

## Designing Data Types

## Outline

- 1 APIs
- 2 Encapsulation
- 3 Immutability
- 4 Polymorphism
- 5 Overloading
- 6 Functions are Objects
- 7 Examples
- 8 Exceptions

## APIs

## APIs

Precisely specifying a data type using an API improves design because it leads to client code that can clearly express its computation

## APIs

Precisely specifying a data type using an API improves design because it leads to client code that can clearly express its computation

By using APIs to separate clients from implementations, we reap the benefits of standard interfaces for every program that we compose

## APIs

Precisely specifying a data type using an API improves design because it leads to client code that can clearly express its computation

By using APIs to separate clients from implementations, we reap the benefits of standard interfaces for every program that we compose

APIs should provide to clients just the methods they need and no others

## Encapsulation

## Encapsulation

The process of separating clients from implementations by hiding information is known as encapsulation



## Encapsulation

The process of separating clients from implementations by hiding information is known as encapsulation

Encapsulation allows one implementation of an API to be substituted for another

## Encapsulation

The process of separating clients from implementations by hiding information is known as encapsulation

Encapsulation allows one implementation of an API to be substituted for another

Encapsulation helps programmers ensure that their code operates as intended

## Encapsulation

The process of separating clients from implementations by hiding information is known as encapsulation

Encapsulation allows one implementation of an API to be substituted for another

Encapsulation helps programmers ensure that their code operates as intended

Python does not enforce encapsulation; instead, through a naming convention, clients are informed that they should not directly access the instance variable, method, or function thus named

## Encapsulation

The process of separating clients from implementations by hiding information is known as encapsulation

Encapsulation allows one implementation of an API to be substituted for another

Encapsulation helps programmers ensure that their code operates as intended

Python does not enforce encapsulation; instead, through a naming convention, clients are informed that they should not directly access the instance variable, method, or function thus named

The API should be the only point of dependence between client and implementation — this is called modular programming

## Immutability

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47	48	49	50	51	52	53	54	55	56	57	58	59	60	61	62	63	64	65	66	67	68	69	70	71	72	73	74	75	76	77	78	79	80	81	82	83	84	85	86	87	88	89	90	91	92	93	94	95	96	97	98	99	100	101	102	103	104	105	106	107	108	109	110	111	112	113	114	115	116	117	118	119	120	121	122	123	124	125	126	127	128	129	130	131	132	133	134	135	136	137	138	139	140	141	142	143	144	145	146	147	148	149	150	151	152	153	154	155	156	157	158	159	160	161	162	163	164	165	166	167	168	169	170	171	172	173	174	175	176	177	178	179	180	181	182	183	184	185	186	187	188	189	190	191	192	193	194	195	196	197	198	199	200	201	202	203	204	205	206	207	208	209	210	211	212	213	214	215	216	217	218	219	220	221	222	223	224	225	226	227	228	229	230	231	232	233	234	235	236	237	238	239	240	241	242	243	244	245	246	247	248	249	250	251	252	253	254	255	256	257	258	259	260	261	262	263	264	265	266	267	268	269	270	271	272	273	274	275	276	277	278	279	280	281	282	283	284	285	286	287	288	289	290	291	292	293	294	295	296	297	298	299	300	301	302	303	304	305	306	307	308	309	310	311	312	313	314	315	316	317	318	319	320	321	322	323	324	325	326	327	328	329	330	331	332	333	334	335	336	337	338	339	340	341	342	343	344	345	346	347	348	349	350	351	352	353	354	355	356	357	358	359	360	361	362	363	364	365	366	367	368	369	370	371	372	373	374	375	376	377	378	379	380	381	382	383	384	385	386	387	388	389	390	391	392	393	394	395	396	397	398	399	400	401	402	403	404	405	406	407	408	409	410	411	412	413	414	415	416	417	418	419	420	421	422	423	424	425	426	427	428	429	430	431	432	433	434	435	436	437	438	439	440	441	442	443	444	445	446	447	448	449	450	451	452	453	454	455	456	457	458	459	460	461	462	463	464	465	466	467	468	469	470	471	472	473	474	475	476	477	478	479	480	481	482	483	484	485	486	487	488	489	490	491	492	493	494	495	496	497	498	499	500	501	502	503	504	505	506	507	508	509	510	511	512	513	514	515	516	517	518	519	520	521	522	523	524	525	526	527	528	529	530	531	532	533	534	535	536	537	538	539	540	541	542	543	544	545	546	547	548	549	550	551	552	553	554	555	556	557	558	559	560	561	562	563	564	565	566	567	568	569	570	571	572	573	574	575	576	577	578	579	580	581	582	583	584	585	586	587	588	589	590	591	592	593	594	595	596	597	598	599	600	601	602	603	604	605	606	607	608	609	610	611	612	613	614	615	616	617	618	619	620	621	622	623	624	625	626	627	628	629	630	631	632	633	634	635	636	637	638	639	640	641	642	643	644	645	646	647	648	649	650	651	652	653	654	655	656	657	658	659	660	661	662	663	664	665	666	667	668	669	670	671	672	673	674	675	676	677	678	679	680	681	682	683	684	685	686	687	688	689	690	691	692	693	694	695	696	697	698	699	700	701	702	703	704	705	706	707	708	709	710	711	712	713	714	715	716	717	718	719	720	721	722	723	724	725	726	727	728	729	730	731	732	733	734	735	736	737	738	739	740	741	742	743	744	745	746	747	748	749	750	751	752	753	754	755	756	757	758	759	760	761	762	763	764	765	766	767	768	769	770	771	772	773	774	775	776	777	778	779	780	781	782	783	784	785	786	787	788	789	790	791	792	793	794	795	796	797	798	799	800	801	802	803	804	805	806	807	808	809	810	811	812	813	814	815	816	817	818	819	820	821	822	823	824	825	826	827	828	829	830	831	832	833	834	835	836	837	838	839	840	841	842	843	844	845	846	847	848	849	850	851	852	853	854	855	856	857	858	859	860	861	862	863	864	865	866	867	868	869	870	871	872	873	874	875	876	877	878	879	880	881	882	883	884	885	886	887	888	889	890	891	892	893	894	895	896	897	898	899	900	901	902	903	904	905	906	907	908	909	910	911	912	913	914	915	916	917	918	919	920	921	922	923	924	925	926	927	928	929	930	931	932	933	934	935	936	937	938	939	940	941	942	943	944	945	946	947	948	949	950	951	952	953	954	955	956	957	958	959	960	961	962	963	964	965	966	967	968	969	970	971	972	973	974	975	976	977	978	979	980	981	982	983	984	985	986	987	988	989	990	991	992	993	994	995	996	997	998	999	1000	1001	1002	1003	1004	1005	1006	1007	1008	1009	1010	1011	1012	1013	1014	1015	1016	1017	1018	1019	1020	1021	1022	1023	1024	1025	1026	1027	1028	1029	1030	1031	1032	1033	1034	1035	1036	1037	1038	1039	1040	1041	1042	1043	1044	1045	1046	1047	1048	1049	1050	1051	1052	1053	1054	1055	1056	1057	1058	1059	1060	1061	1062	1063	1064	1065	1066	1067	1068	1069	1070	1071	1072	1073	1074	1075	1076	1077	1078	1079	1080	1081	1082	1083	1084	1085	1086	1087	1088	1089	1090	1091	1092	1093	1094	1095	1096	1097	1098	1099	1100	1101	1102	1103	1104	1105	1106	1107	1108	1109	1110	1111	1112	1113	1114	1115	1116	1117	1118	1119	1120	1121	1122	1123	1124	1125	1126	1127	1128	1129	1130	1131	1132	1133	1134	1135	1136	1137	1138	1139	1140	1141	1142	1143	1144	1145	1146	1147	1148	1149	1150	1151	1152	1153	1154	1155	1156	1157	1158	1159	1160	1161	1162	1163	1164	1165	1166	1167	1168	1169	1170	1171	1172	1173	1174	1175	1176	1177	1178	1179	1180	1181	1182	1183	1184	1185	1186	1187	1188	1189	1190	1191	1192	1193	1194	1195	1196	1197	1198	1199	1200	1201	1202	1203	1204	1205	1206	1207	1208	1209	1210	1211	1212	1213	1214	1215	1216	1217	1218	1219	1220	1221	1222	1223	1224	1225	1226	1227	1228	1229	1230	1231	1232	1233	1234	1235	1236	1237	1238	1239	1240	1241	1242	1243	1244	1245	1246	1247	1248	1249	1250	1251	1252	1253	1254	1255	1256	1257	1258	1259	1260	1261	1262	1263	1264	1265	1266	1267	1268	1269	1270	1271	1272	1273	1274	1275	1276	1277	1278	1279	1280	1281	1282	1283	1284	1285	1286	1287	1288	1289	1290	1291	1292	1293	1294	1295	1296	1297	1298	1299	1300	1301	1302	1303	1304	1305	1306	1307	1308	1309	1310	1311	1312	1313	1314	1315	1316	1317	1318	1319	1320	1321	1322	1323	1324	1325	1326	1327	1328	1329	1330	1331	1332	1333	1334	1335	1336	1337	1338	1339	1340	1341	1342	1343	1344	1345	1346	1347	1348	1349	1350	1351	1352	1353	1354	1355	1356	1357	1358	1359	1360	1361	1362	1363	1364	1365	1366	1367	1368	1369	1370	1371	1372	1373	1374	1375	1376	1377	1378	1379	1380	1381	1382	1383	1384	1385	1386	1387	1388	1389	1390	1391	1392	1393	1394	1395	1396	1397	1398	1399	1400	1401	1402	1403	1404	1405	1406	1407	1408	1409	1410	1411	1412	1413	1414	1415	1416	1417	1418	1419	1420	1421	1422	1423	1424	1425	1426	1427	1428	1429	1430	1431	1432	1433	1434	1435	1436	1437	1438	1439	1440	1441	1442	1443	1444	1445	1446	1447	1448	1449	1450	1451	1452	1453	1454	1455	1456	1457	1458	1459	1460	1461	1462	1463	1464	1465	1466	1467	1468	1469	1470	1471	1472	1473	1474	1475	1476	1477	1478	1479	1480	1481	1482	1483	1484	1485	1486	1487	1488	1489	1490	1491	1492</
---	---	---	---	---	---	---	---	---	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	--------

## Immutability

An object from a data type is immutable if its data-type value cannot change once created

## Immutability

An object from a data type is immutable if its data-type value cannot change once created

The purpose of many data types (eg, `Stopwatch`) is to encapsulate values that do not change, while for many other data types (eg, `Turtle`), the very purpose of the abstraction is to encapsulate values as they change

## Immutability

An object from a data type is immutable if its data-type value cannot change once created

The purpose of many data types (eg, `Stopwatch`) is to encapsulate values that do not change, while for many other data types (eg, `Turtle`), the very purpose of the abstraction is to encapsulate values as they change

Generally, immutable data types are easier to use and harder to misuse because the scope of code that can change object values is far smaller than for mutable types



## Immutability

An object from a data type is immutable if its data-type value cannot change once created

The purpose of many data types (eg, `Stopwatch`) is to encapsulate values that do not change, while for many other data types (eg, `Turtle`), the very purpose of the abstraction is to encapsulate values as they change

Generally, immutable data types are easier to use and harder to misuse because the scope of code that can change object values is far smaller than for mutable types

In Python, lists are mutable, whereas strings and tuples are immutable

## Polymorphism



## Polymorphism

A method (or function) that can take arguments with different types is said to be polymorphic

## Polymorphism

A method (or function) that can take arguments with different types is said to be polymorphic

Duck typing is a programming style in which the language does not formally specify the requirements for a function's arguments

## Polymorphism

A method (or function) that can take arguments with different types is said to be polymorphic

Duck typing is a programming style in which the language does not formally specify the requirements for a function's arguments

Python uses duck typing for all operations (function calls, method calls, and operators), and raises a `TypeError` at run time if an operation cannot be applied to an object because it is of an inappropriate type

## Polymorphism

A method (or function) that can take arguments with different types is said to be polymorphic

Duck typing is a programming style in which the language does not formally specify the requirements for a function's arguments

Python uses duck typing for all operations (function calls, method calls, and operators), and raises a `TypeError` at run time if an operation cannot be applied to an object because it is of an inappropriate type

Duck typing leads to simpler and more flexible client code and puts the focus on operations rather than the type

## Polymorphism

A method (or function) that can take arguments with different types is said to be polymorphic

Duck typing is a programming style in which the language does not formally specify the requirements for a function's arguments

Python uses duck typing for all operations (function calls, method calls, and operators), and raises a `TypeError` at run time if an operation cannot be applied to an object because it is of an inappropriate type

Duck typing leads to simpler and more flexible client code and puts the focus on operations rather than the type

A disadvantage of duck typing is that it is difficult to know precisely what the contract is between the client and the implementation — the API simply does not carry this information

## Overloading





## Overloading

The ability to define a data type that provides its own definitions of operators is a form of polymorphism known as operator overloading

## Overloading

The ability to define a data type that provides its own definitions of operators is a form of polymorphism known as operator overloading

In Python, we can overload almost every operator, including operators for arithmetic, comparisons, indexing, and slicing

## Overloading

The ability to define a data type that provides its own definitions of operators is a form of polymorphism known as operator overloading

In Python, we can overload almost every operator, including operators for arithmetic, comparisons, indexing, and slicing

We can also overload built-in functions, including absolute value, length, hashing, and type conversion

## Overloading

The ability to define a data type that provides its own definitions of operators is a form of polymorphism known as operator overloading

In Python, we can overload almost every operator, including operators for arithmetic, comparisons, indexing, and slicing

We can also overload built-in functions, including absolute value, length, hashing, and type conversion

Overloading operators and built-in functions makes user-defined types behave more like built-in types

## Overloading

The ability to define a data type that provides its own definitions of operators is a form of polymorphism known as operator overloading

In Python, we can overload almost every operator, including operators for arithmetic, comparisons, indexing, and slicing

We can also overload built-in functions, including absolute value, length, hashing, and type conversion

Overloading operators and built-in functions makes user-defined types behave more like built-in types

To perform an operation, Python internally converts the expression into a call on the corresponding special method

## Overloading

The ability to define a data type that provides its own definitions of operators is a form of polymorphism known as operator overloading

In Python, we can overload almost every operator, including operators for arithmetic, comparisons, indexing, and slicing

We can also overload built-in functions, including absolute value, length, hashing, and type conversion

Overloading operators and built-in functions makes user-defined types behave more like built-in types

To perform an operation, Python internally converts the expression into a call on the corresponding special method

To call a built-in function, Python internally calls the corresponding special method instead

## Overloading

The ability to define a data type that provides its own definitions of operators is a form of polymorphism known as operator overloading

In Python, we can overload almost every operator, including operators for arithmetic, comparisons, indexing, and slicing

We can also overload built-in functions, including absolute value, length, hashing, and type conversion

Overloading operators and built-in functions makes user-defined types behave more like built-in types

To perform an operation, Python internally converts the expression into a call on the corresponding special method

To call a built-in function, Python internally calls the corresponding special method instead

To overload an operator or built-in function, we include an implementation of the corresponding special method with our own code

## Overloading





## Overloading

### Special methods for arithmetic operators

Client Operation	Special Method	Description
$x + y$	<code>--add__(self, y)</code>	sum of $x$ and $y$
$x - y$	<code>--sub__(self, y)</code>	difference of $x$ and $y$
$x * y$	<code>--mul__(self, y)</code>	product of $x$ and $y$
$x ** y$	<code>--pow__(self, y)</code>	$x$ to the power $y$
$x / y$	<code>--div__(self, y)</code>	quotient of $x$ and $y$
$x // y$	<code>--floordiv__(self, y)</code>	floored quotient of $x$ and $y$
$x \% y$	<code>--mod__(self, y)</code>	remainder when dividing $x$ by $y$
$+x$	<code>--pos__(self)</code>	$x$
$-x$	<code>--neg__(self)</code>	arithmetic negation of $x$

## Overloading



## Overloading

### Special methods for comparison operators

Client Operation	Special Method	Description
<code>x == y</code>	<code>--eq__(self, y)</code>	are <code>x</code> and <code>y</code> equal?
<code>x != y</code>	<code>--ne__(self, y)</code>	are <code>x</code> and <code>y</code> not equal?
<code>x &lt; y</code>	<code>--lt__(self, y)</code>	is <code>x</code> less than <code>y</code> ?
<code>x &lt;= y</code>	<code>--le__(self, y)</code>	is <code>x</code> less than or equal to <code>y</code> ?
<code>x &gt; y</code>	<code>--gt__(self, y)</code>	is <code>x</code> greater than <code>y</code> ?
<code>x &gt;= y</code>	<code>--ge__(self, y)</code>	is <code>x</code> greater than or equal to <code>y</code> ?

## Overloading



## Overloading

### Special methods for built-in functions

Client Operation	Special Method	Description
<code>len(x)</code>	<code>__len__(self)</code>	length of $x$
<code>float(x)</code>	<code>__float__(self)</code>	float equivalent of $x$
<code>int(x)</code>	<code>__int__(self)</code>	integer equivalent of $x$
<code>str(x)</code>	<code>__str__(self)</code>	string representation of $x$
<code>abs(x)</code>	<code>__abs__(self)</code>	absolute value of $x$
<code>hash(x)</code>	<code>__hash__(self)</code>	integer hash code for $x$
<code>iter(x)</code>	<code>__iter__(self)</code>	iterator for $x$

## Functions are Objects

Functions are objects

Functions are objects

Functions are objects

Functions are objects

Functions are objects

Functions are objects

Functions are objects

Functions are objects

Functions are objects

Functions are objects

Functions are objects

Functions are objects

Functions are objects

Functions are objects

Functions are objects

Functions are objects

Functions are objects

Functions are objects

Functions are objects

Functions are objects

Functions are objects

Functions are objects

Functions are objects

Functions are objects

Functions are objects

Functions are objects

Functions are objects

Functions are objects

Functions are objects

Functions are objects

Functions are objects

## Functions are Objects

In Python, everything is an object, including functions, which means we can use them as arguments to functions and return them as results

## Functions are Objects

In Python, everything is an object, including functions, which means we can use them as arguments to functions and return them as results

Defining higher-order functions that manipulate other functions is common both in mathematics and scientific computing



## Functions are Objects

In Python, everything is an object, including functions, which means we can use them as arguments to functions and return them as results

Defining higher-order functions that manipulate other functions is common both in mathematics and scientific computing

For example, the following function evaluates the Riemann integral (ie, the area under the curve) of a real-valued function  $f()$  in the interval  $(a, b)$ , using the rectangle rule with  $n$  rectangles

```
def integrate(f, a, b, n = 1000):  
    total = 0.0  
    dt = 1.0 * (b - a) / n  
    for i in range(n):  
        total += dt * f(a + (i + 0.5) * dt)  
    return total
```

## Functions are Objects

In Python, everything is an object, including functions, which means we can use them as arguments to functions and return them as results

Defining higher-order functions that manipulate other functions is common both in mathematics and scientific computing

For example, the following function evaluates the Riemann integral (ie, the area under the curve) of a real-valued function  $f()$  in the interval  $(a, b)$ , using the rectangle rule with  $n$  rectangles

```
def integrate(f, a, b, n = 1000):  
    total = 0.0  
    dt = 1.0 * (b - a) / n  
    for i in range(n):  
        total += dt * f(a + (i + 0.5) * dt)  
    return total
```

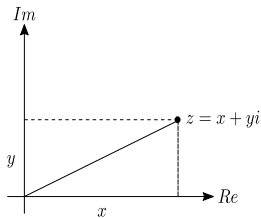
The following statement uses the above function to compute the area under the curve  $f(x) = x^2$  in the interval  $(0, 1)$

```
area = integrate(lambda x : x * x, 0, 1)
```

## Examples

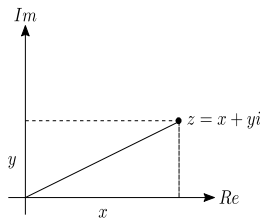
## Examples

A complex number  $z$  in the cartesian form is expressed as  $z = x + yi$ , where  $x$  (the real part) and  $y$  (the imaginary part) are real numbers and  $i = \sqrt{-1}$



## Examples

A complex number  $z$  in the cartesian form is expressed as  $z = x + yi$ , where  $x$  (the real part) and  $y$  (the imaginary part) are real numbers and  $i = \sqrt{-1}$



### Complex arithmetic

- Conjugate:  $(x + yi)^* = x - yi$
- Addition:  $(x + yi) + (v + wi) = (x + v) + (y + w)i$
- Multiplication:  $(x + yi) \times (v + wi) = (xv - yw) + (yv + xw)i$
- Magnitude:  $|x + yi| = \sqrt{x^2 + y^2}$

## Examples

## Examples

A data type `Complex` for representing complex numbers

### Complex

<code>Complex(x, y)</code>	a new complex object <i>c</i> with value $x + yi$
----------------------------	---

<code>c.re()</code>	real part of <i>c</i>
---------------------	-----------------------

<code>c.im()</code>	imaginary part of <i>c</i>
---------------------	----------------------------

<code>c.conjugate()</code>	conjugate of <i>c</i>
----------------------------	-----------------------

<code>c + d</code>	sum of <i>c</i> and <i>d</i>
--------------------	------------------------------

<code>c * d</code>	product of <i>c</i> and <i>d</i>
--------------------	----------------------------------

<code>c == d</code>	are <i>c</i> and <i>d</i> equal?
---------------------	----------------------------------

<code>abs(c)</code>	magnitude of <i>c</i>
---------------------	-----------------------

<code>str(c)</code>	string representation of <i>c</i>
---------------------	-----------------------------------

## Examples



## Examples

complex.py

```
1 import math
2 import stdio
3
4 class Complex:
5     def __init__(self, re=0.0, im=0.0):
6         self._re = re
7         self._im = im
8
9     def re(self):
10        return self._re
11
12    def im(self):
13        return self._im
14
15    def conjugate(self):
16        return Complex(self._re, -self._im)
17
18    def __add__(self, other):
19        re = self._re + other._re
20        im = self._im + other._im
21        return Complex(re, im)
22
23    def __mul__(self, other):
24        re = self._re * other._re - self._im * other._im
25        im = self._re * other._im + self._im * other._re
26        return Complex(re, im)
27
28    def __abs__(self):
29        return math.sqrt(self._re * self._re + self._im * self._im)
30
31    def __eq__(self, other):
32        return self._re == other._re and self._im == other._im
33
34    def __str__(self):
35        SUFFIX = 'i'
```

## Examples

complex.py

```
36     if self._im == 0:
37         return str(self._re)
38     elif self._re == 0:
39         return str(self._im) + SUFFIX
40     elif self._im < 0:
41         return str(self._re) + ' - ' + str(-self._im) + SUFFIX
42     else:
43         return str(self._re) + ' + ' + str(self._im) + SUFFIX
44
45 def _main():
46     a = Complex(5.0, -6.0)
47     b = Complex(3.0, 4.0)
48     stdio.writeln("a      = " + str(a))
49     stdio.writeln("b      = " + str(b))
50     stdio.writeln("conj(a) = " + str((a.conjugate())))
51     stdio.writeln("a + b   = " + str(a + b))
52     stdio.writeln("a * b   = " + str(a * b))
53     stdio.writeln("|b|    = " + str(abs(b)))
54
55 if __name__ == '__main__':
56     _main()
```

## Examples

## Examples

Program: `mandelbrot.py`

## Examples

Program: `mandelbrot.py`

- Command-line input: `xc` (float), `yc` (float), and `size` (float)

## Examples

Program: `mandelbrot.py`

- Command-line input:  $x_c$  (float),  $y_c$  (float), and *size* (float)
- Standard draw output: *size*-by-*size* region of the Mandelbrot set, centered at  $(x_c, y_c)$

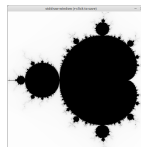
## Examples

Program: `mandelbrot.py`

- Command-line input: `xc` (float), `yc` (float), and `size` (float)
- Standard draw output: *size-by-size* region of the Mandelbrot set, centered at  $(xc, yc)$

```
>_ ~/workspace/ipp/programs
```

```
$ python3 mandelbrot.py -0.5 0 2
```



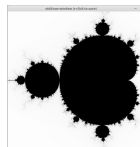
## Examples

Program: `mandelbrot.py`

- Command-line input: `xc` (float), `yc` (float), and `size` (float)
- Standard draw output: *size-by-size* region of the Mandelbrot set, centered at  $(xc, yc)$

```
>_ ~/workspace/ipp/programs
```

```
$ python3 mandelbrot.py -0.5 0 2
```



```
>_ ~/workspace/ipp/programs
```

```
$ python3 mandelbrot.py 0.1015 -0.633 .01
```





## Examples

## Examples

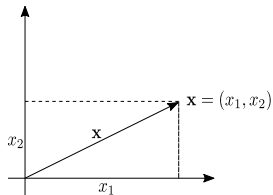
mandelbrot.py

```
1 from color import Color
2 from complex import Complex
3 from picture import Picture
4 import stddraw
5 import sys
6
7 def main():
8     xc = float(sys.argv[1])
9     yc = float(sys.argv[2])
10    size = float(sys.argv[3])
11    N = 512
12    ITERATIONS = 255
13    picture = Picture(N, N)
14    for col in range(N):
15        for row in range(N):
16            x0 = xc - size / 2 + size * col / N
17            y0 = yc - size / 2 + size * row / N
18            z0 = Complex(x0, y0)
19            gray = ITERATIONS - _mandel(z0, ITERATIONS)
20            color = Color(gray, gray, gray)
21            picture.set(col, N - 1 - row, color)
22    stddraw.setCanvasSize(N, N)
23    stddraw.picture(picture)
24    stddraw.show()
25
26 def _mandel(z0, iterations):
27     z = z0
28     for i in range(iterations):
29         if abs(z) > 2.0:
30             return i
31         z = z * z + z0
32     return iterations
33
34 if __name__ == '__main__':
35     main()
```

## Examples

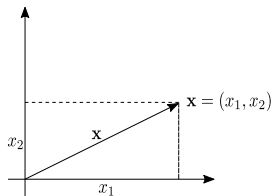
## Examples

A spatial vector is an abstract entity that has a magnitude and a direction



## Examples

A spatial vector is an abstract entity that has a magnitude and a direction



Vector operations, assuming  $\mathbf{x} = (x_1, x_2, \dots, x_n)$ ,  $\mathbf{y} = (y_1, y_2, \dots, y_n)$ , and  $\alpha \in \mathbb{R}$

- Addition:  $\mathbf{x} + \mathbf{y} = (x_1 + y_1, x_2 + y_2, \dots, x_n + y_n)$
- Subtraction:  $\mathbf{x} - \mathbf{y} = (x_1 - y_1, x_2 - y_2, \dots, x_n - y_n)$
- Scalar product:  $\alpha \mathbf{x} = (\alpha x_1, \alpha x_2, \dots, \alpha x_n)$
- Dot product:  $\mathbf{x} \cdot \mathbf{y} = x_1 y_1 + x_2 y_2 + \dots + x_n y_n$
- Magnitude:  $|\mathbf{x}| = (x_1^2 + x_2^2 + \dots + x_n^2)^{1/2}$
- Direction:  $\mathbf{x}/|\mathbf{x}| = (x_1/|\mathbf{x}|, x_2/|\mathbf{x}|, \dots, x_n/|\mathbf{x}|)$

## Examples

## Examples

A data type `Vector` for spatial vectors

### Vector

<code>Vector(a)</code>	a new vector $v$ with Cartesian coordinates taken from the list $a$
<code>v[i]</code>	$i$ th Cartesian coordinates of $v$
<code>v + w</code>	sum of $v$ and $w$
<code>v - w</code>	difference of $v$ and $w$
<code>v.dot(w)</code>	dot product of $v$ and $w$
<code>v.scale(alpha)</code>	scalar product of float $\alpha$ and $v$
<code>v.direction()</code>	unit vector in the same direction as $v$
<code>abs(v)</code>	magnitude of $v$
<code>len(v)</code>	length of $v$
<code>str(v)</code>	string representation of $v$

## Examples



## Examples

 vector.py

```
1 import math
2 import ndarray
3 import stdio
4
5 class Vector:
6     def __init__(self, a):
7         self._n = len(a)
8         self._coords = a[:]
9
10    def __getitem__(self, i):
11        return self._coords[i]
12
13    def __add__(self, other):
14        result = ndarray.create1D(self._n, 0)
15        for i in range(self._n):
16            result[i] = self._coords[i] + other._coords[i]
17        return Vector(result)
18
19    def __sub__(self, other):
20        result = ndarray.create1D(self._n, 0)
21        for i in range(self._n):
22            result[i] = self._coords[i] - other._coords[i]
23        return Vector(result)
24
25    def dot(self, other):
26        result = 0
27        for i in range(self._n):
28            result += self._coords[i] * other._coords[i]
29        return result
30
31    def scale(self, alpha):
32        result = ndarray.create1D(self._n, 0)
33        for i in range(self._n):
34            result[i] = alpha * self._coords[i]
35        return Vector(result)
```

## Examples

vector.py

```
36
37     def direction(self):
38         return self.scale(1.0 / abs(self))
39
40     def __abs__(self):
41         return math.sqrt(self.dot(self))
42
43     def dimension(self):
44         return self._n
45
46     def __str__(self):
47         return str(self._coords)
48
49 def _main():
50     xCoords = [1.0, 2.0, 3.0, 4.0]
51     yCoords = [5.0, 2.0, 4.0, 1.0]
52     x = Vector(xCoords)
53     y = Vector(yCoords)
54     stdio.writeln('x      = ' + str(x))
55     stdio.writeln('y      = ' + str(y))
56     stdio.writeln('x + y   = ' + str(x + y))
57     stdio.writeln('x - y   = ' + str(x - y))
58     stdio.writeln('x dot y = ' + str(x.dot(y)))
59     stdio.writeln('10x    = ' + str(x.scale(10.0)))
60     stdio.writeln('xhat   = ' + str(x.direction()))
61     stdio.writeln('|x|    = ' + str(abs(x)))
62     stdio.writeln('ydim   = ' + str(y.dimension()))
63
64 if __name__ == '__main__':
65     _main()
```

## Examples

## Examples

A data type `Sketch` for compactly representing the content of a document

### Sketch

<code>Sketch(text, k, d)</code>	a new sketch $s$ built from the string $text$ using $k$ -grams and dimension $d$
<code>s.similarTo(t)</code>	similarity measure between sketches $s$ and $t$ (a float between 0.0 and 1.0)
<code>str(s)</code>	string representation of $s$

## Examples

## Examples

sketch.py

```
1 from vector import Vector
2 import stdarray
3 import stdio
4 import sys
5
6 class Sketch:
7     def __init__(self, text, k, d):
8         freq = stdarray.createID(d, 0)
9         for i in range(len(text) - k + 1):
10             kgram = text[i:i + k]
11             h = hash(kgram)
12             freq[abs(h % d)] += 1
13         vector = Vector(freq)
14         self._sketch = vector.direction()
15
16     def similarTo(self, other):
17         return self._sketch.dot(other._sketch)
18
19     def __str__(self):
20         return str(self._sketch)
21
22 def _main():
23     k = int(sys.argv[1])
24     d = int(sys.argv[2])
25     text = stdio.readAll()
26     sketch = Sketch(text, k, d)
27     stdio.writeln(sketch)
28
29 if __name__ == '__main__':
30     _main()
```

## Examples

## Examples

Program: `comparedocuments.py`



## Examples

Program: `comparedocuments.py`

- Command-line input:  $k$  (int),  $d$  (int), and *path* (str)

## Examples

Program: `comparedocuments.py`

- Command-line input:  $k$  (int),  $d$  (int), and *path* (str)
- Standard input: a document list

## Examples

Program: `comparedocuments.py`

- Command-line input:  $k$  (int),  $d$  (int), and *path* (str)
- Standard input: a document list
- Standard output: computes  $d$ -dimensional profiles based on  $k$ -gram frequencies for all those documents under the *path* directory, and writes a matrix of similarity measures between all pairs of documents

## Examples

Program: `comparedocuments.py`

- Command-line input:  $k$  (int),  $d$  (int), and *path* (str)
- Standard input: a document list
- Standard output: computes  $d$ -dimensional profiles based on  $k$ -gram frequencies for all those documents under the *path* directory, and writes a matrix of similarity measures between all pairs of documents

```
>_ ~/workspace/ipp/programs
```

```
$ cat ../data/documents.txt
```

```
constitution.txt
```

```
tomsawyer.txt
```

```
huckfinn.txt
```

```
tale.txt
```

```
prejudice.txt
```

```
actg.txt
```

```
djia.csv
```

```
$ python3 comparedocuments.py 5 10000 ../data < ../data/documents.txt
```

	cons	toms	huck	tale	prej	actg	djia
cons	1.00	0.66	0.60	0.67	0.64	0.11	0.18
toms	0.66	1.00	0.93	0.92	0.88	0.15	0.23
huck	0.60	0.93	1.00	0.84	0.81	0.13	0.21
tale	0.67	0.92	0.84	1.00	0.87	0.14	0.21
prej	0.64	0.88	0.81	0.87	1.00	0.15	0.24
actg	0.11	0.15	0.13	0.14	0.15	1.00	0.12
djia	0.18	0.23	0.21	0.21	0.24	0.12	1.00

## Examples

## Examples

comparedocuments.py

```
1 from instream import InStream
2 from sketch import Sketch
3 import stdarray
4 import stdio
5 import sys
6
7 def main():
8     k = int(sys.argv[1])
9     d = int(sys.argv[2])
10    path = sys.argv[3]
11    filenames = stdio.readAllStrings()
12    n = len(filenames)
13    sketches = stdarray.create1D(n, None)
14    for i in range(n):
15        inStream = InStream(path + '/' + filenames[i])
16        text = inStream.readAll()
17        sketches[i] = Sketch(text, k, d)
18    stdio.write(' ')
19    for filename in filenames:
20        stdio.writefile('%8.4s', filename)
21    stdio.writeln()
22    for i in range(n):
23        stdio.writefile('%8.4s', filenames[i])
24        for j in range(n):
25            stdio.writefile('%8.2f', sketches[i].similarTo(sketches[j]))
26        stdio.writeln()
27
28 if __name__ == '__main__':
29     main()
```

## Examples

### A data type `Counter` for counting

Counter	
<code>Counter(id, maxCount)</code>	a new counter <i>c</i> named <i>id</i> , with maximum value <i>maxCount</i>
<code>c.increment()</code>	increment <i>c</i> , unless its value is <i>maxCount</i>
<code>c.tally()</code>	value of <i>c</i>
<code>c.reset()</code>	reset value of <i>c</i>
<code>c &lt; d</code>	is <i>c</i> less than <i>d</i> ?
<code>c == d</code>	are <i>c</i> and <i>d</i> equal?
<code>str(c)</code>	string representation of <i>c</i>



## Examples

## Examples

counter.py

```
1 import stdarray
2 import stdio
3 import stdrandom
4 import sys
5
6 class Counter:
7     def __init__(self, id):
8         self._id = id
9         self._count = 0
10
11     def increment(self):
12         self._count += 1
13
14     def tally(self):
15         return self._count
16
17     def reset(self):
18         self._count = 0
19
20     def __lt__(self, other):
21         return self._count < other._count
22
23     def __eq__(self, other):
24         return self._count == other._count
25
26     def __str__(self):
27         return str(self._count) + ' ' + self._id
28
29 def _main():
30     n = int(sys.argv[1])
31     trials = int(sys.argv[2])
32     counters = stdarray.create1D(n, None)
33     for i in range(n):
34         counters[i] = Counter('counter ' + str(i))
35     for i in range(trials):
```

## Examples

 counter.py

```
36         counters[stdrandom.uniformInt(0, n)].increment()
37     for counter in sorted(counters):
38         stdio.writeln(counter)
39
40 if __name__ == '__main__':
41     _main()
```

## Examples

## Examples

```
>_ ~/workspace/ipp/programs
```

```
$ python3 counter.py 6 10000  
1620 counter 0  
1629 counter 3  
1653 counter 2  
1686 counter 1  
1686 counter 4  
1726 counter 5
```

## Examples

## Examples

A comparable data type `Country` that represents a country by its name, capital, and population

### Country

`Country(name, capital, population)`

constructs a country  $c$  given its name, capital, and population

`c < d`

is the country  $c$  less than country  $d$  by name?

`c == d`

is the country  $c$  equal to country  $d$  by population?

`str(c)`

string representation of  $c$

## Examples



## Examples

country.py

```
1 import stdarray
2 import stdio
3
4 class Country:
5     def __init__(self, name, capital, population):
6         self._name = name
7         self._capital = capital
8         self._population = population
9
10    def __lt__(self, other):
11        return self._name < other._name
12
13    def __eq__(self, other):
14        return self._name == other._name
15
16    def __str__(self):
17        return self._name + ' (' + self._capital + '): ' + str(self._population)
18
19 def _main():
20     countries = stdarray.create1D(5, None)
21     countries[0] = Country('United States', 'Washington, D.C.', 329334246)
22     countries[1] = Country('Pakistan', 'Islamabad', 218719520)
23     countries[2] = Country('India', 'New Delhi', 1358989650)
24     countries[3] = Country('China', 'Beijing', 1401463880)
25     countries[4] = Country('Indonesia', 'Jakarta', 266911900)
26     stdio.writeln('Unsorted:')
27     for country in countries:
28         stdio.writeln(country)
29     stdio.writeln()
30     stdio.writeln('Sorted by name:')
31     for country in sorted(countries):
32         stdio.writeln(country)
33     stdio.writeln()
34     stdio.writeln('Sorted by capital:')
35     for country in sorted(countries, key=lambda country: country._capital):
```

## Examples

## Examples

 country.py

```
36         stdio.writeln(country)
37     stdio.writeln()
38     stdio.writeln('Sorted by population:')
39     for country in sorted(countries, key=lambda country: country._population):
40         stdio.writeln(country)
41     stdio.writeln()
42     stdio.writeln('Reverse sorted by population:')
43     for country in sorted(countries, key=lambda country: country._population, reverse=True):
44         stdio.writeln(country)
45
46 if __name__ == '__main__':
47     _main()
```

## Examples

## Examples

```
>_ ~/workspace/ipp/programs
```

```
$ python3 country.py
```

```
Unsorted:
```

```
United States (Washington, D.C.): 329334246
```

```
Pakistan (Islamabad): 218719520
```

```
India (New Delhi): 1358989650
```

```
China (Beijing): 1401463880
```

```
Indonesia (Jakarta): 266911900
```

```
Sorted by name:
```

```
China (Beijing): 1401463880
```

```
India (New Delhi): 1358989650
```

```
Indonesia (Jakarta): 266911900
```

```
Pakistan (Islamabad): 218719520
```

```
United States (Washington, D.C.): 329334246
```

```
Sorted by capital:
```

```
China (Beijing): 1401463880
```

```
Pakistan (Islamabad): 218719520
```

```
Indonesia (Jakarta): 266911900
```

```
India (New Delhi): 1358989650
```

```
United States (Washington, D.C.): 329334246
```

```
Sorted by population:
```

```
Pakistan (Islamabad): 218719520
```

```
Indonesia (Jakarta): 266911900
```

```
United States (Washington, D.C.): 329334246
```

```
India (New Delhi): 1358989650
```

```
China (Beijing): 1401463880
```

```
Reverse sorted by population:
```

```
China (Beijing): 1401463880
```

```
India (New Delhi): 1358989650
```

```
United States (Washington, D.C.): 329334246
```

```
Indonesia (Jakarta): 266911900
```

```
Pakistan (Islamabad): 218719520
```

## Examples

An iterable `FibonacciSequence` data type for iterating over Fibonacci sequences

### `FibonacciSequence`

<code>FibonacciSequence(n)</code>	a new object $f$ for iterating over the first $n$ Fibonacci numbers
<code>iter(f)</code>	an iterable object <i>fiter</i> on $f$
<code>next(fiter)</code>	the next number in the Fibonacci sequence <i>fiter</i>

## Examples



## Examples

📄 fibonaccisequence.py

```
import stdio
import sys

class FibonacciSequence:
    def __init__(self, n):
        self._n = n
        self._a = 1
        self._b = 1
        self._count = 0

    def __iter__(self):
        return self

    def __next__(self):
        self._count += 1
        if self._count > self._n:
            raise StopIteration()
        if self._count <= 2:
            return 1
        temp = self._a
        self._a = self._b
        self._b += temp
        return self._b

def _main():
    n = int(sys.argv[1])
    for v in FibonacciSequence(n):
        stdio.writeln(v)

if __name__ == '__main__':
    _main()
```

## Examples

## Examples

```
>_ ~/workspace/ipp/programs
```

```
$ python3 fibonaccisequence.py 10
```

```
1  
1  
2  
3  
5  
8  
13  
21  
34  
55
```

## Exceptions

## Exceptions

An exception is a disruptive event that occurs while a program is running, often to signal an error

## Exceptions

An exception is a disruptive event that occurs while a program is running, often to signal an error

The action taken in response is known as raising an exception (or error)

## Exceptions

An exception is a disruptive event that occurs while a program is running, often to signal an error

The action taken in response is known as raising an exception (or error)

We can raise our own exceptions as follows

```
raise Exception('Error message here.')
```

## Exceptions

An exception is a disruptive event that occurs while a program is running, often to signal an error

The action taken in response is known as raising an exception (or error)

We can raise our own exceptions as follows

```
raise Exception('Error message here.')
```

We can handle exceptions using a try-except block



## Exceptions

## Exceptions

Program: `errorhandling.py`

## Exceptions

Program: `errorhandling.py`

- Command-line input:  $x$  (float)

## Exceptions

Program: `errorhandling.py`

- Command-line input:  $x$  (float)
- Standard output: square root of  $x$ , reporting an error if  $x$  is not specified, is not a float, or is negative

## Exceptions


Program: `errorhandling.py`

- Command-line input:  $x$  (float)
- Standard output: square root of  $x$ , reporting an error if  $x$  is not specified, is not a float, or is negative

```
>_ ~/workspace/ipp/programs  
  
$ python3 errorhandling.py  
x not specified  
$ python3 errorhandling.py two  
x must be a float  
$ python3 errorhandling.py -2  
x must be positive  
$ python3 errorhandling.py 2  
1.4142135623730951
```

## Exceptions

# Exceptions

 errorhandling.py

```
1 import math
2 import stdio
3 import sys
4
5 def main():
6     try:
7         x = float(sys.argv[1])
8         result = _sqrt(x)
9         stdio.writeln(result)
10    except IndexError as e:
11        stdio.writeln('x not specified')
12    except ValueError as e:
13        stdio.writeln('x must be a float')
14    except Exception as e:
15        stdio.writeln(e)
16    finally:
17        stdio.writeln('Done!')
18
19 def _sqrt(x):
20     if x < 0:
21         raise Exception('x must be positive')
22     return math.sqrt(x)
23
24 if __name__ == '__main__':
25     main()
```