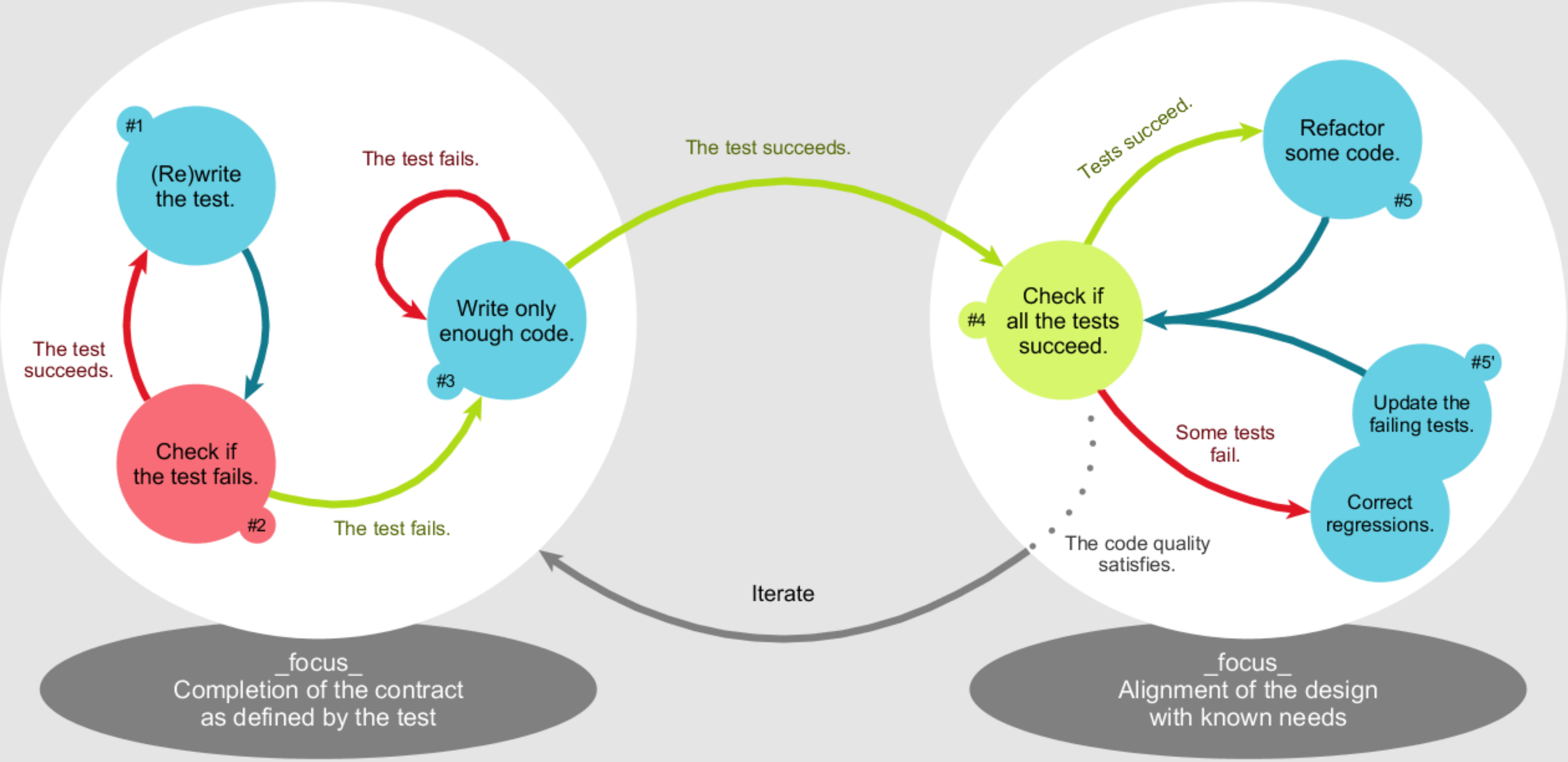
Unit Testing with Python

Test–Driven Development

- Add a test
- Run all tests and see if the new test fails
- Write the code
- Run tests
- Refactor Code
- Repeat

TEST-FIRST DEVELOPMENT

REFACTORING



#1 (Re)write the test.

#2 Check if the test fails.

The test succeeds.

The test fails.

#3 Write only enough code.

The test fails.

The test succeeds.

#4 Check if all the tests succeed.

Tests succeed.

#5 Refactor some code.

Some tests fail.

Correct regressions.

#5' Update the failing tests.

The code quality satisfies.

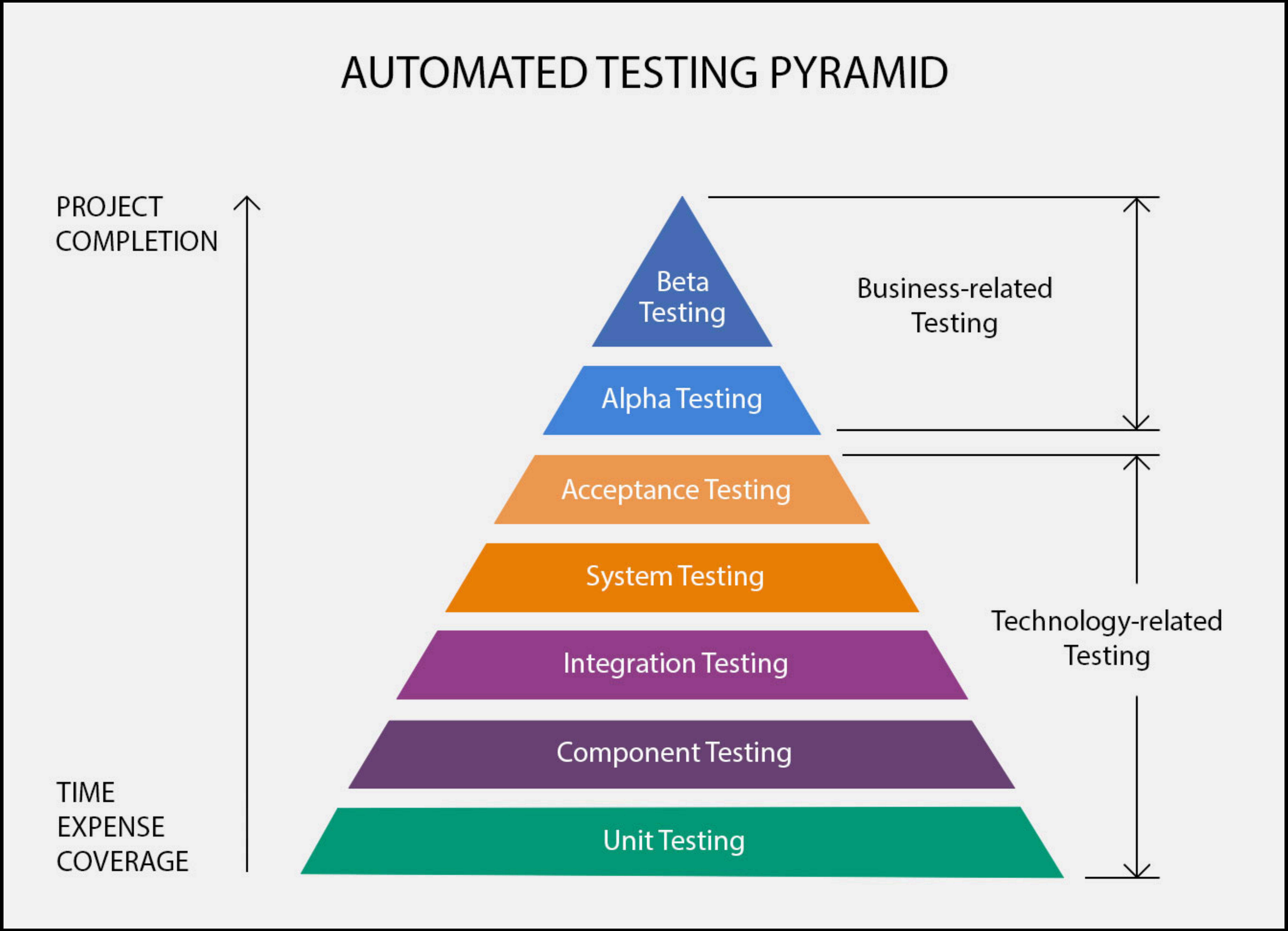
Iterate

focus
Completion of the contract
as defined by the test

focus
Alignment of the design
with known needs

Benefits of Test–Driven Development

- When writing, tests keep from over–coding.
- When refactoring and maintaining, tests ensure new changes don’t break old functionality.
- When working with a team, a comprehensive test suite ensures that one person’s changes don’t break someone else’s.



Credit: https://sphereinc.com/achieve-quality-code-and-roi-through-test-automation/

Requirements for a Unit Test

- Run automatically, without human input
- Determine automatically whether the test has been passed or failed, without human interpretation
- Run in isolation, separate from other test cases, even if multiple cases test the same code

unittest: Python’s Unit Testing Framework

Errors and Exceptions

- An ‘exception’ is an error that occurs when the code is run.
- Exceptions are not always fatal. They can be ‘handled’ by the program without exiting, or they can be ‘raised’ voluntarily.
- While Python has a number of built-in exceptions that it will raise when it encounters a certain error, custom exceptions can also be defined.

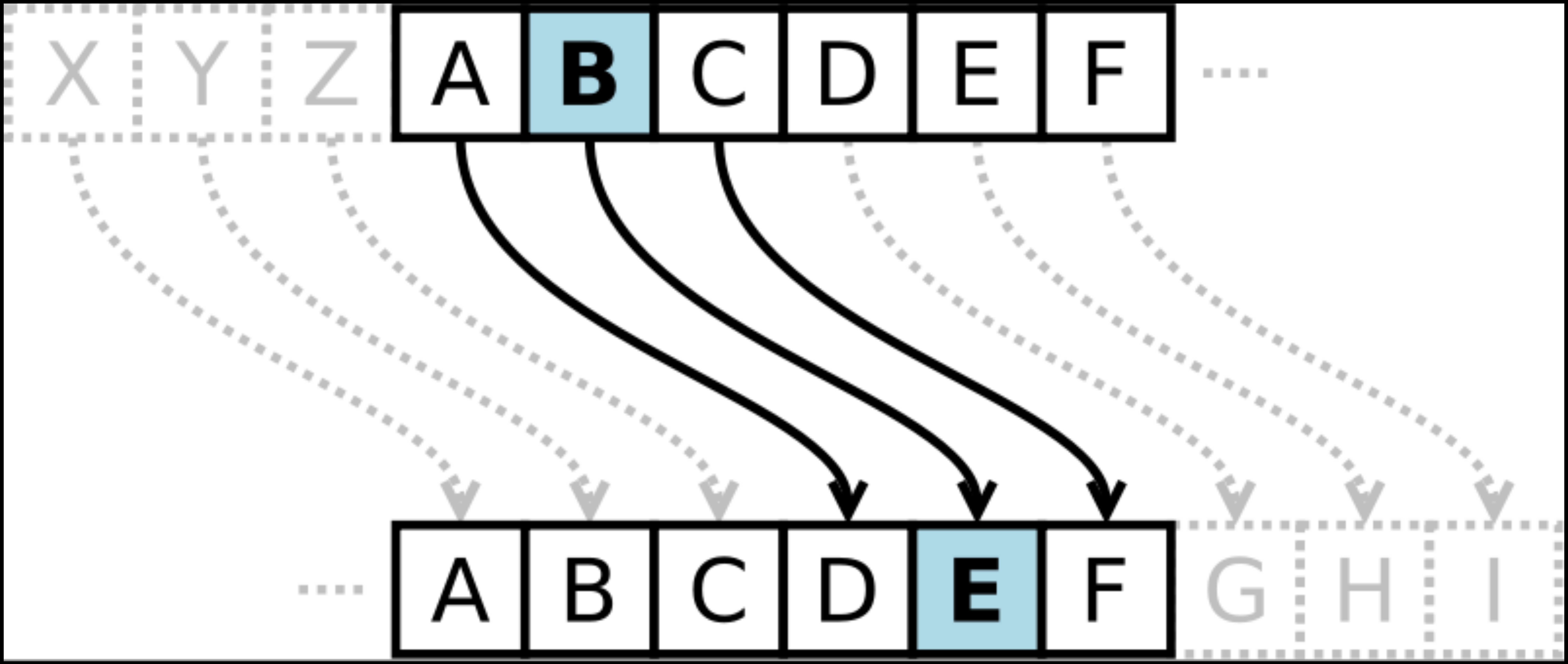
Common Built—In Python Exceptions

- IndexError
- KeyError
- NameError
- SyntaxError
- TypeError
- ValueError

Common unittest Assert Methods

- assertEqual(a, b)
- assertNotEqual(a,b)
- assertTrue(x)
- assertFalse(x)
- assertIs(a, b)
- assertIsNot(a, b)
- assertIsNone(x)
- assertIsNotNone(x)
- assertIn(a,b)
- assetNotIn(a,b)
- assertRaises(e, func, *args)

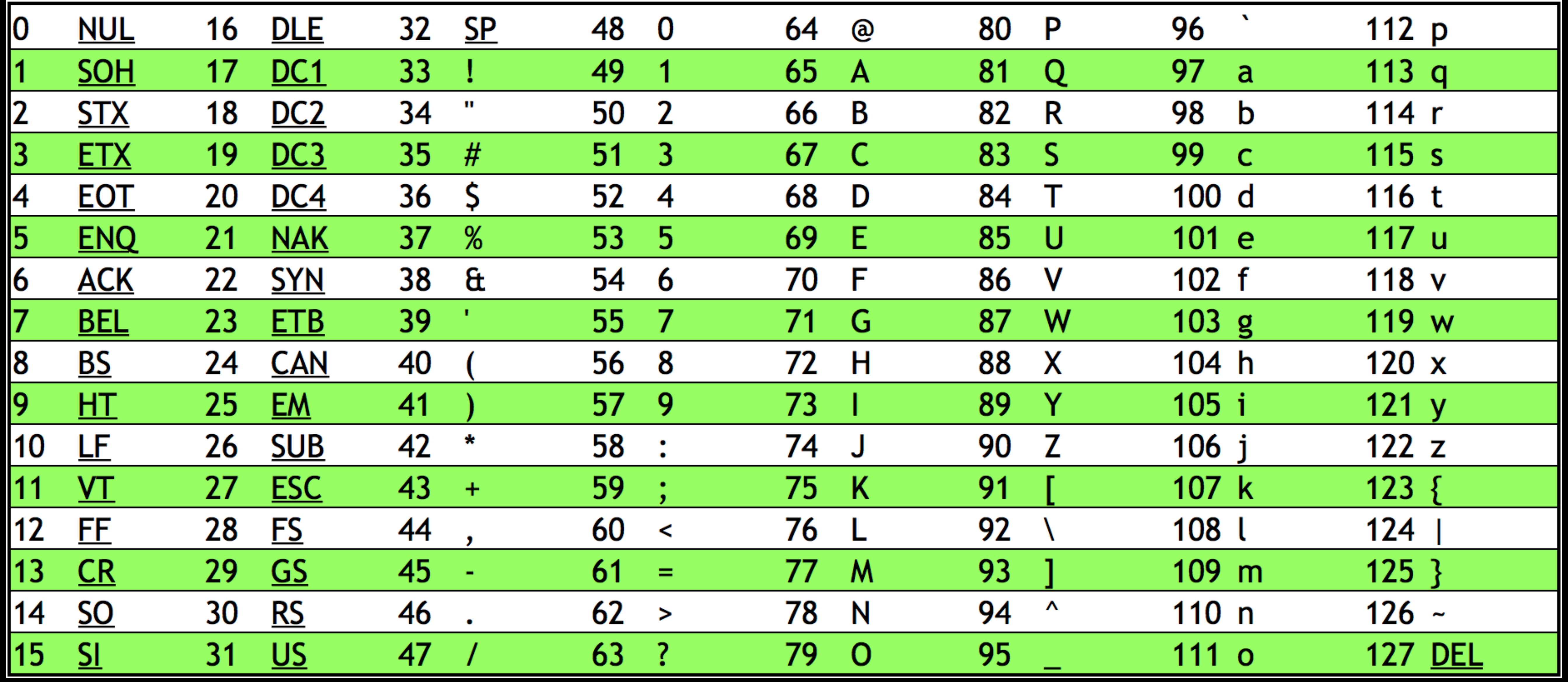
Caesar Cipher



Credit: https://brilliant.org/wiki/caesar-cipher/

ASCII

- American Standard Code for Information Exchange
- ‘Character encoding’ that maps English characters to numbers



0	NUL	16	DLE	32	SP	48	0	64	@	80	P	96	`	112	p
1	SOH	17	DC1	33	!	49	1	65	A	81	Q	97	a	113	q
2	STX	18	DC2	34	"	50	2	66	B	82	R	98	b	114	r
3	ETX	19	DC3	35	#	51	3	67	C	83	S	99	c	115	s
4	EOT	20	DC4	36	$	52	4	68	D	84	T	100	d	116	t
5	ENQ	21	NAK	37	%	53	5	69	E	85	U	101	e	117	u
6	ACK	22	SYN	38	&	54	6	70	F	86	V	102	f	118	v
7	BEL	23	ETB	39	'	55	7	71	G	87	W	103	g	119	w
8	BS	24	CAN	40	(	56	8	72	H	88	X	104	h	120	x
9	HT	25	EM	41	)	57	9	73	I	89	Y	105	i	121	y
10	LF	26	SUB	42	*	58	:	74	J	90	Z	106	j	122	z
11	VT	27	ESC	43	+	59	;	75	K	91	[	107	k	123	{
12	FF	28	FS	44	,	60	<	76	L	92	\	108	l	124	|
13	CR	29	GS	45	-	61	=	77	M	93	]	109	m	125	}
14	SO	30	RS	46	.	62	>	78	N	94	^	110	n	126	~
15	SI	31	US	47	/	63	?	79	O	95	_	111	o	127	DEL

Credit: http://www.asciichart.com

Modular Arithmetic

- The modulo operation (%) returns the remainder (modulus) after division of one number by another.

- 5 % 2 == 1, because 5 / 2 == 2 with a remainder of 1.

Modular Arithmetic in Python

- Python’s modulo operation obeys the following two rules:
  - (a // b) * b + (a % b) == a
    - // indicates floor division, which always rounds down.
  - a % b has the same sign as b

Modular Arithmetic in Python

- Calculating –5 % 26:
    - (–5 // 26) * 26 + (–5 % 26) == –5
    - –1 * 26 + (–5 % 26) == –5
    - –26 + (–5 % 26) == –5
    - –5 % 26 == 21