符号は必ず表示、小数部は正規化された後に小数第3位で四捨五入されて、2桁で表示される
- %10.3s…表示部分は10文字分確保されるが、そのうち実際に文字が表示されるのは3文字分だけ

# 文字列の%演算子によるフォーマット例

- 整数
  - value = 123 # 整数値
  - "%d" % value　→　"123"
  - "%06d" % value　→　"000123"
  - "%6d" % value　→　"　　123"
  - "%+d" % value　→　" +123"
  - "%o %#o" % (value, value)　→　"173 0o173"
  - "%x %#x" % (value, value)　→　"7b 0x7b"
- 実数
  - rvalue = 123.4567 #実数値
  - "%f" % rvalue　→　"123.456700"
  - "%12f" % rvalue　→　"  123.456700"
  - "%.3f" % rvalue　→　"123.457"
  - "%08.2f" % rvalue　→　"00123.46"
  - "%+.4f" % rvalue　→　"+123.4567"
- 文字列
  - aiueo = "あいうえおかきくけこ"
  - "%s" % aiueo　→　"あいうえおかきくけこ"
  - "%.4s" % aiueo →　"あいうえ"
  - "%10.3s" % aiueo　→　"　　あいう"
  - "%-10.2s" % aiueo　→　"あい　　"
  - "%c" % aiueo[ 0 ]　→　"あ"
- 複合
  - "%.5s %d %+.2f" % (aiueo, value, rvalue)　→　"あいうえお 123 +123.46"

# Python独自のフォーマット様式

- フォーマット文字列の文法
  - format_spec ::= [[fill]align][sign][#][0][width][grouping_option][.precision][type]

  - fill　　::= <any character> 間を埋める文字
  - align　　::= "<" | ">" | "=" | "^" 左詰め, 右詰め, 数字, 中央寄せ
  - sign　　::= "+" | "-" | " " 符号付き, 負数だけ符号, 正の数は空白で負の数はマイナスがつく
  - #　　16進数で0xがつく、8進数で0oがつく、2進数で0bがつく
  - 0　　数値を埋める為に、0が前につく
  - width　　::= digit+ 最低幅の文字数を指定　digit=0〜9　+は1回以上の繰返し
  - grouping_option ::= "_" | "," 3桁ごとに記号をつける
  - precision　　::= digit+ 最大幅の文字数を指定
  - type　　::= "b" | "c" | "d" | "e" | "E" | "f" | "F" | "g" | "G" | "n" | "o" | "s" | "x" | "X" | "%"

- BNFで書かれていて、[]は省略可能、::= は置き換え、| は、どれか1つを選択する記号

# 型指定（整数）

- 'b' … 2進数。出力される数値は2を基数とする。
- 'c' … 文字。数値を対応する Unicode 文字に変換する。
- 's' … 文字列。
- 'd'… 10進数。出力される数値は10を基数とする。
- 'o'… 8進数。出力される数値は8を基数とする。
- 'x'… 16進数。出力される数値は16を基数とする。 10進で9を越える数字には小文字が使われる。
- 'X'… 16進数。出力される数値は16を基数とする。 10進で9を越える数字には大文字が使われる。

# 型指定（数値）

- 'e'… 指数表記。指数を示す 'e' を使って数値を表示する。デフォルトの精度は 6 。
- 'E'… 指数表記。大文字の 'E' を使うことを除いては、 'e' と同じ。
- 'f'… 固定小数点数。数値を固定小数点数として表示する。デフォルトの精度は 6 。
- 'F'… 固定小数点数。nan が NAN に、inf が INF に変換されることを除き 'f' と同じ。
- 'g'… 汎用フォーマット。精度を p >= 1 の数値で与えた場合、数値を有効桁 p で丸め、桁に応じて固定小数点か指数表記で表示する。複素数にも使用可能。
- 'G'… 汎用フォーマット。数値が大きくなったとき、 'E' に切り替わることを除き、 'g' と同じ。
- 'n'… 数値。現在のロケールに従い、区切り文字を挿入することを除けば、 'd' あるいは 'g' と同じ。複素数にも使用可能。
- '%'… パーセンテージ。数値は 100 倍され、固定小数点数フォーマット ('f') でパーセント記号付きで表示される。
- None （型を指定しない）… 'd' あるいは'g'と同じ。複素数にも使用可能。

# format組込み関数によるフォーマット例

- format組込み関数を使ったフォーマットができ、フォーマットされた新しい文字列が生成される
  - format( 値, "フォーマット文字列" )

- 例：
  - format( 123.4, "+4.2" )　⇒ "+1.2e+02" # 自動的に実数と判定
  - format( -123, "-08,d" )　⇒ "-000,123"
  - format( 4+5j, "10g" )　⇒ "　　4+5j"
  - format( 123.4567, "+6.1f" )　⇒ "+123.5"
  - format( 0.568, ".2%" )　⇒ "56.80%"
  - format( "あいうえお", ">10.2s" )　⇒ "　　あい"

# Python文字列クラスのformatメソッドによるのフォーマット

- 書式
  - 文字列あるいは文字列変数 .format( 値, ... )
- 文字列の中
  - {} … パラメータの値に順次対応
  - {n} … n番目のパラメータの値（0から始まる）
  - {}の中で、:以降にフォーマット文字列を指定することができる、指定がないと自動的に判断
  - {}の中で変数を指定でき、formatのパラメータで、変数=値で、その変数に代入することができる
- 使用例
  - "{}, {}, {}".format( 12, 34, "ABC" )
    → "12, 34, ABC"
  - "{0}, {1}, {2}".format("a", "b", "c")
    → "a, b, c"
  - "int: {0:d};  hex: {0:x};  oct: {0:o};  bin: {0:b}".format(42)
    → "int: 42;  hex: 2a;  oct: 52;  bin: 101010"
  - "int:{0:d}; hex:{0:#x}; oct:{0:#o}; bin:{0:#b}".format(42)
    → "int:42;  hex:0x2a;  oct:0o52;  bin:0b101010"
- 変数代入の使用例
  - "Coord: {lat}N, {lon}E".format(lat="35.39", lon="139.437")
    → "Coord: 35.39N, 139.437E"
- 詳しくは、
  https://docs.python.org/ja/3.9/library/string.html

# フォーマット済み文字列リテラル

- Python3.6から利用可能
- 文字列の前にfを付けると、format関数を介さなくても、自動的にフォーマットした文字列を生成してくれる機能がついた。
  - 文字列のformatメソッドと使い方は似ているが、{}の中に必ず値や式を記述する必要がある
  - {}の中の:以降でフォーマットの指定をすることができる、指定がないと自動的に判断される
- 使用例：

```
name = "Fred"
print( f"He said his name is {name}." )
```

  - 出力：He said his name is Fred.

```
width = 10
precision = 4
value = decimal.Decimal("12.34567")
f"result: {value:{width}.{precision}}"  # フォーマットの指定の中に更に変数を使うこともできる（ネスト指定）
```

  - 結果の文字列：'result:      12.35'

# 文字端末（シェル）への表示

- print()関数を使う
  - print( 式 )
  - print( 式, 式, … ) # 各式の間は空白 1 文字で区切る

- 改行や区切り文字の指定が出来る
  - print( 式 ) # 表示したあと改行
  - print( 式, end="" ) # 改行せず
  - print( 式, 式, …, sep=":" ) # 区切り文字を変える
  - 改行用の特殊記号として\nが使える

# 文字端末（シェル）からの入力

- input()関数を使う
  - 文字列を保持する変数 = input( )
  - 文字列を保持する変数 = input( "入力を促すプロンプト" )

- 文字列を返してくるので、数値など他の型にするためには、型を変換する必要がある

# 文字列から数への変換

- 整数
  - int( 文字列 )
  - int( 文字列, base=基数 ) # 2〜36進数（a〜zを使う）まで対応
- 実数
  - float( 文字列 )
- 論理値
  - bool( 文字列 ) …空の文字列""だけがFalseになり、それ以外の文字列はTrue
- 複素数
  - complex( 文字列 )

# 数値から文字列への変換

- 標準ライブラリで定義されている
  - str( 整数 )　… 10進数の文字列に変換される
  - str( 実数 )　… 文字列に変換される
  - str( 複素数 ) … 文字列に変換される
  - str( 論理値 ) …  文字列に変換される
  - bin( 整数 )　… 2進数に変換される（0b付き）
  - oct( 整数 )　… 8進数に変換される（0o付き）
  - hex( 整数 )　… 16進数に変換される（0x付き）

# n進数の文字列の生成

- numpyには、整数をn進数の文字列に変換する関数が用意されている
  - base_repr( 整数, n )
- numpyをインストールする
  - pip install numpy　←ターミナル（Shell）から実行する
  - arch -arm64 pip3 install numpy… M1/M2/M3 Mac の場合
- 使い方の例

```
import numpy
numpy.base_repr( 17, 6 )  →  '25'
numpy.base_repr( 1840, 13 )  → 'AB7'
```

# 暗黙の型の変換

- 暗黙の型変換
  - 実数が式の中に出てくると、式の型が実数へ自動的に変換される
  - 例：45 * 1.23 ⇒ 55.35
  - 複素数が式の中に出てくると、式の型が複素数へ自動的に変換される
  - 例：45 * (1.23 + 3.4j) ⇒ (55.35+153j)
  - 除算演算子 / が出てくると、実数になる
  - 例：3 / 2 ⇒ 1.5
  - 整数除算・剰余は、両方の項が整数のときだけ、整数になる
  - 例：432 % 37 ⇒ 27,　 5.0 % 2 ⇒ 1.0 （片方が実数だと、結果も実数になる）

- 明示的な型変換
  - 変換用の関数を使う

# 型の変換（明示）

- 明示的な型変換
  - 変換の関数を用いる
  - 変換可能であれば、その型に変換される
- **int( 式 )**　整数への変換
  - 実数から変換する場合は、小数部が切り取られる
- **int( 式, base=n )**　n進数の文字列から10進数の整数への変換
  - 変換後は、10進数として保持
- **float( 式 )**　実数への変換
- **complex( 式 )**　複素数への変換
- **bool( 式 )**　論理値への変換…数値が0あるいは0.0, 0j, 0+0jなどの場合だけFalse、それ以外はTrue
- **str( 式 )** 文字列への変換