スクリプト言語プログラミング
Pythonによる数値解析

第３回講義資料

箕原辰夫

文字端末（シェル）への表示

- print()関数を使う
  - print(式)
  - print(式, 式, …) # 半角の空白 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　標準では、print(式, sep=" ", end="\n")
  - print(式, end="") # 改行せず
  - print(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
  - 文字列変数 = input( )
  - 文字列変数 = input("入力を促すプロンプト")

文字列から数への変換

- 整数
  - int(文字列)
  - int(文字列, base=基数)　基数の範囲は2〜36
- 実数
  - float(文字列)
- 論理値
  - bool(文字列) … 空の文字列のときだけFalseになる
- 複素数
  - complex(文字列)

数値の文字列への変換

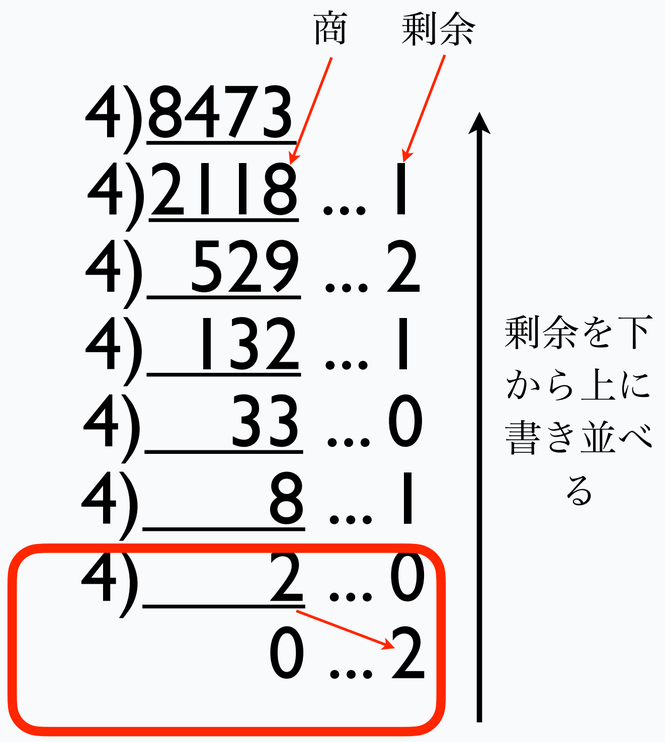
- 標準ライブラリで定義されている
  - str(整数)　… 10進数に変換される
  - str(実数)　… 10進数標記の実数の文字列に変換される
  - str(複素数)　… 10進数標記の複素数の文字列に変換される
  - str(論理値) … "True" / "False"のいずれかに変換される
  - bin(整数)　… 2進数に変換される(0bが先頭に付く）
  - oct(整数)　… 8進数に変換される（0oが先頭につく）
  - hex(整数)　… 16進数に変換される（0xが先頭につく）

- 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)

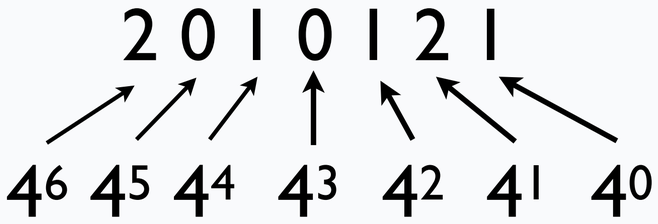
- 使用例

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

n進数への変換



割られる数が割る数よりも大きくなった場合は、商に0を剰余に割られる数を求める

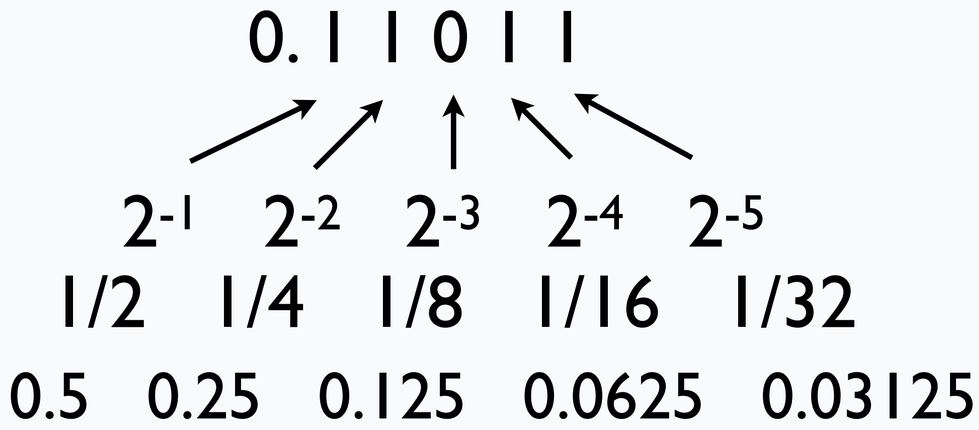


2 x 4096
0 x 1024
1 x 256
0 x 64
1 x 16
2 x 4
1 x 1

8192+256+16
+8+1=8473

２進数の小数の表現

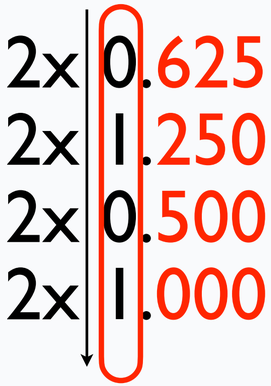
- ２のべき乗（マイナス）との掛算になる



- 例：　　0.11011 = 0.5 + 0.25 + 0.0625+0.03125 = 0.84375
  0.101　= 0.5 + 0.125 = 0.625
  0.0111 = 0.25 + 0.125 + 0.0625 =0.4375

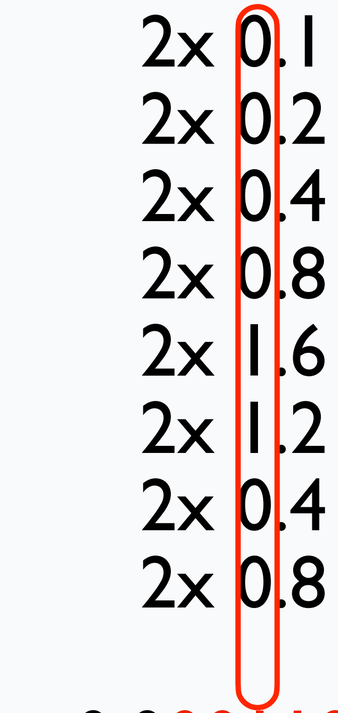
２進数の小数への変換

- 小数部だけを２を掛けていく
- 小数部が0になったら終了
- 整数部だけを上から読んでいく



= 0.101

- 循環小数になることが多い



= 0.0001100....

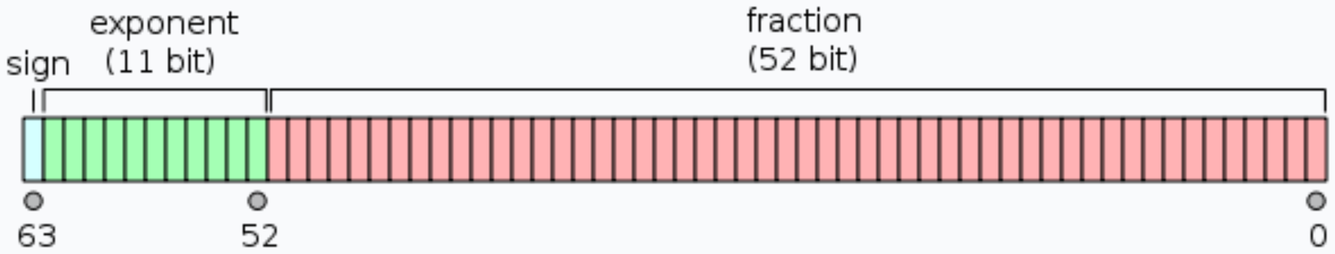
循環する部分

IEEE-754の実数の内部表現との変換

- 実数は、IEEE-754の規格に基づく倍精度実数表現で内部的に表現されており、64bitの2進数の3つの部分に分かれて表現されている
  - 符号（＋…0, −…1）
  - 指数部（11ビット）

    bin(2 ** (exponent+1024))
  - 仮数部（52ビット）

    bin(1 + fraction)



  - 数としては、(-1) * sign * (1+fraction) * (2**(exponent-1024))の指数標記で表わされる実数となる
  - また、なるべく仮数部の桁数を多くするように正規化して表現される（例：345.6e23 → 3.456e25）
- Pythonでは、この内部表現形式を16進数の文字列を使って直接標記することができる
  - 文法：[sign] ['0x'] integer ['.' fraction] ['p' exponent]
- floatクラスのfromhexとhexメソッド（関数）を使って文字列との変換ができる
  - 使用例：float.fromhex('0x3.a7p10') → 3740.0

    float.hex(3740.0) → '0x1.d380000000000p+11' # 正規化されるので、元の数ではなく、"0x1.d38p+11"となる

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
  - 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 → 実数型, 5 * 1j + 12.4 → 複素数型
  - 除算はすべて、実数になる（複素数が含まれる除算は、複素数になる）
  - 整数除算//の場合は、被除数と除数の両方が整数の場合だけ整数になる

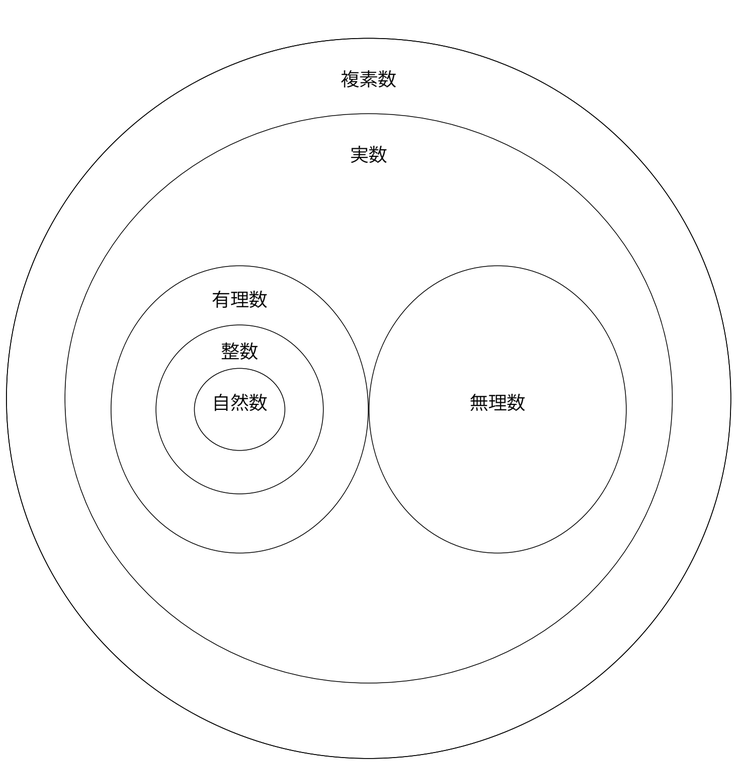
- 明示的な型変換
  - 変換用の関数を使う

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)**　整数や数字列からの実数への変換
  - 複素数からは直接変換できない
  - 整数で桁数が多い場合は、仮数部が17桁ぐらいで切り捨てとなる
- **complex(式)**　整数、実数や数字列からの複素数への変換
- **bool(式)**　論理値への変換
  - 文字列の場合、空の文字列""だけがFalse、それ以外はTrue
  - 数値の場合は、0だけがFalse、それ以外の値はTrue
- **str(式)** 文字列への変換

有理数と無理数について

- 数を表わせる範囲は、右図のようになっています。
- ただし、Pythonでは、実数よりも整数の方が精度が良いので、精度的には内包状態が逆転しています。
- 有理数（分数）を表わす為のクラスが、Pythonには用意されています。
  - 標準ライブラリfractions：Fraction　例：Fraction(1, 3)
  - Sympy：Rational　例：Rational(5, 7)
- 無理数を表わすクラスは、Sympyにだけ用意されています。
  - Sympy：sqrt(value), log(n), log(n, b), value**Rational(1,n), pi, Eなど
- 無理数を扱う場合、実数に始めから変換せずに、Sympyの有理数を使って、一番最後にfloat関数で実数に変換してください。



構造的な型のリテラル

- タプル
  - 書式：(値, 値, …)
  - 複数の値をまとめることができる
  - 値が置かれている順序を保持
  - 異種の型のリテラルをまとめることが多い
  - それぞれの要素を書き換えることはできない（immutable）
  - 例：

    () … 空のタプル

    (324.2,) … 要素が1つのタプル

    (34, "ABC", 78.9, "一弐参")

    (23.4, 45.6, 56.7)

- リスト
  - 書式：[値, 値, …]
  - 複数の値をまとめることができる
  - 値が置かれている順序を保持
  - 同種の型のリテラルをまとめることが多い
  - それぞれの要素を取り換えることが可能（mutable）
  - 例：

    [] … 空のリスト

    [342] … 要素が1つのリスト

    [34, "ABC", 78.9, "あいう"]

    ["東京", "京都", "都島"]

    [12, 23, 34 ,45]

リスト

- 複数の値をまとめておける
  - リスト、タプルがPythonには用意されている
- リストの記述は、[ ]を用いる
  - 例：numlist = [2, 3, 5, 7, 11, 13, 17, 19, 23, 31] # 素数リスト
- リストの個々の値は、要素と呼ばれる
- Pythonでは、要素の型は異なっていても構わない
  - 例：heterogeneous = [345, True, "WOW", 45.6e-12]
- リストの要素の個数を調べるのにlen組込み関数が使える
  - 例：len(numlist)

リストへの変換

- list()関数を使う
  - 書式：list(値)
  - 例：list((56, 32, 193, 23))
  - 評価は、[56, 32, 193, 23]となる
- 他のオブジェクトから、リストの形にするときにもlist関数が用いられる
  - 例：list(range(5))
  - 評価：[0, 1, 2, 3, 4]

タプル

- タプルは変更不可能なデータの連なりで、一般的に異種のデータの集まりを示す
- 丸括弧の対を使い、空のタプルを表す: ()
- カンマを使い、単要素のタプルを表す: a, または (a,)
- 項目をカンマで区切る: a, b, c または (a, b, c)
- 組み込みの tuple()関数でタプルに変換することが可能

リスト・タプル・文字列・Rangeのサイズ（長さ）とインデックス式

- リスト・タプル・文字列・集合・辞書の長さ（要素の個数）を求める
  - len(s)　# sに代入されている構造リテラルの長さを返す
- リスト・タプルから要素を取り出す、または、文字列の中から 1 文字を取り出す
  - シーケンス型の値あるいは変数[インデックス] を使う
  - シーケンス型（sequence type）に含まれるのは、リスト・タプル・文字列・rangeクラスのオブジェクトの3つ
  - インデックスは整数式で、範囲は0〜(文字列の長さあるいは要素の個数)-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〜-(文字列の長さあるいは要素の個数)
- 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・Rangeへのインデックス式の値の範囲

- 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
- インデックスの範囲は、
  正の整数の場合：0〜要素の個数-1、0〜文字列の長さ-1（文字列の場合）
  負の整数の場合：-1 〜 - 要素の個数、-1〜-文字列の長さ（文字列の場合）
- 例
  - numlist[0] # 最初の要素
  - numlist[-1] # 最後の要素
  - numlist[len(numlist) -1] # 最後の要素
- 範囲外の要素をアクセスした場合は、実行時エラー「IndexError: list index out of range」が表示される

リスト・タプル・文字列・Rangeのスライス式

- リスト・タプル・文字列・Rangeクラスのオブジェクトの中から範囲を指定して一定の範囲のシーケンスを新たに作り出す（コピーされる）
- スライスの記法は、以下の通り、インデックスを用いる
  - 対象[始め:終わった次]
  - 対象[始め:終わった次:間隔]
- 例：
  - s[0: 5]　# 最初の5つ
  - s[-5: -1]　# 最後から５つ目から４つ分
  - s[7: 12: 2]　# １つ飛ばしつつ
- 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

リスト・タプル・文字列のスライス式

- スライスで複数の要素を取り出すことができる
- 結果は、新しく生成されたリスト・タプル・文字列として返される
- 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、以下の書き方で回避

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

リスト・タプル・文字列に適用できる組込み関数

- all(論理値のリスト)…すべての要素がTrueのときだけTrue（数値のリストの場合は、すべてが0以外のときTrue）
- any(論理値のリスト)…すべての要素がFalseのときだけFalse（数値のリストの場合は、すべてが0以外のときTrue）
- enumerate(リスト・タプル・文字列, start=0)…インデックスと要素の対のシーケンス（enumerated object）を発生させる
- len(リスト・タプル・文字列)…リストの長さを求める
- list(リスト・タプル・文字列)…新たなリストを生成する
- max(リスト・タプル・文字列)…リスト中の最大値を求める
- min(リスト・タプル・文字列)…リスト中の最小値を求める
- sorted(リスト・タプル・文字列)…ソートされた新たなリストを返す
- sum(リスト・タプル [, 初期値])…要素の総和を求める（初期値が指定されると最初の値は初期値で設定される）
- reversed(リスト)…反対順になった新たなオブジェクト（reversed object）を返す
- tuple(リスト・タプル・文字列)…新たなタプルを生成する
- zip(リスト・タプル・文字列, リスト・タプル・文字列, …)…各リストの先頭からの要素を順番にまとめたタプルから構成されるシーケンス（zip object）を発生させる

リストの追加・削除・検索

- リスト.append(value)…最後に要素を追加
- リスト.remove(value)…該当する値の要素を削除
- リスト.insert(i, value)…i番目に、その値をもつ要素を挿入
- リスト.sort()…そのリスト自体をソートする
- リスト.reverse()…要素の順番を反転させる
- リスト.extend(リスト)…リストの結合（+演算と同じ）
- リスト.clear( )…空リストにする

- リスト.index(value)…その値を持つ要素のインデックスを返す
- リスト.count(value)…その値を持つ要素が何個あるか返す
- リスト.copy( )…浅いコピーで、コピーのリストを作って返す

del文による要素の削除

- **del** s[i]
  - i番目の要素を削除する
- **del** s[i:j]
  - i番目からj-1番目までの要素を削除する
- **del** s[i:j:k]
  - i番目から、kずつインデックスを増やしながら、j-1番目までの要素を削除する

リスト・タプル・文字列の共通の関数（メソッド）

- 以下でsは、リスト・タプル・文字列のリテラルを示す
- s.index(x)
  - s中で指定された値が何番目にあるかが返される（ただし、0番始まり）
  - s.index(x, 検索開始のインデックス)
  - s.index(x, 検索開始のインデックス, 検索終了+1のインデックス)
- s.count(x)
  - sの中に何個指定された値が何個あるのか整数で返される

リスト・タプル・文字列の共通演算

- xを式とし、sおよびtをリストあるいはタプル・文字列とする
- x **in** s　あるいは　x **not in** s
  - xがsに含まれている（いないかどうか）
  - True/Falseが返される
- s + t
  - 結合された新しいリスト・タプル・文字列が返される
- s * 整数　あるいは 整数 * s
  - sが整数回繰り返されたリスト・タプル・文字列が返される
  - ただし、個々の要素が更にリスト・タプルなどのようなオブジェクトの場合は、共有されている可能性もあるので注意のこと

リスト・タプル・文字列の演算の注意点

- 対象同士の足し算（加算）は、足し合わされた別の構造物が新たに生成される
  - "2345" + "6789" → "23456789"
  - (23, 45) + (67, 89) → (23, 45, 67, 89)
  - ["千里の道も"] + ["一歩から"] → ["千里の道も", "一歩から"]

- 対象と整数値の掛け算（乗算）は、対象の要素が複数回コピーされた別の構造物が新たに生成される
  - "うらら" * 2 + "うら" * 5 → "うららうららうらうらうらうらうら"
  - ("たこ", "いか", "うに") * 2 → ('たこ', 'いか', 'うに', 'たこ', 'いか', 'うに')
  - [0] * 8 → [0, 0, 0, 0, 0, 0, 0, 0]

構造的なリテラル（集合と辞書）

- 集合
  - 書式：{ 値, 値, … }
  - 複数の値をまとめることができる
  - 同じ値の要素は 1 つしか登録されない
  - 要素の型は、異種でも構わない
  - 追加・削除可能（mutable）
  - 例：　{ 34, "ABC", 78.9, True }
    {} …空の辞書（型は辞書）
    set( )…空の集合
    { 89 } …要素が1つの集合
- 辞書
  - 書式：{ キー：値, キー：値, … }
  - 複数の値をまとめることができる
  - 同じキーの要素は 1 つしか登録されない
  - キーと値の型は、要素によって、異種でも構わない
  - 書き換えは可能（同じキーに対して後から別の値をペアにすることができ
  - 追加・削除可能（mutable）
  - 例：　{ 34: 56, "ABC": 12, 78.9: "A" }
    {66: "Sixtysix" } …要素が1つの辞書
    dict() …空の辞書　あるいは {}

集合・辞書に適用できる組込み関数

- all(集合・辞書)…すべての要素がTrueのときだけTrue（数値のリストの場合は、すべてが0以外のときTrue）
- any(集合・辞書)…すべての要素がFalseのときだけFalse（数値のリストの場合は、すべてが0以外のときTrue）
- enumerate(集合・辞書, start=0)…インデックスと要素の対のシーケンス（enumerated object）を発生させる
- len(集合・辞書)… 集合・辞書 の要素の個数を求める
- list(集合・辞書)…新たなリストを生成する
- max(集合・辞書)… 集合・辞書中の最大値を求める（要素あるいはキーが比較できる場合）
- min(集合・辞書)…集合・辞書中の最小値を求める（要素あるいはキーが比較できる場合）
- sorted(集合・辞書)…ソートされた新たなリストを返す（辞書の場合はキーのみ）
- sum(集合・辞書 [, 初期値])…要素の総和を求める（初期値が指定されると最初の値は初期値で設定される）（要素あるいはキーが加算できる場合）
- reversed(集合・辞書)…反対順になった新たなオブジェクト（reversed object）を返す
- zip(集合・辞書, 集合・辞書, …)…各リストの先頭からの要素を順番にまとめたタプルから構成されるシーケンス（zip object）を発生させる

集合・辞書・Rangeにも以下の演算・関数が適用可能

- **in , not in**
  - 要素が含まれるかどうか（辞書の場合は、キーにあるかどうかが判別される）
- len(s)
  - 要素の個数が返される
- min(s), max(s)
  - それぞれ最小・最大の要素が返される（辞書の場合はキーについて）
- sum(s)
  - 各要素が足し合わされた値が返される
- s[key]
  - 辞書のみ、キーに対応した値が返される

Set

- 集合型は、{ }で囲むか、set関数を使ってリストなどから生成する、**for**文を使っても生成することができる
  - 例：fruits = { "りんご", "みかん", "なし", "いちご" }
  - 例：xset = { x**2 **for** x **in** range(1, 11) **if** x % 2 == 1 }
  - 例：drinks = set(["紅茶", "コーヒー" , "ジュース"])
- 集合なので、重複項目は、除去される
- **in** / **not in** 演算子を使って、集合に入っているかどうかを判定
  - 例："なし" **in** fruits
- 組込み関数lenで、集合内の要素の個数を調べることができる

Setでの演算

- 集合同士の演算→結果の集合が新たに生成されて返される
  - 和集合 union　A | B
  - 差集合 difference　A - B
  - 交差集合 intersection A & B
  - 排他的論理和集合 symmetric_difference　A ^ B = (A | B) - (A & B)
- 部分集合の判定→論理値が返される
  - 部分集合(subset）か　A <= B
  - 真部分集合か　A < B
  - 上位集合（superset）か　A >= B
  - 真上位集合か A > B

Setと要素

- 集合.add(要素) で、要素を追加する
- 集合.remove(要素)で、要素を削除する、ただし要素がないとエラー
- 集合.discard(要素)で、含まれていればその要素を削除する
- 集合.pop( )で、任意の要素を返し、その要素を削除する、ただし 1 つも要素がないとエラー
- 集合.clear( )で、集合の要素を全部削除する

辞書（Dictionary）

- 辞書では、キーでエントリの値を引いてくることができる、ただしキーは一意である必要がある
- キー値：対応する値のペア（エントリ: entryと呼ばれる）を、{ }の中に入れることで、辞書を作ることができる。for文を利用して作成することも可能
- 例：flowers = { "rose" : "薔薇", "lily": "百合", "hydrangea": "紫陽花" }
- 例： numbers = { n+1: name **for** n, name **in** enumerate(["one", "two", "three"]) }
- dict関数を使っても、タプルを要素としてもつリストなどから辞書を作成することができる
- 例：numbers = dict(zip(["one", "two", "three"], range(1, 4)))

辞書とエントリへの演算

- インデックス式を使った検索と登録
  - 辞書[key]…keyの値を持つvalueを返す
  - 辞書[key] = value…keyとvalueの組み合わせを登録
- 削除
  - **del** 辞書[key]…指定されたキーのエントリを削除する
- キーの存在の確認
  - key **in** 辞書 / key **not in** 辞書…keyが辞書にあるか／ないかどうか論理値で返す

辞書のメソッド一覧

- 辞書.keys( )…辞書にあるすべてのキーを返す
- 辞書.values( )…辞書にあるすべての値を返す
- 辞書.items( )…辞書にあるすべてのエントリを返す
- 辞書.clear( )…全キー（エントリ）をクリアする
- 辞書.pop(キー [, デフォルト値])…キーの値を返し、そのエントリを削除する（デフォルト値が指定されていない場合は、そのキーがなければエラーになる）
- 辞書.clear()…辞書のすべての要素を削除する
- 辞書.copy()…辞書の浅いコピーを返す
- 辞書.fromkey(キーのリスト)…辞書中の指定されたキーのリストから構成される新しい辞書を作成する
- 辞書.get(key, default)…指定したキーの値を取得。キーがなければデフォルト値を返す
- 辞書.popitem()…最後に追加されたキーと値のペアを削除して返す
- 辞書.setdefault(key, default)…指定したキーが存在しない場合のデフォルト値を設定し、その値を返す（エントリは追加される）辞書[key] = defaultと同じ

構造型の型変換

- それぞれの型に変換できる
  - list(リスト・タプル・文字列・集合・辞書・range)
    …リストに変換（辞書の場合はキーだけ）、文字列の場合は、各文字に分解して、リストにする、集合・辞書は順番は不定
  - tuple(リスト・タプル・文字列・集合・辞書・range)
    …タプルに変換（辞書の場合はキーだけ）、文字列の場合は、各文字に分解して、タプルにする、集合・辞書は順番は不定
  - set(リスト・タプル・文字列・集合・辞書・range)
    …集合に変換（辞書の場合はキーだけ）、文字列の場合は、各文字に分解して、集合の要素にする、重複は除去される
  - str(リスト・タプル・文字列・集合・辞書・range)
    …文字列に変換（Pythonインタープリタで表示される文字列）
  - dict(2重リスト・2重タプル・2文字の文字列から構成されるリスト／タプル／集合)
    …辞書に変換、2重リストの各要素、2重タプルの各要素のシーケンスの要素の個数は、2つである必要がある（最初の要素・文字がキーになり、2番目の要素・文字が値になる）

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　%*n*d　%0*n*d　　整数用　*n*は数字
  - %x　%*n*x　　整数用16進数　*n*は数字
  - %f　%*n*.*m*f　　実数用　*n*,*m*は数字
  - %e　%*n*.*m*e　　実数用、指数表記、*n*,*m*は数字
  - %s　%*n*.*m*s　　文字列用　*n*,*m*は数字
  - %c　　1文字用
  - -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）
- %,6d…3桁ごとに、カンマ「,」を入れる
- %#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
- %10.3f…全体で10桁、小数部は四捨五入されて3桁になる
- %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"
  - "%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, 正数は空白
    - width ::= digit+ 最低幅の文字数を指定
    - # ::= 16進数に"0x"、8進数に"0o"、2進数に"0b"をつける
    - 0 ::= 最低幅を満たさない場合、左側に0をつける
    - 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(Backus–Naur form)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、+は1回以上の繰返し

BNF形式

- BNF の表記は次のような導出規則の集合である。
  - <非終端記号> ::= <終端記号または非終端記号や終端記号を使った式>
- 式の中には、以下の演算子が使える
  - []は省略可能
  - | は、どれか1つを選択する
  - +は、1回以上の繰返し
- 例：通常の四則演算を伴う整数の式の記述
  - <expression> ::=
    <factor> '+' <factor> | <factor> '-' <factor> | <factor>
  - <factor> ::=
    <element> '*' <element> | <element> / <element> | <element>
  - <element> ::=
    <constant> | <variable> | (<expression>) | '-'<constant> | '-'<variable>
  - <constant> ::=
    整数値（0〜9から構成される数字）+
  - <variable> ::=
    変数名（アルファベット小文字）+
- これで、以下の式を解釈してみる
  - 45 * 23 - 7 / -y
  - ::= <factor> - <factor>
  - ::= <element> * <element> - <element> / <element>
  - ::= <constant> * <constant> - <constant> / -<variable>

型指定（整数と文字列）

- '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。
- '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。
- 's'…文字列。

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 … '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
- 詳しくは、
  https://docs.python.org/ja/3.9/library/string.html

- 使用例
  - "{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"

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.

```
print( f"He said his name is {name:.2s}." )
```

  - 出力：He said his name is Fr.

```
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'

文字列の評価

- eval組込み関数を使うと、文字列中のPythonの式を評価させることができる
- 書式：eval(文字列)
- 例：
  - eval("56+0x78*0b11101") … 3536と評価される