Python 基礎

Lesson_1　Pythonの基本

変数

●変数宣言

変数 ＝ 値

pythonでは変数宣言の際に型指定は必要ありません。

変数名に「10」や「"Hello world!"」を代入することで自動で型を決めてくれます(型推論)。

print()で表示できますので下記のコードを入力し確認してみましょう。

	構文	表示結果	備考
1	num = 10		int型
2	st = "Hello world!"		文字列型
3	flag = True		boolean型
4	print(num,st)	10 Hello world!	

※print(引数,引数)のようにprint関数の引数は「,」(カンマ)で区切り表示することができます。

●明示的な変数宣言

以前は型宣言はできませんでした。python3.6から明示的に変数宣言できるようになりました。

変数 ： 変数型 ＝ 値

	構文	備考
1	num: int = 10	
2	name: str = "Hello world!"	

●変数の型の確認

type(変数)

pythonでは基本的に型推論で自動的に型が決まります。デバックなどで変数の型を調べたい場合などにtype()関数を使用し型を調べることができます。type()関数の引数に変数を指定します。

	構文	表示結果	備考
1	num , st , flag = 10 , "Hello" , True		1行で複数の変数に代入可能
2	print(num," : ",type(num))	10 ： <class 'int'>	
3	print(st," : ",type(st))	Hello ： <class 'str'>	
4	print(flag," : “,type(flag))	True ： <class 'bool'>	

●型の変換(キャスト)

変更したい型(変数)

pythonでは型変換は自動で行われますが、型作成後、明示的に変更することもできます。

	構文	表示結果	備考
1	num = 10		int型で宣言
2	print(type(num))	<class 'int'>	型を表示。int型
3	str(num)		str型に明示的に変更
4	print(type(str(num)))	<class 'str'>	型を表示。str型に変更されている

計算

●四則演算

pythonの四則演算はCやJavaとほぼ同じですが、modと商の記述方法が少し違います。

	構文	表示結果	備考
1	a ,b = 17,3		1行で複数の変数に代入可能
2	print(a + b)	20	足算
3	print(a - b)	14	引算
4	print(a * b)	51	掛算
5	print(a ** b)	4913	べき乗
6	print(a / b)	5.66666666666667	割算（表示可能領域まで小数点以下を表示）
7	print(a // b)	5	商（整数）
8	print(a % b)	2	mod

●四捨五入

round(引数,桁数)

標準ライブラリに含まれtる「round」関数を使用することで、指定された桁数で引数を四捨五入することができます。

	構文	表示結果	備考
1	a ,b = 17,3		
2	print(round(a / b,3))	5.667	

●Math関数

import math

math.sqrt(引数)

「math」ライブラリをインポートすることにより複雑な計算をすることができます。

	構文	表示結果	備考
1	import math		mathライブラリインポート
2	print(math.sqrt(10))	3.16227766016838	平方根
3	print(math.trunc(3.1415))	3	小数点以下切り捨て

●その他のmathメソッド

メソッド	意味
math.fabs(引数)	引数の絶対値
math.floor(引数)	浮動小数点数の最大の整数を返す
math.ceil(引数)	浮動小数点数以上で最小の整数
math.pow(引数)	べき乗計算
math.log2(引数)	対数計算

上記以外にもmathメソッドがあります。print(help(math))でmathのヘルプを出力しメソッドを確認してみてください。

str（文字列型）

●文字列を代入する

文字列を変数に代入する際は、「'」(シングルクォート)、「"」(ダブルクォート)どちらで囲んでも構いません。ほかの言語では文字が「'」(シングルクォート)文字列が「"」(ダブルクォート)と決まっているものもありますが、pythonでは特に決まっていません。

	構文	備考
1	s1 = "Hello World!"	
2	s2 = 'こんにちは　世界！'	「'」でも文字列を扱える
3	s3 = "今日の天気は'晴れ'です。"	「"」の中で「'」を使用できる

●エスケープシーケンス

「"」や「'」で囲まれた部分で、「"」「'」「¥」などを表示しようとするとエラーが表示されます。「\」を入力してエスケープします。

	構文	表示結果	備考
1	print('I don\'t know')	I don't know	
2	print("D:\\python\\test.txt")	D:¥python¥test.txt	

●rowデータ

r"文字列"

上記のようにエスケープしーけんすを使用し特殊文字をエスケープする方法の他に、文字列の囲みの前に「r」をつけrowデータを出力する方法があります。

	構文	表示結果	備考
1	print(r"D:\python\test.txt")	D:¥python¥test.txt	row(生)データ

●文字列の繰り返し

"繰り返し文字" * 繰り返し回数

文字列を繰り返し表示したい場合は通常for文などを使用しますが、pythonでは、簡易的に文字列の繰り返しを行う事ができます。

	構文	表示結果	備考
1	print("#" * 4)	####	#が4回繰り返される

●文字列の連結

Javaと同様に「+」を使用し文字列を連結することもできます。

	構文	表示結果	備考
1	num ,name = 20,"斎藤"		
2	print(name+"さんは、" +str(num)+"才です")	斎藤さんは、20才です	
3	print(name + "さんは、" + num + "才です")	エラー	文字列＋数値

※文字列と数値を連結する場合はエラーが出るので、数値を文字列型にキャストしなくてはいけません。

●文字のインデックス(文字列から、１文字取り出す)

”文字列”[引数]

文字列から特定の文字を一文字抜き出したい場合は、引数にインデックス番号を指定します。インデックス番号は文字列の先頭文字を0として順に番号が付けられています。
インデックス番号に、「-」(マイナス)を付けると、最後尾から数えた数になります。

	構文	表示結果	備考
1	az = "abcdefghi"		
2	print(az[3])	d	インデクス番号3(4文字目)の文字
3	print(az[-1])	i	最後尾の文字
4	print(az[-3])	g	最後尾から３番目の文字

●文字のスライス(複数文字を取り出す)

”文字列”[開始：終了:間隔]

開始から終了の一つ前までの文字列を抜き出すことができます。インデックス番号は「0」から始まります。

	構文	表示結果	備考
1	az = "abcdefghi"		
2	print(az[3:5])	de	インデックス番号3～4までを表示
3	print(az[0:3])	abc	最初の文字から3文字目までを表示
4	print(az[:3])	abc	[0:3]の0を省略できる
	print(az[2:])	cdefghi	終了のインデックス番号を省略すると最後まで表示
5	print(az[3:-1])	defgh	「-1」は最後の一つ前の文字まで表示
6	prrint(az[:])	abcdefghi	インデックス番号を省略すると文字列全体を表示
7	prrint(az[: :2])	acegi	stepに「2」を入れると１文字飛ばしで表示
8	prrint(az[: :-1])	ihgfedcba	stepに「-1」を入れると後ろから表示

●str（文字列型）のメソッド

変数の後ろに「.」ドットを入力することでメソッドを使用することができます。

メソッド	意味
find(sub[,start,[,end]]) -> int	sub「検索文字」が最初に見つかったインデックス番号を返す。startやendが指定されている場合、その範囲内でsub「検索文字」を探す。検索文字が見つからない場合は「-1」を返す。
count(sub[, start[, end]]) -> int	sub「対象の文字列」の数を返すstartやendが指定されている場合、その範囲内でsub「対象の文字列」を数える。
replace(self, old, new, count=-1, /)	old「元の文字列」をnew「新しい文字列」に置き換えたコピーを返す。第３引数countを指定しない場合は自動的にcountが–1になります。countが-1の場合は全て、countが指定されている場合はその数だけ置換されたコピーを返します。
capitalize(self, /)	先頭文字を大文字に変更し、それ以外の文字を小文字に変更したコピーを返す。
title(self, /)	単語の頭文字を大文字に変更し、それ以外を小文字に変更したコピーを返す。
upper(self, /)	大文字に変換したコピーを返す
lower(self, /)	小文字に変換したコピーを返す

	構文	表示結果	備考
1	s = "春はあけぼの。やうやう白くなりゆく山ぎは、"		
2	print(s.find("あけぼの"))	2	文字列「s」の3文字目で発見
3	print(s.find("あけぼの",5))	-1	文字列「s」の6文字目以降を検索
4	print(s.find("白く",5,11))	-1	文字列「s」の6文字目から11文字目までの範囲で検索
5	print(s.count("やう"))	2	「やう」の数
6	print(s.count("やう",8))	1	9文字目以降の「やう」の数
7	print(s.count("やう",0,7))	0	最初から7文字目の範囲の「やう」の数
8	print(s.replace("や","よ"))	春はあけぼの、ようよう白くなりゆく山ぎは、	
9	print(s.replace("や","よ",1))	春はあけぼの、ようやう白くなりゆく山ぎは、	
10	print(s)	春はあけぼの、やうやう白くなりゆく山ぎは、	

	構文	表示結果	備考
11	f_name = "class_NAME.text"		
12	print(f_name.capitalize())	Class_name.text	先頭だけ大文字
13	print(f_name)	class_NAME.text	f_nameは変更されていない
14	print(fname.title())	Class_Name.Text	単語の頭文字だけ大文字
15	print(f_name)	class_NAME.text	f_nameは変更されていない
16	print(fname.upper())	CLASS_NAME.TEXT	
17	print(f_name)	class_NAME.text	f_nameは変更されていない
18	print(fname.lower())	class_name.text	
19	print(f_name)	class_NAME.text	f_nameは変更されていない

※上記以外にもstr型のメソッドはたくさんあります。print(help(str))でstr型のメソッドの一覧と説明が確認できます。

●helpの見方

capitalize(self, /)

項目	意味
capitalize	メソッド名
self	今は無視してください。classのところで説明します
/	これ以上引数は無い

replace(self, old, new, count=-1, /)

項目	意味
old , new , count	引数名
count=-1	引数を指定しない場合は「-1」がcountに代入される(デフォルト値)

find(sub[,start,[,end]]) -> int

項目	意味
[ ]	[ ] の中身は省略可能。引数として指定しなくても良い
-> int	戻り値がint型

●format：文字列に文字を挿入

"文字列{挿入箇所1}文字列{挿入箇所2}".format(引数1,引数2)

文字列の中に挿入箇所を作成しておいて、formatメソッドの引数をその箇所に挿入できます。

	構文	備考
1	s1 = input("誰が：")	input(引数)： コンソールから入力された文字を変数に格納する
2	s2 = input("何を")	
3	s3 = input(“どうした”)	
4	print("{ }が{ }を{ }".format(s1,s2,s3))	入力した文字が代入されて表示される。

●f-string：文字列に文字を挿入

f"文字列{変数1}文字列{変数2}"

python3.6から、f-stringという書き方もできるようになりました。文字列のクォートのまえにfを付け、{ }の中に挿入する変数名を入力します。

	構文	備考
1	x,y,z = "今日","天気","晴れ"	
2	print(f"{x}の{y}は{z}だ")	今日の天気は晴れだ
3	num = 1234	

●書式指定

f"文字列{変数1:書式指定子}"

文字列のフォーマット（右寄せ、表示桁指定など）を指定する事ができます。

	構文	書式指定子	表示	備考
1	num = 1234			
2	print(f'{num}')	なし	1234	
3	print(f'{num:,}')	,	1,234	桁区切り
4	print(f'{num:8}')	8	1234	8桁で表示。指定桁未満の場合はスペース
5	print(f'{num:08}')	0	00001234	0埋め。＃や＊なども指定できます
6	print(f'{num:*>8}')	>	****1234	右寄せ、８桁
7	print(f'{num:*^8}')	^	**1234**	中央寄せ、８桁
8	print(f'{num:*<8}')	<	1234****	左寄せ、８桁
9	num = 1234.567			
10	print(f'{num:.2f}')	.2f	1234.57	小数点第3位を四捨五入し、2桁表示

Lesson_2　データ構造

list（リスト型）

●リストの作成と表示

リスト変数名　＝　[リスト内容]

リストは**[ ]**で囲んで宣言します。

	構文	表示結果	備考
1	lis = ["1月","2月","3月","4月"]		空のリストの宣言：lis = [ ]
2	print(lis)	['1月', '2月', '3月', '4月']	リストの中身を表示
3	print(lis[2])	3月	リストの要素番号は「0」から開始
4	print(lis[-1])	4月	リストの最後の要素を表示
5	print(lis[1:3])	['2月', '3月']	スライスも指定できます
6	print(lis[::2])	['1月', '3月']	１つ飛ばしで表示
7	print(lis[::-1])	['4月', '3月', '2月', '1月']	逆順で表示
8	print(len(lis))	4	要素数を表示

●リストの操作

リストの結合や要素の追加削除なども簡単に行えます。

	構文	表示結果	備考
1	a ,b = [1,2,3,4],[5,6,7,8,9]		
2	x = [a,b]		二次元配列
3	print(x)	[[1, 2, 3, 4], [5, 6, 7, 8, 9]]	
4	print(x[1])	[5, 6, 7, 8, 9]	
5	x = a + b		結合してxに格納
6	print(f"結合：{x}")	結合：[1, 2, 3, 4, 5, 6, 7, 8, 9]	
7	a += b		aに、 b を追加
8	print(f"aを拡張：{a}")	aを拡張：[1, 2, 3, 4, 5, 6, 7, 8, 9]	
9	a[4] = 20		指定した内容を置換
10	print(f"置換：{a}")	置換：[1, 2, 3, 4, 20, 6, 7, 8, 9]	
11	a[5:7]=[21,22]		範囲指定し複数置換

	構文	表示結果	備考
12	print(f"複数置換：{a}")	複数置換：[1, 2, 3, 4, 20, 21, 22, 8, 9]	

●リストのメソッド

リストにも様々なメソッドがあります。

メソッド	意味
append(self, object, /)	objectをリストに追加する。
clear(self, /)	リストの内容を削除する。
copy(self, /)	リストの内容をコピーする。
count(self, value, /)	リストに含まれる、valueの数を数える。
extend(self, iterable, /)	iterableの中身を追加する。
index(self, value, start=0, stop=9223372036854775807, /)	リストからvalueを検索して、インデックス番号を返す。start,stopの指定があるときは、その範囲内で探す。値が見つからないときは「ValueError」を返す。
insert(self, index, object, /)	指定されたindex番号の前に、objectを追加する
pop(self, index=-1, /)	指定されたindex番号の要素を削除する。リストが空、indexが範囲外の場合は「IndexError」を返す。
remove(self, value, /)	リストに含まれるvalueを削除（最初の値のみ）。リストにvalueが含まれない場合は「ValueError」を返す。
reverse(self, /)	リストの中身を反転する。
sort(self, /, *, key=None, reverse=False)	リストの中身を昇順で並び替えます。 「reverse=True」にすると降順で並び替えます。

	構文	表示結果	備考
1	lis ,lis2 = [1,2,3,4,5,1,2,3],[6,7,8,9,10]		
2	lis.append(10)		リストに追加
3	print(f"append：{lis}")	append：[1, 2, 3, 4, 5, 1, 2, 3, 10]	
4	lis.insert(0,200)		指定した箇所に値を追加
5	print(f"insert：{lis}")	insert：[200, 1, 2, 3, 4, 5, 1, 2, 3, 10]	
6	x = lis.pop()		最後の値を取り出す
7	print(f"x:{x},lis：{lis}")	pop：[200, 1, 2, 3, 4, 5, 1, 2, 3]	
8	del lis[0]		指定した要素番号を削除
9	print(f"del：{lis}")	del：[1, 2, 3, 4, 5, 1, 2, 3]	

	構文	表示結果	備考
10	lis.remove(2)		指定した値を削除
11	print(f"remove：{lis}")	remove：[1, 3, 4, 5, 1, 2, 3]	
12	lis.extend(lis2)		引数のリストを結合
13	print(f"extend：{lis}")	extend：[1, 3, 4, 5, 1, 2, 3, 6, 7, 8, 9, 10]	
14	ans = lis.index(3)		値のインデックスを取得
15	print(f”インデックス番号：{ans}”)	インデックス番号:1	
16	ans = lis.count(3)		値(3)をカウント
17	print(f"カウント：{ans}")	カウント：2	
18	lis.sort()		昇順で並び替え
19	print(f"sort(昇順)：{lis}")	sort(昇順)：[1, 1, 2, 3, 3, 4, 5, 6, 7, 8, 9, 10]	
20	lis.sort(reverse=True)		降順で並び替え
21	print(f"sort(降順)：{lis}")	sort(降順)：[10, 9, 8, 7, 6, 5, 4, 3, 3, 2, 1, 1]	

●リストのコピー(値渡し、参照渡し)

リストのコピーには、代入する方法とcopyメソッドを使用する方法があります。
2つのコピーのは、参照渡しと値渡しといって違いがあります。

・参照渡し(「=」で代入)

コピー先リスト　＝　元リスト

参照渡しはコピー元の変数を参照します。コピー元の変数のショートカットを作成するようなものです。新しく作成されたリストの内容を変更すると、参照元のリストも変更されます。

	構文	備考
1	lis = [1,2,3,4,5,6,7,8,9]	
2	**tmp = lis**	**tmpに参照渡し**
3	tmp[0] = 100	コピーしたtmp[0]を100に変更
4	print(f"lis:{lis}\tアドレス番地:{id(lis)}")	「lis」の内容も変更されていること、「lis」と「tmp」のアドレスが同じであることを確認
5	print(f"lis:{tmp}\tアドレス番地:{id(tmp)}")	

参照渡しのため、メモリのアドレス番地が同じ。
参照渡しのため、コピー先の値を変更すると、元のリストも変更されている。

・値渡し(「copyメソッド」使用)

コピー先リスト　＝　元リスト.copy()

値渡しはコピー元の変数の値を渡します。渡された値で新しく変数を宣言します。新しく作成されたリストは元リストとは全く別の変数になるため、新しいリストを変更しても、参照元のリストは変更されません。

	構文	備考
1	lis = [1,2,3,4,5,6,7,8,9]	
2	**tmp = lis.copy()**	**copy関数を使用し、値渡し**
3	tmp[0] = 100	コピーしたtmp[0]を100に変更
4	print(f"lis:{lis}\tアドレス番地:{id(lis)}")	「lis」と「tmp」のアドレスが違うことを確認
5	print(f"lis:{tmp}\tアドレス番地:{id(tmp)}")	

値渡しのため、メモリのアドレス番地が違う。
値渡しのため、元のリストは変更されない。

●リストを空にする

リストの内容を空にする方法は「lis.clear()」と「lis=[]」の2種類あります。
この2つは同じように見えますが若干の違いがあります。

	構文	表示結果	備考
1	original_lis = [1,3,5,7]		
2	lis1 = orijinal_lis		元リストをlis1に参照渡し
3	lis2 = orijinal_lis		元リストをlis2に参照渡し
4	lis1 = []		[]で空にする
5	print(f"{orijinal_lis},{lis1}")	[1, 3, 5, 7],[]	orijinal_lisはクリアされていない
6	lis2.clear()		
7	print(f"{orijinal_lis},{fis2}")	[],[]	

「lis2」は「original_lis」を参照渡ししているので、clearメソッドでlis2を削除すると、
「original_lis」の中身も削除されます(7行目)。
同じように「lis1」も「original_lis」を参照渡ししているので、「original_lis」も削除されるはずですが、削除されていません(5行目)。
4行目の「lis1 = []」という操作は、メモリ上にある「lis1」を一旦削除し、新しく「lis1」を作り直し（宣言のやり直し）ています。

	構文	備考
1	lis = [1,3,5,7]	
2	print(f"lis:{lis}\tアドレス番地:{id(lis)}")	
3	lis = []	
4	print(f"lis:{lis}\tアドレス番地:{id(lis)}")	アドレス番地が変わっている事を確認

「lis」を新しく作り直しているため、アドレス番地が変わります。

●文字列を区切ってリストに格納

文字列を区切り文字(「,」カンマなど)で区切ってリストに格納します。

リスト = 文字列.split(区切り文字)

	構文	表示結果	備考
1	csv_str = "リンゴ,バナナ,リンゴ,パイナップル"		
2	lis = csv_str.split(",")		カンマ「,」で区切ってリストに格納
3	print(str(lis))	['リンゴ', 'バナナ', 'リンゴ', 'パイナップル']	

●リストを結合して文字列に変更

リストの中身を結合し、文字列に変更します。

変数 = 区切文字.join(リスト)

	構文	表示結果	備考
1	lis = ['リンゴ', 'バナナ', 'リンゴ', 'パイナップル']		
2	s1 = ",".join(lis)		
3	s2 = "".join(lis)		
4	print(f"カンマ付:{s1}")	カンマ付:リンゴ,バナナ,リンゴ,パイナップル	
5	print(f"無し:{s2}")	無し:リンゴバナナリンゴパイナップル	

tuple（タプル型）

●タプルの作成と表示

タプル変数名　＝　（タプル内容）

タプルは、値を代入できないリストです。タプルの値を変更しようとしたり、空のタプルを宣言するとエラーになります。タプルは( )マルカッコで囲んで作成します。

	構文	表示結果	備考
1	t = ('リンゴ', 'バナナ','リンゴ', 'パイナップル')		
2	print(type(t),t)	<class 'tuple'> ('リンゴ', 'バナナ', 'リンゴ', 'パイナップル')	
3	print(t[0])	リンゴ	
4	t[0] = "ゴリラ"		エラー
5	t2 = 1,2,3,4,5,6,7		( )を省略してもタプルになる
6	print(type(t2))	<class 'tuple'>	
7	print(t2[-2])	6	
8	print(t2[2:5])	(3, 4, 5)	スライスが使用できる
9	print(t.index("バナナ"))	1	indexも使用できる
10	print(t.count("リンゴ"))	2	
11	new_taple = (1,2,3)+(4,5,6)		結合して新しタプルを作成できる
12	print(f"結合:{new_taple}")	結合:(1, 2, 3, 4, 5, 6)	

●アンパッキング

タプルやリストの中身を複数の変数に格納する事ができます。この操作の事をアンパッキングと言います。

	構文	表示結果	備考
1	t3 = (10,20,30)		
2	x,y,z = t3		
3	print("アンパッキング：",x,y,z)	アンパッキング：　10 20 30	
4	a,b = t3		エラー

要素数と格納する変数の数が一致していないとエラーになります。

dict（辞書型）

●辞書型の作成と表示

辞書型変数名　＝　{キー1：値1，キー2：値2}

辞書型はキーと値(バリュー)をセットで格納するデータ構造です。キーに対して値が紐づいています。リストやタプルのようにインデックス番号で値を取り出すのではなく、キーから値をとりだします。

1	dic = {"RX-78":"ガンダム","RX-75":"ガンタンク"}	
2	print(type(dic),dic)	
3	print(dic["RX-75"])	キーで値を取り出す
4	dic["RGM-79"] = "ジム"	辞書に追加　（変数名["キー"] = 値）
5	print(dic)	
6	dic["RGM-79"] = "ジム(WD)"	上書き　（変数名["キー"] = 変更値）
7	print(dic)	
8	dic[100]="百式"	キーは数字でもかまわない
9	print(dic)	
10	aa = dict(a = 10,b=20)	**辞書型変数名　＝ dict(キー1=値1，キー2 =値2)**
11	print(aa)	
12	bb = dict([('a',10),('b',20)])	**辞書型変数名　＝ dict([(キー1,値1)，(キー2 ,値2)])**
13	print(bb)	

実行結果

2	<class 'dict'> {'RX-78': 'ガンダム', 'RX-75': 'ガンタンク'}
3	ガンタンク
5	{'RX-78': 'ガンダム', 'RX-75': 'ガンタンク', 'RGM-79': 'ジム'}
7	{'RX-78': 'ガンダム', 'RX-75': 'ガンタンク', 'RGM-79': 'ジム(WD)'}
9	{'RX-78': 'ガンダム', 'RX-75': 'ガンタンク', 'GM-79': 'ジム(WD)', 100: '百式'}
11	{'a': 10, 'b': 20}
13	{'a': 10, 'b': 20}

●辞書型のメソッド

1	dic = {'RX-78': 'ガンダム', 'RX-75': 'ガンタンク', 'RGM-79': 'ジム'}	
2	print(dic.keys())	キーのみ取得
3	print(dic.values())	値のみ取得
4	dic2 = {'RGM-79': 'ジム(WD)', 100: '百式'}	
5	dic.update(dic2)	dic2でdicを更新(「RGM-79」の値がdic2の内容に更新される)
6	print(dic)	
7	print(dic.get("RX-75"))	dic["RX-75"]と同じ
8	print(dic.get(200))	dic[200]だとエラーになる
9	x = dic.pop(100)	dicから「100："百式"」を取り出してxに値を格納
10	print(x,dic)	
11	del dic["RGM-79"]	
12	print(dic)	
13	dic.clear()	辞書の中身を空にする **変数.clear()**
14	print(dic)	

実行結果

2	dict_keys(['RX-78', 'RX-75', 'RGM-79'])	
3	dict_values(['ガンダム', 'ガンタンク', 'ジム'])	
6	{'RX-78': 'ガンダム', 'RX-75': 'ガンタンク', 'RGM-79': 'ジム(WD)', 100: '百式'}	
7	ガンタンク	
8	None	getで値を取得するときはデータが存在しなくてもエラーにならない
10	百式 {'RX-78': 'ガンダム', 'RX-75': 'ガンタンク', 'RGM-79': 'ジム(WD)'}	
12	{'RX-78': 'ガンダム', 'RX-75': 'ガンタンク'}	
14	{ }	変数dicの中身が空になる

●辞書型のコピー(参照渡しと値渡し)

1	dic = {'RX-78': 'ガンダム', 'RX-75': 'ガンタンク', 'RGM-79': 'ジム'}	
2	dic2 = dic	参照渡し
3	dic2 = dic.copy()	値渡し

set（集合型）

●集合型の作成と表示

変数名 = { 要素1,要素2,・・・・}

集合は「重複した値をもたない」、「インデックスをもたない」という特徴を持っています。
Javaのコレクションの「set」と同じようなものです。

1	set_1 = {1,2,2,2,3,4,4,5,6}	
2	print(type(set_1),set_1)	
3	set_2 = {2,3,3,6,7}	
4	print(set_1 - set_2)	set1からset2の集合の値が引かれたものが表示される
5	print(set_1 & set_2)	set1とset2の共通の値が表示される
6	print(set_1 | set_2)	set1またはset2にあるものが表示される
7	print(set_1 ^ set_2)	set1,set2どちらか一方にだけあるもの

実行結果

2	<class 'set'> {1, 2, 3, 4, 5, 6}	
3	{1, 4, 5}	
5	{2, 3, 6}	
6	{1, 2, 3, 4, 5, 6, 7}	
7	{1, 4, 5, 7}	

●集合型のメソッド

1	set_a = {1,2,3,4,5}	
2	set_a.add(6)	集合に値を追加する **変数.add(値)**
3	print(set_a)	
4	set_a.remove(6)	集合から値を削除する **変数.remove(値)**
5	print(set_a)	
6	set_a.clear()	集合の中身を空にする **変数.clear()**
7	print(set_a)	

実行結果

3	{1, 2, 3, 4, 5, 6}	
5	{1, 2, 3, 4, 5}	
7	set()	

Lesson_3　制御フロー

条件判断

●if文

if 条件：
　処理

if文の初めの行の最後に「：」コロンを記述し、処理はインデントを開けます。インデントを戻すまでがif文の処理になります。インデントはタブでもスペースでもどちらでも問題ありません。が一般的にはタブ(スペース4つ)を使用します。

1	x = input("数値を入力してください")	
2	if int(x) >= 10:	
3	print("入力値は10以上です")	必ずインデントを開ける

実行結果

2	数値を入力してください 10	10を入力した場合
3	入力値は10以上です	

●if、else文

C言語やJavaと同様に「else」が使用できます。「else」の後にも「：」を記述します。
「else」の記述は「if」と同じインデントに記述します。

1	x = input("数値を入力してください")	
2	if int(x) >= 10:	
3	print("入力値は10以上です")	必ずインデントを開ける
4	else:	
5	print("入力値は10より小さいです")	

実行結果

2	数値を入力してください 5	5を入力した場合
3	入力値は10より小さいです	

if文にif文をネストする際は、もう一つインデントを下げます。
pythonでは、ブロック({ })を使用しないので、インデントでif文のブロックを識別しますので、インデントの有無を意識して記述してください。

●if、elif文

C言語やJavaでは「else if」を使用しますが、pythonでは「elif」を使用します。特に使い方に差はありません。

```
1 x = input("数値を入力してください")
2 if int(x) < 0:
3     print("負の数が入力されました")
4 elif int(x)==0:
5     print("ゼロが入力されました")
6 else:
7     print("入力値は生の整数です")
```

実行結果

```
2 数値を入力してください0   0を入力した場合
3 ゼロが入力されました
```

●if文、ネスト

C言語やJavaではインデントはあまり気にしなくてよいですが、pythonではインデントが違っているだけで結果が違ってきます。

```
1 x= input("学科の点数と実技の点数をカンマ区切りで入力しなさい")
2 a,b =x.split(",")
3 if int(a) > 60:
4     if int(b) > 60:                インデントを１つ開ける
5         print("学科、実技ともに合格点です")   インデントを２つ開ける
6     else:
7         print("学科は合格点です")
8 else:
9     if int(b) > 60:
10        print("実技は合格点です")
```

●三項演算子

真の時の処理　if 条件　else　偽の時の処理

if文を一行で記述することができます。

```
1 x = input("数値を入力してください")
2 print("10以上です" if int(x) >= 10 else "10より小さいです")
```

●比較演算子と論理演算子

1	a,b = 10,20	
2	print(a==b)	aとbが等しい
3	print(a!=b)	aとbが等しくない
4	print(a<b)	大小
5	print(a<=b)	以上、以下
6	print(a>10 and b>10)	どちらの条件も満たす
7	print(a>10 or b>10)	最低でも、どちらか一方の条件を満たしている

実行結果

2	FALSE	
3	TRUE	
4	TRUE	
5	TRUE	
6	FALSE	
7	TRUE	

●inとnot

1	x,y = 1,[1,2,3]	
2	print(x in y)	y に x が含まれている
3	print(100 not in y)	y に100が含まれていない
4	a,b = 10,20	
5	print(not a==b)	a!=bと同じため、この表現はあまりつかわれない

実行結果

2	TRUE	
3	TRUE	
5	TRUE	

繰り返し

●while文

while 条件：
　処理

条件を満たす間処理を繰り返す。

1	num = 0	
2	while num < 5:	numが５より小さい間繰り返す
3	print(num)	
4	num += 1	

実行結果

	0	numが０から4までの間繰り返される
	1	
	2	
	3	
	4	

●break

1	num = 0	
2	while True:	無限ループ
3	if num >= 5:	numが5になったらbreakでwhileを抜ける
4	break	
5	print(num)	
6	num += 1	

実行結果

	0	
	1	
	2	
	3	
	4	

●for文(拡張for文)

for 変数 in リスト：
処理

pythonで通常よく使われるのが拡張for文です。リストから一つずつ抜き出して変数に格納してリストの中身がなくなるまで繰り返し処理が行われます。

1	lis = ["リンゴ","ゴリラ","ラッパ","パン","パンダ"]	
2	for a in lis:	
	print(a)	
3	if a[-1] == "ン":	
4	print("末尾に「ん」が付きました")	
5	break	
7	else:	for文に対する「else」
8	print("正常終了")	for文を「break」で抜けた場合は実行されない

実行結果

	リンゴ	「break」でfor文を抜けたため、「else」が実行されていない。 「else」はデバックなどで使用する場合があるが、通常はあまり使わない。while文でも「else」は使用できる
	ゴリラ	
	ラッパ	
	末尾に「ん」が付きました	

●for文(range関数)

for 変数 in range(引数1,引数2,引数3)：
処理

回数を決めたループ処理を行うこともできます。引数1：初期値、引数2：終了条件、
引数3：step

1	for i in range(1,5,1):	iは1から始まり、１つずつ増え、5になると処理を終了
2	print(f"{i}回目")	

実行結果

	1回目	
	2回目	
	3回目	
	4回目	

●enumerate関数

for 変数1,変数2 in enumerate(リスト,カウンター開始番号) :
　処理

拡張for文では、リストの内容だけしか取り出せませんでしたが、カウンターも同時に作成したい場合もあります。その時にenumerate関数を使用すると、リストの内容取得とカウンターを作成することができます。変数1：カウンター、変数2：リスト内容が格納されます。

```
1 for i,cont in enumerate (["リンゴ","ゴリラ","ラッパ"]):
2     print(i,cont)
```

実行結果

```
0 リンゴ
1 ゴリラ
2 ラッパ
```

開始番号を省略すると0から始まる

●zip関数

for 変数1,変数2 in zip(リスト1,リスト2) :
　処理

複数のリストをまとめて処理することができます。

```
1 days = ["月曜","火曜","水曜"]
2 foods = ["天ぷら","焼肉","寿司"]
3 drinks = ["日本酒","ビール","ハイボール"]
4 for a,b,c in zip(days,foods,drinks):
5     print(f"{a}日は、{b}を食べ、{c}を飲む")
```

実行結果

```
月曜日は、天ぷらを食べ、日本酒を飲む
火曜日は、焼肉を食べ、ビールを飲む
水曜日は、寿司を食べ、ハイボールを飲む
```

●辞書をfor文で処理

for 変数1,変数2 in 辞書.items() :
**　処理**

辞書型の変数もループで処理することができます。

```
1 dic = {"承太郎":"オラオララッ","ディオ":"無駄無駄無駄ッ",}
2 for k,v in dic.items():
3     print(f"{k}が無意識に叫んでいる言葉、「{v}」")
```

実行結果

```
承太郎が無意識に叫んでいる言葉、「オラオララッ」
ディオが無意識に叫んでいる言葉、「無駄無駄無駄ッ」
```

辞書.items()は辞書型の持つメソッド、辞書のキーと値をタプルで返す。

●enumerateと組み合わせる

●enumerateとzipの組み合わせ。

for カウンター,(変数1,変数2) in enumerate(
zip(リスト1,リスト2),カウンター開始番号) :

```
1 name_lis = ["佐藤","田中","竹田","湯川"]
2 score_lis= [90,70,80,100]
3 for num(name,score) in enumerate(zip(name_lis,score_lis),1):
4     print(f"出席番号{num}番の、{name}さんは、{score}点です。")
```

●enumerateとdic.items()の組み合わせ
を同時に使用することができます。

for カウンター,(変数1,変数2) in enumerate(
dic.items(),カウンター開始番号) :

```
1 data_dic = {"佐藤":90,"田中":70,"竹田":80,"湯川":100}
2 for num(name,score) in enumerate(data_dic.items(),1):
3     print(f"出席番号{num}番の、{name}さんは、{score}点です。")
```

Lesson_４　関数と例外処理

関数の定義と呼び出し

●関数

def 関数名():
**　処理**

関数名()　　#呼び出し

関数を定義するときは、「def」と書き関数名():を書きます。関数を使用する際は、関数名()で呼び出します。スクリプト言語は上から順次処理していきますので、関数は先に宣言してください。

1	def hw():	
2	print("Hello World!")	
3	return "こんにちわ"	戻り値の変数型も指定しなくてよい
4		
5	s = hw()	
6	print(s)	

実行結果

5	Hello World!	
6	こんにちわ	

戻り値が複数あってもreturnで返すことができる。

1	def hw():	
2	print("Hello World!")	
	num=5	
3	return "こんにちわ!",num	戻り値の変数型も指定しなくてよい
4		
5	s,n = hw()	s=hw()でもよい。sはタプルになる
6	print(s*n)	こんにちわが5回表示される

●引数(型宣言)

```
1 def vegetable(color):
2     if color == "red":
3         return "トマト"
4     elif color == "green":
5         return "ピーマン"
6     else:
7         return "いも"
8 ve = vegetable("red")
  print(ve)
```

def vegetable(color : str) -> str:
python3.6以降では、上記のように変数の型を宣言できます。

実行結果
トマト

●引数複数(位置引数)

```
1 def cal(a:int,b:int,c:int) -> int:
2     if c == 0:
3         return a+b
4     elif c == 1:
5         return a-b
6     else:
7         return a*b
8 n = cal(3,5,3)
9 print(n)
```

順番通りに引数の値が変数に代入される

実行結果
15

●引数複数(キーワード引数)

```
8 n = cal(a=3,c=3,b=5)
9 print(n)
```

「引数名=値」と記入することで、順番が違っても正しい値が出力される

実行結果
15

●デフォルト引数

```
1 def hw(s="名無し"):
2     print(f"{s}さん、こんにちわ")
3
4 hw()
5 hw("田中")
```

引数が省略されたときに、設定される値を指定

実行結果
名無しさん、こんにちわ
田中さん、こんにちわ

●引数複数(可変)

1	def hw(*args):	引数を複数可変型で指定できます 引数の変数名の前に「*」を指定します 引数の型はタプルになります 実行結果 田中さん , 斎藤さん , 武田さん , こんにちわ 。
2	for a in args:	
3	print(str(a)+"さん",end=' , ')	
4	print("こんにちわ", end=' 。')	
5		
6	hw("田中","斎藤","武田")	

●アンパッキング

6	names =("田中","斎藤","武田")	タプルを作成し、引数に使用することもできます、その場合は引数の前に「*」を付けアンパッキングします
7	nhw(*names)	

●引数(キーワード引数：辞書型)

1	def price(**kwargs):	引数に「**」を付けることで、辞書型の引数を受け取ることができます。 実行結果 リンゴ、200円 みかん、180円 パイナップル、450円
2	for k,v in kwargs.items():	
3	print(f"{k}、{v}円")	
4		
5	dic ={"リンゴ":200,"みかん":180,"パイナップル":450}	
6	price(**dic)	

上記の引数の使用方法は、複数同時に使用することができます、いろいろ試してみてください。

●docstring

"""

説明文

"""

print(関数名__doc__)

docstringは関数の説明を記述した文章です。関数の中に「"」ダブルクォート3つで囲んで書きます。記述した関数の説明は、__doc__で確認できます。

関数の応用

●関数内関数

関数内関数とは、関数の中に定義された関数のことです。関数の中だけで繰り返し使う処理がある場合に作成します。

1	def outer(a,b):	
2	def plus(c,d):	outer関数内で関数を宣言
3	return c + d	
4	r = plus(a,b)	outer関数で関数内関数を使用
5	return r	リテラルをリターン
6	ans=outer(1,2)	
7	print(ans)	3

●クロージャー

外側の関数に引き渡した値を関数内関数が保持することができるクロージャー

1	def outer(a):	エンクロージャー
2	def inner(b):	クロージャー
3	return a+b	
4	return inner	オブジェクトをリターン
6	f1 = outer(1)	f1にはinnerオブジェクト(1を内包)が代入
7	f2 = outer(10)	f2にはinnerオブジェクト(10を内包)が代入
8	print(f1(5))	6　(引数5 + 内包値1)
9	print(f2(5))	15　(引数5 + 内包値10)

f1とf2は、与えられた引数を内包することによって結果が異なる。このクロージャーとは簡易的なクラスと考えるとよい。

f1やf2は、6行目、7行目の時点では、innerが代入されただけです、f1()、f2()と「()」をつけると、innerが実行されます。

●lambda

def 関数名(引数1,引数2,…):
**　return 式**
↓
関数名=lambda 引数1,引数2,…:式

lambdaは短い関数を簡潔に定義することができます。

・lambdaを使わない場合

1	**def func(s:str):**	
2	**　return s.capitalize()**	引数の先頭文字を大文字にして返す
3		
4	lis=["Mon","tue","wed","Thu","fri","sat","Sun"]	先頭文字を全て大文字にしたい
5	for a in lis:	
6	print(func(a))	

・lambdaを使用した場合

1	lis=["Mon","tue","wed","Thu","fri","sat","Sun"]	
2	**func = lambda s: s.capitalize()**	関数を 1 行で記述
3	for a in lis:	
4	print(func(a))	通常の関数のように呼び出す

例題）

	lambda無し	lambda使用
1	def cal(a,b,c):	cal = lambda a,b,c:(a+b)*c print(cal(2,3,5))
2	return (a+b)*c	
3		
4	print(cal(2,3,5))	
1	def pay(num):	
2	if num <=12:	
3	return 600	
4	else:	
5	return 1200	
6	num = int(input("年齢を入力してください"))	
7	print(f"料金は、{pay(num)}円です")	

内包表記

●リスト内包表記

リスト＝[変数　for　変数　in　リスト]

リスト内包表記は、既存のリストから新しいリストを作ることができるリスト作成方法の一つです。通常for文を使用して複数行で記述しますが、リスト内包表記を使用すると一行で記述することができます。

1	t = (1,2,3,4,5)	
2	r1 = []	ループ処理
3	for i in t:	
4	r1.append(i)	
5	print(r1)	
6		
7	r2 = [i for i in t]	リスト内包表記
8	print(r2)	

●辞書包括表記

辞書型＝[キー：値　for　キー,値　in　zip(リスト1,リスト2)]

リスト内包表記と同様に、既存のリストを使用し、辞書を一行で作成することができます。

1	lis1=["リンゴ","ミカン","ナシ","ブドウ"]	
2	lis2=[250,100,300,500]	
3	dic = {x:y for x,y in zip(lis1,lis2)}	辞書包括表記
4	print(dic)	

●集合内包表記

集合＝{変数　for　変数　in　range()}

同様に集合も内包表記で記述できます

1	s = {i for i in range(10)}	
2	print(s)	

内包表記は、ループ条件などで使用されたりします。pythonでは頻繁に使用されていますので、練習して慣れてください。

変数の有効範囲

●グローバル変数

global　変数

グローバル変数は、pyhonでも使用されます。インデントがない状態で宣言された変数はグローバル変数となりますが、関数内で使用する際は明示的にグローバルであることを宣言しないといけません。

1	num = 10	グローバル変数
2	def func():	
3	global num	グローバル変数を使用する宣言
4	print(f"変数内num：{num}")	
5	num += 10	
6		
7	print(f"メインnum：{num}")	
8	func()	
9	print(f"func実行後num：{num}")	

ローカル変数、グローバル変数の確認方法

1	num = 10	
2	def func():	
3	s1 = "こんにちわ"	
4	print(f"ローカル変数：{locals()}")	
5		pythonがあらかじめ作成しているグローバル変数も表示される
6	print(f"グローバル変数：{globals()}")	
7	func()	

例外処理

●例外処理

try:
**　　処理**
except　エラー　as　変数:　(「as 変数」は省略可能)
**　　処理　　(変数にエラー内容が格納される)**

javaのtry-catch文と同じような形式で、エラーを処理することができます。どのエラーが発生するかわからない時は、上位階層の「Exception」を処理すると意図しないエラーでも対処することができます。が、pythonでは、よくわからないエラーもすべてキャッチして次のコードに進むという処理は推奨されていませんので、「Exception」ですべての処理をするというコードはあまり書かないほうがよいです。

1	lis=[1,2,3]	
2	i=int(input("数字を入力"))	リストの要素番号外を入力するとIndexErrorが発生
3	#del lis	リストを削除するとNameErroが発生
4	try:	
5	print(lis[i])	インデックスエラーをキャッチする
6	except IndexError as ex:	
7	print(f"エラー：{ex}")	
8	except NameError as ex:	
9	print(f"エラー：{ex}")	

●finally

finaryで書いた処理は、エラーが出ても出なくても必ず実行されます。

1	lis=[1,2,3]	
2	i=int(input("数字を入力"))	正常終了とエラーどちらも出して確認してみる
3	try:	
4	print(lis[i])	インデックスエラーをキャッチする
5	except IndexError as ex:	
6	print(f"エラー：{ex}")	
7	finally:	
8	print("プログラムを終了します")	

●else

elseで書いた処理は、エラーが発生しなかったときのみ実行されます。

1	lis=[1,2,3]	
2	i=int(input("数字を入力"))	正常終了とエラーどちらも出して確認してみる
3	try:	
4	print(lis[i])	インデックスエラーをキャッチする
5	except IndexError as ex:	
6	print(f"エラー：{ex}")	
7	else:	
8	print("正常終了")	
9	finally:	
10	print("プログラムを終了します")	

●raise

raise　エラークラス(エラー詳細):

raiseは例外を発生させることができます。自作の例外クラスや、標準の例外クラスなどを発生させることができます。

1	pw=input("8文字以上でパスワードを設定してください")	
2	try:	
3	if len(pw) < 8:	
4	raise IndexError("文字数が足りません")	例外を発生
5	except IndexError as ex:	
6	print(f"エラー：{ex}")	
7	else:	
8	print("正常終了")	
9	finally:	
10	print("プログラムを終了します")	

Lesson_5　モジュールとパッケージ

import文

●パッケージの作成

pythonではimportできるファイルをパッケージと呼びます。パッケージを読み込むことで、パッケージ内で書かかれた、関数やクラスなどを使うことができます。
パッケージは以下の内容で構成されます。

パッケージ　　：モジュールを格納するディレクトリ
　┣　モジュール　　：pythonのコードが記述されたファイル
　┗　__init__　　：パッケージのインポート時に初期化するための処理が書かれているファイル。初期化が必要ない場合は何も書かなくてよい

パッケージの作成		
1	プロジェクト名右クリック→「新規」 →「ディレクトリ」→パッケージ名を記入	今回は「lesson_pk」
2	作成したディレクトリを右クリック→「新規」 →pythonファイル→モジュール名を記入	今回は「utils」
3	作成したディレクトリを右クリック→「新規」 →pythonファイル→「__init__」を記入	「__init__」がないとパケージとして認識しない(Python3.3以降は無くてもよい)

・utils.py

1	def hw(s):	
2	return s + "さん、こんにちわ"	

●別のパッケージからutilsモジュールをインポート

import　パッケージ名.モジュール名

1	import lesson_pk.utils	
2	r=lesson_pk.utils.hw("田中")	パッケージ名.モジュール名.関数名()
3	print(r)	

●読み込み時にfromを使用してインポート

from　パッケージ名　import　モジュール名

1	from lesson_pk import utils	
2	r=utils.hw("田中")	モジュール名.関数名()
3	print(r)	

●asを使用してモジュール名を変更してインポート

from　パッケージ名　import　モジュール名　as　仮名

モジュール名に一時的な仮の名前を付けて使用することができます。

1	from lesson_pk import utils as a	
2	r=a.hw("田中")	モジュール変更名.関数名()
3	print(r)	

●パッケージ内の複数のモジュールをインポート

from　パッケージ名　import　＊

パッケージ内の複数のモジュールをインポートする際は、上記のようにアスタリスクを使用しますが、そのままでは使用できません。「__init__」に「__all__=リスト」を指定する必要があります。ただpythonでは推奨されていないので具体例は書きません。インポート文にアスタリスクが含まれている場合は、同じパッケージにある複数のモジュールをインポートしているということだけわかるようにしておいてください。

パッケージの配布

●パッケージ出力

pycharmの機能を使用し、パッケージを配布可能な形で作成することができます。

パッケージ出力		
1	プロジェクト名選択→「ツール」 →「setup.pyの作成」	
2	表示される一覧から「sdist」をダブルクリック→コマンドは入力せずに「ok」をクリック	
3	バージョンなどの情報を記載	
4	プロジェクト名選択→「ツール」 →「setup.pyの実行」	

作成されたファイルは「C:\Users\ユーザ名\PycharmProjects\pythonProject\dist」に作成されます。

●pyinstallerを使用した実行ファイル作成

pyinstallerを使用することによって実行ファイル(exe)を作成することができます。プログラムを実行ファイルにするとpython開発環境が整っていないPCでもプログラムを実行することができます。

・環境構築

pyinstallerのインストール		
1	コマンドプロンプトを起動	
3	py -m pip install pyinstaller	pipでpyinstallerをインストール
4	pyinstallerがインストールされていることを確認 C:\Users\ユーザ名\AppData\Local\Programs\Python\Python312\Scripts\	
環境変数pathに追加		
1	左下のスタータボタンを右クリック→「システム」 →「システムの詳細設定」	
2	「システムのプロパティ」が表示される 「詳細設定」→「環境変数」	
3	「環境変数」が表示される 「システム環境変数」→「path」を選択し「編集」	
4	「環境変数名の編集」 「新規」→pyinstallerのインストール場所を記入	
5	C:\Users\ユーザ名\AppData\Local\Programs\Python\Python312\Scripts\	
6	「ok」ボタンですべての画面を閉じる	

・実行ファイル作成

カレントディレクトリ変更		
1	コマンドプロンプトを起動	exeファイル化したいパッケージまで移動
2	cd C:\Users\ユーザ名\PycharmProjects\pythonProject\lesson_pk	
exeファイル化		
1	pyinstaller utils.py --onefile	-F でも良い
2	「dist」ファイルが作成されその中に、exeファイルが作成される	
4	--noconsole：コンソール無し	-w でも良い
5	--version-file=[バージョン情報を記載した.txt]：バージョン情報	

アプリのプロパティにバージョン情報などを記載する場合は、バージョン情報を記載したファイルを一緒にコンパイルすること、バージョン情報の記述フォーマットは別途記載する。

ライブラリ

●組み込み関数

組み込み関数とは、あらかじめ用意され標準で使用することができる関数のことです。インポートしなくても使える関数のことです。
組み込み関数は「builtins」に記述されています。
インポート無しに使える関数は組み込み関数です。以下の関数以外にもたくさんあります。

●標準ライブラリ

標準ライブラリとは、インポートすることで使用できるようになります。
標準ライブラリの一覧は公式ドキュメントから確認できます。
(https://docs.python.org/3/library/)　日本語：(https://docs.python.org/ja/3/library/)

1	from tkinter import *	
2	root = Tk()	
3	root.mainloop()	

標準ライブラリは、インポートするだけで使用することができるようになります。

●サードパティのライブラリ

P34で行ったように、パッケージを配布することができます。また、他の人が作ったライブラリを使用することもできます。
P35でコンソールから「pip」コマンドで「pyinstaller」をインストールしましたが、それと同じことがPyCharmの機能でできます。

pyinstallerのインストール		
1	左上「設定」→「プロジェクト」→「インタプリタ」	
3	「+」→インストールしたいライブラリを検索	
4	パッケージのインストール	
環境変数pathに追加		
1	環境変数の設定が必要な場合もある	

●ライブラリをインポートするときの注意点

標準ライブラリをインポートする際、「,」カンマでつなげて1行でインポートすることができますが、推奨されていないので、複数行でインポートするようにしましょう。
ライブラリをインポートする際は、初めに標準ライブラリをインポートし、一行空けてからサードパティのライブラリをインポートし、自作のライブラリがあれば、さらに一行空けてインポートしましょう。絶対的なルールではありませんが、なるべくならこのような順番で書きましょう。

1	import os,sys,tkinter	使用できるが推奨されていない
2		
3	import os	１行ずつ複数行書くほうが良い。 アルファベット順
4	import sys	
5	import tkinter	
6		サードパーティのライブラリは、標準関数の後に一行開けて記述
7	import termcolor	
8		自作のライブラリは、サードパティのライブラリの後に一行開けて記述
9	import lesson_pk	

Lesson_6　オブジェクトとクラス

クラス

●クラスの作成

class　クラス名(object):

「class」の後にクラス名を記述します。クラス名の後の()の中には「object」と書きます。
python2ではクラスの後に(object)を記述しないといけませんでしたが、python3では必須ではなくなりました。
(object)の意味は、スーパークラスである「object」を継承しているという意味になります。
python3では、()の中身が空の時はスーパークラスのである「object」を継承しますので、
python3では明示的に宣言しなくてもよくなっています。

1	class Person(object):	クラス宣言
2	def say_something(self):	関数宣言
3	print("hello")	
4		
5	pason=Person()	インスタンスの作成
6	pason.sey_something()	インスタンスのメソッドを呼び出す

クラス内で宣言した関数の引数に「self」が入っています。この「self」はオブジェクト(インスタンス)自身を指しています。必ず第一引数に「self」を記述しなければいけません。

●クラス名　命名規則

クラス名をつける際は、「CapWords方式」で名前をつけます。CapWords方式では、各単語の先頭文字を大文字にします。また単語と単語の間に「_」(アンダーバー)を使用しません。

●コンストラクタ(クラスが初期化されるときの処理)

def __init__(self):

インスタンスを作成すると「init」に記述された内容が実行されます。javaのコンストラクタはクラス名と同じものですが、pythonのコンストラクタは「__init__」で記述します。

1	class Person(object):	
2	def __init__(self):	コンストラクタ
3	print("hello")	
4		
5	pason=Person()	initに記述された処理が実行される

●コンストラクタ 引数あり

def __init__(self,引数1,引数2,,,):

1	class Person(object):	
2	def __init__(self,name):	
3	self.name=name	インスタンス変数
4	print(f"{name}さん、hello")	
5		
6	pason=Person("田中")	initに記述された処理が実行される

インスタンス変数には、必ずselfを付けます。
「init」で宣言されたインスタンス変数は値を保持されるため、インスタンス生成後
「self.name」で別のメソッドでも使用することができる

1	class Person(object):	
2	def __init__(self,name="名無し"):	デフォルト引数を指定できる
3	self.name=name	
4	def say_something(self):	
5	print(f"{self.name}さん、hello")	self.nameを使用できる
6		
7	pason=Person()	
8	pason.say_something()	

selfを使用することにより、自身のメソッドを使用できる

1	class Person(object):	
2	def __init__(self,name="名無し"):	
3	self.name=name	
4	def say_something(self):	
5	print(f"{self.name}さん、hello")	
6	self.muda(10)	自身のメソッドを呼び出す
7	def muda(self,num):	
8	print("無駄ッ"*num)	
9		
10	pason=Person()	
11	pason.say_something()	

●デストラクタ(インスタンスが破棄される際に実行される処理)

def __del__(self):

インスタンス作成時に実行される処理を、コンストラクタ。インスタンスが破棄される際に実行される処理を、デストラクタといいます。デストラクタはインスタンスが使われなくなったタイミングで実行されます。

1	class Person(object):	
2	def __init__(self,name="名無し"):	
3	self.name=name	引数を、メンバ変数に格納
4	print("インスタンス生成")	
5	def __del__(self):	
6	print("インスタンスを破棄します")	
7	def call_name(self):	
8	print(self.name)	1
9	parson=Person()	コンストラクタのメッセージが表示される
10	print("#####")	
11	parson.call_name()	
12	print("#####")	インスタンスが破棄されて、デストラクタのメッセージが表示される
13	del parson	

「del pason」と明示的にインスタンスを破棄しなくても、プログラム終了時にインスタンスも破棄されるので、デストラクタは動作する。

●クラスの継承

class　クラス名(スーパークラス):

クラスの継承は、クラス名の後ろの()の中に、スーパークラスの名前を記述するだけで継承することができます。

1	class Car(object):	
2	def run(self):	
3	print("run")	
4	class MyCar(Car):	()に「Car」(スーパークラス名)を記入
5	pass	何もしないクラス
6	car=Car()	
7	car.run()	
8	mycar=MyCar()	
9	mycar.run()	スーパークラスの「run」を継承している

●メソッドのオーバーライド

pythonのクラスもメソッドをオーバーライドすることができます。
下記コードでは、CarクラスのrunをMyCar、SuperCarでオーバーライドしています。

1	class Car(object):	
2	def run(self):	
3	print("run")	
4		
5	class MyCar(Car):	
6	def run(self):	runをオーバーライド
7	print("fast")	
8		
9	class SuperCar(Car):	
10	def run(self):	runをオーバーライド
11	print("super fast")	
12		
13	car=Car()	
14	car.run()	
15	print("###############")	
16	mycar=MyCar()	
17	mycar.run()	
18	print("###############")	
19	scar=SuperCar()	
20	scar.run()	

●superを使用する

superを使用して、スーパークラスのメソッドを使用しようする。super()の後に「.」ドットを付けスーパークラスのメソッドを呼び出す。
下記コードでは、MyCarのinitでスーパークラスのinitを呼び出しています。

1	class Car(object):	
2	def __init__(self,model=None):	
3	self.model=model	
4	def p(self):	
5	print(self.model)	
6		
7	class MyCar(Car):	
8	def __init__(self,model="SUV",drive="4WD"):	
9	super().__init__(model)	スーパークラスのinitを呼び出し
10	self.drive=drive	
11		
12	car=Car()	
13	car.p()	
14	print("###############")	
15	mycar=MyCar()	
16	mycar.p()	

●プロパティを使用する

pythonのクラスのインスタンス変数はセッター無しで書き換えが可能です。

1	class Car(object):	
2	def __init__(self,model=None,drive="4WD"):	
3	self.model,self.drive=model,drive	
5	def p(self):	
6	print(f"モデル：{self.model}、駆動：{self.drive}")	
7	car=Car()	
8	car.p()	
9	car.model="セダン"	インスタンス変数を変更
10	car.drive="2DW"	
11	car.p()	

外部からインスタンス変数の書き換えを制限したい場合は、プロパティを使用します。インスタンス変数の前に「_」アンダーバーを付け、インスタンス変数を返すメソッドを作成し、そのメソッドの前に「@property」を付けます

1	class Car(object):	
2	def __init__(self,model=None,drive="4WD"):	
3	self.__model,self.__drive=model,drive	
4	def p(self):	
5	print(f"モデル：{self.model}、駆動：{self.drive}")	プロパティを参照
6	@property	プロパティ(ゲッター)
7	def model(self):	modelを返すメソッド
8	return self.__model	
9	@property	プロパティ(ゲッター)
10	def drive(self):	driveを返すメソッド
11	return self.__drive	
12	car=Car()	
13	car.p()	
14	car.__model="セダン"	
15	car.p()	変更されていない

プロパティを使用することで、インスタンス変数を変更できなくなりました。プロパティのゲッターを使用することでインスタンス変数の値を変更することができます。

1	class Car(object):	
2	def __init__(self,model=None,drive="4WD"):	
3	self.__model,self.__drive=model,drive	
4	def p(self):	
5	print(f"モデル：{self.model}、駆動：{self.drive}")	
6	@property	プロパティ(ゲッター)
7	def model(self):	
8	return self.__model	
9	@property	プロパティ(ゲッター)
10	def drive(self):	
11	return self.__drive	
12	@model.setter	プロパティ(セッター)
13	def setmodel(self,model):	
14	self.__model=model	
15	@drive.setter	プロパティ(セッター)
16	def setdrive(self,drive):	
17	self.__drive=drive	
18		
19	car=Car()	
20	car.p()	
21	car.setmodel="セダン"	
22	car.setdrive="2DW"	
23	car.p()	

Lesson_7　ファイル操作とシステム

ファイル

●ファイルの作成

ファイルを作成するには、「open」関数でファイルを開きますが、指定されたファイルが存在しない場合は、ファイルを作成します。

オブジェクト名=open("ファイルパス","モード")

モード	内容
"r"	読込のみ
"r+"	読込と書込
"a"	追記モード
"w"	書込(上書き)モード
"w+"	書込と読込

●読込モード

1	f=open("test.txt","r")	読込モードでオープン。
2	print(f.read())	fを読込
3	f.write("本日も晴天なり")	エラー(writeメソッドをもっていないため)
3	f.close()	fをクローズ

●書込モード

1	f=open("test.txt","w")	書き込みモードでオープン。fというオブジェクトを作成
2	f.write("本日は晴天なり")	fに書き込み
3	print(f.read())	エラー(readメソッドをもっていないため)
4	f.close()	fをクローズ

●読込と書込

1	f=open("test.txt","w")	
2	f.write("明日も晴天なり")	
3	f.close()	一旦クロースしないといけない
4	f=open("test.txt","r")	
5	print(f.read())	
6	f.close()	

●r+モード

1	f=open("test.txt","r+")	読込+書込モードでオープン。
2	print(f"読込1：{f.read()}")	fを読込
3	f.write("本日は晴天なり")	
4	f.seek(0)	fをクローズ
5	print(f"読込2：{f.read()}")	追記されている
6	f.close()	

r+モードの書き込みは追記されます。r+でopenした際に、指定のファイルが無い場合はエラーが出ます。

●w+モード

1	f=open("test.txt","w+")	
2	print(f"読込1：{f.read()}")	書込(上書き)モードで開いているので、中身は空の状態
3	f.write("本日は晴天なり")	
4	f.seek(0)	読込位置を先頭に戻す
5	print(f"読込2：{f.read()}")	上書きされている
6	f.close()	

w+モードの場合、ファイルを開いた直後は、上書きモードで開いているので中身が空になっています。

●ファイル内の位置の把握と移動

ファイル内のどの位置にいるのか確認

オブジェクト名.tell( )

ファイル内の指定位置に移動

オブジェクト名.seek(位置番号)

1	f=open("test.txt","r")	
2	print(f.tell())	現在の位置を確認
3	print(f.read())	
4	f.seek(10)	seekを移動する
5	print(f.tell())	移動したため表示が変わることを確認
6	print(f.read())	
7	f.close()	

●with文

ファイルをopenで開いて操作を行った後は必ずcloseでファイルを閉じます。このcloseを忘れると無駄にメモリを消費してしまいます。
withを使用するとこで、closeを記述しなくてもよくなります。openの直前にwithを記述し、操作する行にインデントを開けるだけで、インデントを戻すタイミングでファイルをクローズしてくれます。

with　open(ファイルパス,モード)　as　変数名
**　　操作**

1	with open("test.txt","r+") as f:	オープンしたファイルの中身は変数fに格納される
2	print(f"1回目：{f.read()}")	
3	f.write("withで追加")	
4	f.seek(0)	
5	print(f"2回目：{f.read()}")	
6	print(f"3回目：{f.read()}")	ファイルがクローズ(fがメモリから解放)されているためエラーが表示される

closeの書き忘れなどの心配をしなくてよいので、ファイル操作の際はwith文を使用するほうが良いとされています。

●Template

Template関数は、標準ライブラリのstringをインポートすることで使用できるようになります。
Template関数は、定型文を扱う場合に便利です。例えば定期報告メールや議事録などのある程度のフォーマットが決まっていて、日時や名前などが少し変わるような文章に使用できます。

import string
s="""
定型文の内容(変更部分は$から始まる変数を記載)
"""

t=string.Template(s)
con=t.substitute(変数に変更値を代入)

1	import string	
2	s="""	定型文をコメントブロックで変数sに格納
3	定期報告	
4	お疲れ様です。　$nameです。	変更する部分に変数を入れる
5	本日$dateの定期報告を行います。	変数は＄から始める

6	$cont	
7	以上です。	
8	よろしくお願いします。	
9	"""	
10		
11	t=string.Template(s)	sをテンプレートにする
12	contents=t.substitute(name="谷口",date="12月20日",cont="特に異常はありませんでした。")	テンプレートの変数に変更する値を代入する
13	print(contents)	

テンプレートを使うメリットの一つに元になっている「s」を直接扱わないので原本のフォーマットが保たれるよいう利点もあります。よって定型文を保存しておき、openで開きreadで読み込んだ内容をTemplateの引数にして、substituteで操作するとよいでしょう。
NASにある欠席届ファイルを原本にして欠席届を作成してみましょう。
ファイルパス(\\192.168.10.201\share\python\欠席届.txt)
変数：$name、$date、$cont

1	import string	
2	with open(r"\\192.168.10.201\share\python\欠席届.txt","r") as f:	定型文をコメントブロックで変数sに格納
3	s=f.read()	
4	t=string.Template(s)	変更する部分に変数を入れる
5	contents=t.substitute(name="谷口",date="12月20日",cont="感謝の正拳突きを行っていたため、時間を忘れ電車に乗り遅れてしまいました。")	変数は＄から始める
6	print(contents)	

●CSVファイル作成

CSVとはカンマ区切りのテキストファイルのことです。標準ライブラリのcsvをインポートすると使えます。

```
import　csv
with　open(ファイルパス,"w",newline="")　as　変数1
    変数2=[列名1,列名2,,,,]
    変数3=csv.DictWriter(変数1,変数2)
    変数3.writeheader
    変数3.writerow(内容)
```

1	import csv	
2	with open("test.csv","w",newline="") as csv_f:	
3	head_name=["名前","住所","TEL"]	ヘッダを作成
4	writer=csv.DictWriter(csv_f,head_name)	
5	writer.writeheader()	ヘッダを記入
6	writer.writerow({"名前":"田中宏","住所":"福岡県福岡市","TEL":"0123456789"})	1行ごと記入
7	writer.writerow({"名前":"武田哲也","住所":"福岡県福岡市","TEL":"9876543210"})	
8		
9	with open("test.csv","r") as csv_f:	
10	print(csv_f.read())	

ファイルのopenの第三引数が「newline=""」になっています。これはCSVファイルの改行コードが「\r\n」であるため、２行改行されてしまいます。これを防ぐために「newline=""」を第三引数に記入しています。

●CSVファイル読込

上記の方法(read())でも読込めますが、「DictReader」に格納しループで処理することもできます。

```
import　csv
with　open(ファイルパス,"r")　as　変数1
　　変数2=csv.DictReader(変数1)
```

1	import csv	
2		
3	with open("test.csv","r") as csv_f:	
4	reader=csv.DictReader(csv_f)	
5	for row in reader:	
6	print(row["名前"],row["住所"],row["TEL"])	

システム

●様々なファイル操作

標準ライブラリのosをインポートすることで、様々なファイル操作を実行できます。

import　os

メソッド	内容
os.path.exists(引数)	引数に与えられたファイルが存在するか確認できます
os.path.isfile(引数)	引数にあたあられたものがファイルかどうか確認できます
os.path.isdir(引数)	引数にあたあられたものがディレクトリかどうか確認できます
os.path.rename(引数1,引数2)	引数1を引数2にリネーム
os.mkdir(引数)	ディレクトリを作成する
os.rmdir(引数)	ディレクトリを削除する
os.remove(引数)	ファイルを削除する
os.listdir(引数)	ディレクトリの中にあるディレクトリを列挙できる

・その他

pathlib.Path(引数).touch()	空のファイルを引数に与えられたファイル名で作成(import pathlib)
shutil.copy(引数1,引数2)	引数1のファイルを、引数2という名前でコピーする(import shutil)

1	import os	インポート、本来は三行で書く 行数の問題で今回は二行で書いている
2	import pathlib,shutil	
3	print(os.path.exists("test.txt"))	ファイル存在確認
4	print(os.path.isfile("test.txt"))	ファイルであるか
5	print(os.path.isdir("test.txt"))	ディレクトリであるか
6	os.rename("test.txt","readme.txt")	ファイル名変更
7	os.mkdir("test_dit")	ディレクトリ作成
8	os.rmdir("test_dir")	ディレクトリ削除
9	pathlib.Path("empty.txt").touch()	空のファイル作成
10	os.remove("empty.txt")	ファイル削除
11	print(os.listdir("d:"))	Dドライブのディレクトリ表示
12	shutil.copy("test.txt","test2.txt")	ファイルコピー