Chapter 5

Operators and Expressions

The purspose of computing is insight, not numbers.
Richard Hamming

Chapter Objectives

- Learn Python's arithmetic, string, relational, logical, bit–wise operators
- Learn Python's sequence operators (with examples from the string type)
- Learn about implicit/explict conversions between values of different types
- Learn how to build complicated expressions from operators and functions
- Learn how operator precedence/associativity control order of evaluation
- Learn how Python translates operators into equivalent method calls
- Learn about the built in `eval(str) -> object` function
- Learn how to write single–statement functions that abstract expressions

5.1 Introduction

Expressions in programming are like formulas in mathematics: both use values (in Python literals and names bound to values) to compute a result. But unlike mathematics, expressions in Python can compute results of a wide variety to types (e.g., boolean and string) not just mathematical results.

In this chapter we will study the structure of expressions (their syntax) and the meaning of expressions (their semantics). Expressions are like molecules: they are built from atoms (literals and names, which have values) and chemical bonds that hold the atoms in place (operators and function calls). We understand expressions by understanding their components.

We have already studied literals and names; we will now study the syntax and semantics of a laundry list of operators and then learn the general rules that we can use in Python to assemble and understand complicated expressions. As an aid to studying the laundry list of operators, we will use categorize each operator as arithmetic, relational, logical, bit–wise, or sequence (e.g, string).

When we learn to build complicated expressions in Pythong, we will study oval diagrams as the primary analytic tool for understand them. Oval diagrams help us understand whether an expression is syntactically correct and what value the expression evaluates to (computes). We will use our understanding of

oval diagrams to help us learn to translate complicated formulas into Python expressions.

We start by using the terminiology of operators "returning values" as we did for functions, and also speak about operators evaluating to a value. Ultimately we will learn that operator calls are translated by Python into method calls.

Finally, we will use our new knowledge of expressions to write more complicated assignnment statements, and later find we will find that `bool` expressions are used in all interesting control structures, which we study in Chapter **??**.

5.2 Operators

In this section we will examine details of all the Python operators introduced in Table 2.3. We classify these operators, both symbols and identifiers, into four categories and examine them separately: Arithmetic (`+ - * / // % **`), Relational: (`== != < > <= >= is in`), Logical (`and not or`), and finally Bit–wise (`& | ~ ^ << >>`).

When describing these operators we will uses a header-like notation called a "prototype". A prototype includes the symbol/name of the operator, the type(s) of its operand(s), and the type of result it returns; unlike headers, prototypes do not specify parameter names nor default argument values. As with headers, if a symbol/name has multiple prototypes we say what it is overloaded. Most operators are overloaded.

For example, the prototype `< (int, int) -> bool` specifies that one of the overloaded prototypes of the less–than operator has two `int` operands and returns a `boolean` result. Semantically, this operator returns a result of `True` when its left operand is strictly less than its right operand; otherwise it returns a result of `False`: so `1 < 3` returns a result of `True` and `3 < 1` returns a result of `False`.

We categorize prototypes by the number of operands they specify. Python's operator prototypes specify either one or two operands: operators with one operand are called "unary" operators (think uni-cycle) and operators with two operands are called "binary" operators (think bi-cycle). We write unary operators in "prefix" form (meaning before their single operand) and we write binary operators in "infix" form (meaning in–between their two operands): the first type in a prototype specifies the left operand and the second specifies the right operand.

5.2.1 Arithmetic Operators

This section explains the prototypes (syntax) and semantics of all the arithmetic operators in Python, using its three numeric types: `int`, `float` and `complex`. Some are familiar operators from mathematics, but others are common only in computer programming. The end of this section discusses how Python's arithmetic operators apply to `bool` values and how Python interprets operands of mixed types (e.g., `3 + 5.0`)

`Addition`: The `+` operator in Python can be used in both the binary and unary form. The binary form means add, returning a result that is the standard arithmetic sum of its operands. The unary form means identity, returning the same value as its operand.

Prototype	Example
`+ (int,int) -> int`	`3 + 5` returns the result `8`
`+ (float,float) -> float`	`3.0 + 5.0` returns the result `8.0`
`+ (complex,complex) -> complex`	`3j + 5j` returns the result `8j`
`+ (int) -> int`	`+3` returns the result `3`
`+ (float) -> float`	`+3.0` returns the result `3.0`
`+ (complex) -> complex`	`+3j` returns the result `3j`

`Subtraction`: The `-` operator in Python can be used in both the binary and unary form. The binary form means subtract, returning a result that is the standard arithmetic difference of its operands: left operand minus right operand. The unary form means negate, returning the negated value as its operand: zero to zero, positive to negative, and negative to positive.

Prototype	Example
`- (int,int) -> int`	`3 - 5` returns the result `-2`
`- (float,float) -> float`	`3.0 - 5.0` returns the result `-2.0`
`- (complex,complex) -> complex`	`3j - 5j` returns the result `-2j`
`- (int) -> int`	`-3` returns the result `-3`
`- (float) -> float`	`-3.0` returns the result `-3.0`
`- (complex) -> complex`	`-3j` returns the result `-3j`

Note that we will write negative values like `-2` as the result of computations, even though they are not literals.

`Multiplication`: The `*` operator in Python can be used only in the binary form, which means multiplication, returning a result that is the standard arithmetic product of its operands.

Prototype	Example
`* (int,int) -> int`	`3 * 5` returns the result `15`
`* (float,float) -> float`	`3.0 * 5.0` returns the result `15.0`
`* (complex,complex) -> complex`	`3j * 5j` returns the result `(-15+0j)`

Complex numbers, which have real (`r`) and imaginary (`i`) parts display in Python as `(r + ij)`; the product of two purely imaginary numbers has a real part that is the negative product of the imaginary parts with zero as their imaginary part: `3j * 5j` returns the result `(-15+0j)`, which is stored and displayed as a complex number (see the prototyp), even though its imaginary part is `0`.

`Division`: The `/` operator (slash) in Python can be used only in the binary form, which means division, returning a result that is the standard arithmetic quotient of its operands: left operand divided by right operand.

Prototype	Example
`/ (int,int) -> float`	`3 / 5` returns the result `0.6`
`/ (float,float) -> float`	`3.0 / 5.0` returns the result `0.6`
`/ (complex,complex) -> complex`	`3j/5j` returns the result `(.06+0j)`

Notice here that dividing two `int` values always returns a `float` result (unlike addition, subtraction, and multiplication): so even `4 / 2` —which has an

integral value— returns the result `2.0`.

`Floor Division`: The `//` operator (double slash) in Python can be used only in the binary form, which also means division, but returning an integral result of the standard arithmetic quotient of its operands: left operand divided by right operand.

Prototype	Example
`// (int,int) -> int`	`3 // 5` returns the result `0`
`// (float,float) -> float`	`3.0 // 5.0` returns the result `0.0`

In fact, writing `x//y` is a simpler way to write the equivalent `math.floor(x/y)`, which calls the `math.floor` function on the actual quotient of `x` divided by `y`. The `math.floor` function of an integral value is that value; but for a non–integral is the closest integral value lower (think floor) than it; so `math.floor(1.5)` returns a result of `1` and `math.floor(-1.5)` returns a result of `-2`. Note that `int(1.5)` also returns a result of `1`, but `int(-1.5)` returns a result of `-1`: both throw away the decimal part, which actually raises a negative value. Finally, `math.ceil` is the opposite of `math.floor`, raising non–integral values.

Why is the floor division operator useful? If we want to make change for `84` cents, we can use floor division to determine how many quarters to give: `84//25`, which returns a result of `3`: the same as `math.floor(3.36)`.

`Modulo`: The `%` operator in Python can be used only in the binary form, which means remainder after the left operand divided by the right operand.

Prototype	Example
`% (int,int) -> int`	`8 % 3` returns the result `2`
`% (float,float) -> float`	`8.0 % 3.0` returns the result `2.0`

In Python, the sign of the returned result is the same as the sign of the divisor and the magnitude of the resulted result is always less than the divisor: so `17%5` returns a result of `2` because when `17` is divided by `5` the quotient is `3` and the remainder is `2`; but `17%-5` returns a result of `-2` because the divisor is negative. Mathematically `a%b = a - b*(a//b)` for both `int` and `float` operands. Most uses of the `%` operator in programming have two positive operands.

Why is the modulo division operator useful? If we want to make change for `84` cents, we can use modulo to determine how much change is left to make after giving quarters: using the formula above, the result is `84 - 25*(84//25)` where we subtract from `84` the product of `25` times the number of quarters returned as change, which computes the amount of change given by `3` quarters.. So, in the problem of making change, both the floor division (integer quotient) and modulo (remainder after division) operators are useful.

`Power`: The `**` operator in Python can be used only in the binary form, which means power returning a result that is the left operand raised to the power of the right operand.

Prototype	Example
`** (int,int) -> int`	`3 ** 5` returns the result `243`
`** (float,float) -> float`	`3.0 ** 5.0` returns the result `243.0`
`** (complex,complex) -> complex`	`3j ** 5j` returns the result `(0.00027320084764143374-0.00027579525809376897j`

Here are some general observations about arithmetic operators. For most operators (`+ - * // % **` but not `/`), their result types match their operand types.

Likewise, the most familiar arithmetic operators (`+ - * / **` but not `//` or `%`) have prototoypes for all three numeric types: `int`, `float`, and `complex`; and for these familiar operators, their semantics are the same as their semantics in mathematics. The two special quotient and remainder operators have simple and intuitive semantics when their operands are both positive.

5.2.2 Conversions: Implicit and Explicit

Before finishing our discussion of arithmetic operators in Python, there are two topics we must cover: arithmetic on boolean operands and arithmetic on mixed–type operands. Both involve the general concept of implicit "type–conversion": Python arithmetic operators implicitly convert values of the type `bool` to `int`, `int` to `float`, `float` to `complex` when necessary. Think about these types as a hierarchy with `complex` at the top and `bool` at the bottom: any value from a lower type can be converted to an equivalent value from an upper type: e.g, `True` converts to `1` converts to `1.0` converts to `(1+0j)`. Note that conversion the other way might not be equivalent: e.g., there is no way to convert `1.5` to an equivalent integer.

`Arithmetic on Boolean Values`: To perform any arithmetic operators on boolean values, first they are promoted (up the hierarchy) to integers: `False` is promoted to `0` and `True` is promoted to `1`: so, in `True * False` Python promotes the boolean values to `1 * 0`, which returns the result `0`. The term "promotion" implies movement up the numeric type hierarchy.

`Arithmetic on Mixed Types`: To perform any arithmetic operator on two operands that do not have the same type, the lower type operand is promoted to the type of the higher type operand: so, in `True * 1.5` the boolean value `True` is promoted to the integer value `1`, which is promoted to the floating–point value `1.0`, which is then —satisfying one of the prototypes for `*`— multiplied by `1.5`, which returns the result `1.5`.

Conversions in these two cases are implicit: they are performed automatically by Python, to be able to satisfy the prototypes of the operators. But we can also explicitly convert between numeric types, both up and down the numeric type hierarchy. In fact, we can use `bool`, `str`, and each numeric type name (`int`, `float`, and `complex`) as the name of a function that converts values to that type from other types. We have already seen some examples of explict conversion in Section 4.5.3, which discussed converting strings input from the user to values of the type `int` and `float`.

The table below summarizes the prototype and semantics of each conversion function. Note that when any conversion function is passed an argument of its own type, it returns a reference to its argument: e.g., `int(3)` returns `3`. See Section 4.5.3, for how `int` and `float` convert strings to their types; `complex` is similar: e.g., `complex('(3+5j)')` returns `(3+5j)`.

Prototype	Semantics
`str (T) -> str`	returns string showing literal (possible signed for numeric types)
`int (T) -> int`	returns `0` for `False`, `1` for `True`; truncated `float`; truncated real–part of `complex`
`float (T) -> float`	returns `0.0` for `False`, `1.0` for `True`; equivalent for `int`; real–part for `complex`
`complex (T) -> complex`	returns `(0+0j)` for `False`, `(1+0j)` for `True`; equivalent `int` and `float`
`bool (T) -> bool`	`False` for empty string and zero value, `True` for non–empty string and non–zero value

We have seen that when converting upward in the numeric type hierarchy, no information is lost. But, when converting downward, information can be lost: converting from `complex` to `float` the imaginary part is lost; converting from `float` to `int` the decimal part is lost; converting from `int` to `bool` zero (or the empty string) converts to `False` and any non–zero value (or any non–empty string) converts to `True`.

A surprising result of these semantics is that `bool('False')` returns a result of `True`; check it. In Section ?? we will learn how to convert `'False'` to `False` and `'True'` to `True` correctly.

Experiment with the Python interpreter to explore the arithmetic operators and conversion functions.

5.2.3 String Operators

Two of Python's arithmetic operators (`+` and `*`) are also overloaded further, allowing string operands and producing string results.

`Concatenation`: The `+` operator has the prototype `+ (str,str) -> str` and it returns a result that is a string containning all the character in its left operand followed by all the character in its right operand: e.g., `'acg' + 'tta'` returns the result `'acgtta'`. Note that as with all operators, neither operand changes, but a new value object (of type `str`, and all characters in both operands) is returned.Sometimes we call this just "catenation".

`Shallow Copy`: The `*` operator has the two prototypes `* (int,str) -> str` and `* (str,int) -> str`, which are symmetic; it returns a result that is a string containing all the characters in its string operand replicated the number of times specifed by its integer operand: e.g., both `'acg' * 3` and `3 * 'acg'` return the result `'acgacgacg'`. If the integer operand is zer or negative, the returned result is `''` (the empty string).

Technically, both these operators work with any sequence type, not just `str`: the only sequence type we know in Python (which are sequences of characters). We will generalize these operators to other sequence types when we learn them.

5.2.4 Relational Operators

Relational operators always return a boolean result that indicates whether some relationship holds between their operands. Most relational operators are symbols (`== != < > <= >=` but two are identifiers (`is in`) and one is a compound identfier (`not in`). The table below lists the prototypes and the relational operators to which they apply. All their prototypes are similar: same-type operands returning a boolean result.

Prototype Form	Operators Allowsable for *r-op*
r-op `(int,int) -> bool`	`== != < > <= >= is`
r-op `(float,float) -> bool`	`== != < > <= >= is`
r-op `(complex,complex) -> bool`	`== != is`
r-op `(bool,bool) -> bool`	`== != < > <= >= is`
r-op `(str,str) -> bool`	`== != < > <= >= is in not in`

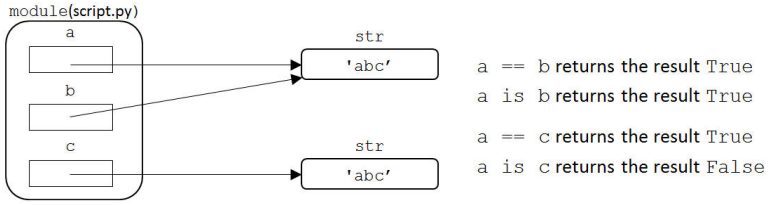
Another way to classify these operators is equality operators (`== !=`), ordered comparison (`< > <= >=`), identity (`is`), and inclusion (`in` and also `not in`). The table belows lists the relational operators, their names, and their semantics.

r-op	Name	Semantics
`==`	equals	`True` when the operands have equal values
`!=`	not equals	`True` when the operands have unequal values
`<`	less than	`True` when the left operand is less than the right operand
`>`	greater than	`True` when the left operand is greater than the right operand
`<=`	less than or equals	`True` when the left operand is less than or equal to the right operand
`>=`	greater than or equals	`True` when the left operand is greater than or equal to the right operand
`is`	object identity	`True` when the left operand refers to the same object as the right operand
`in`	inclusion	`True` when the left string appears in the right string: e.g., `'gt' in 'acgtta'` is `True`.

`Equality and Identity`: The equality operators (`== !=`) are the easiest to understand: they have prototypes for each type in Python and return a result based on the whether the two operands store references to **objects storing the same value**. The identity operator (`is`) is similar but more restrictive: it also has prototypes for each type, but returns a result based on whether the two operands store references to the **same** object. Read over these last two sentences carefully, because the difference is subtle. Figure 5.1 illustrates the difference between `==` and `is`: note that there are two `str` objects that each store the same value: `'abc'`. This picture is the result of Python executing the following script.

```
1  a = 'abc'
2  b = a
3  c = input('Enter␣String:')  # enter abc; Python creates a new value object
```

Figure 5.1: Illustrating the difference between `==` and `is`



There are a few cases where Python creates two different objects that both store the same value: input of strings from the console; writing integers literals with many digits; writing floating–point and complex literals. But in most programs when we compare values we want to check equality not identity, so we don't care whether Python creates multiple objects. But, in some advanced programs we will want to check identity, and now we know how. Finally, note that if the `is` operator returns `True` the `==` operator must return `True`: if two references refer to the same object, then the value of that object must be the same as itself. Experiment with these concepts in the interpreter: try `>>> a = 1.5 >>> b = 1.5 >>> a == b >>> a is b`.

Ordered Comparison: The ordered comparsion operators (`< > <= >=`) are simple to understand for the numeric types (`int` and `float`; there is no prototype for `complex`) and `bool`: they have the same meanings as mathematics, with `False` considered to be less than `True` –which reinforces the idea `False` promoting to `0` and `True` promoting to `1`. If we try an ordered comparison

of `complex` values, Python will raise the `TypeError` exception, describing this type as unorderable.

Let's now learn the algorithm for comparing strings, which are sequences of zero or more characters. First, we must learn to compare the individual characters in strings: we do so according to their decimal values, illustrated in the ASCII table below. Although Python uses unicode characters, we will still stick to the ASCII subset, and use the following table to compare characters.

Figure 5.2: Character Values in ASCII

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

To answer the question, "How does the character `a` compare to the character `1`?" we translate it into the question, "How does the decimal value of the character `a` compare to the decimal value of the character `1`?" According to the ASCII table, the decimal value of the character `a` is `97` and the decimal value of the character `1` is `49`, so `a` is greater than `1`.

Don't memorize the ASCII table, but know that the digit characters are less than the upper–case characters, which are less than the lower–case characters; and within each of these character groupings, decimal values increase: e.g., `A` is less than `B` is less than ... is less than tt Z. In fact, the `builtins` module defines the `ord(str) -> int` function which returns the ASCII value of any single character: `ord('a')` returns the result `97`; if the argument to `ord` is not a single character, it raises the `TypeError` exception.

Given this information, the algorithm for comparing strings is follows. This is called dictionary or "lexicographic" ordering.

Step 1 Start at the leftmost character of each string; skip all characters that are pairwise the same, trying to find a character that is different.

Step 2 Based on the outcome, compare the strings as follows:

- Step 2A **Found a different character**: the strings compare according to how the different characters compare.
- Step 2B **Found no different characters** (e.g., all characters in one string appear in the other): the strings compare according to their number of characters.

For example, to compare `'anteaters'` to `'ants'` Python finds the first three characters (`'ant'`) are the same but the fourth characters are different: because `e` (from `'anteaters'`) is less than `s` (rom `'ants'`) Python determines `'anteaters'` is less than `'ants'`. Similarly, to compare `'anteater'` to `'ant'` Python finds that all the characters in `'ant'` are in `'anteater'`): because `'anteater'` has `8` characters and `'ant'` has `3` characters, Python determines `'anteater'` is greater than `'ant'`. Finally, notice that to compare `'ant'` to `'ant'` Python finds that all the characters in `'ant'` are in `'ant'`: because both `'ant'` strings have `3`, Python determines `'ant'` is equal to `'ant'`.

The functions `builtins.min` and `builtins.max` both take one or more arguments —each of their headers contains `(*args)`— so long as they are types on which Python can perform ordered comparisions. They return a reference to the minimum/maximum argument respectively: e.g., `min(1,-2,4)` returns a result of `-2` and `min('aunteater','aunt','ant')` returns a result of `'ant'`. If any two arguments cannot be compared, Python raises a `TypeError` exception, including a comment about `unorderable types`: Typing `>>> min(1,'a')` in the Python interpreter displays `TypeError: unorderable types: int() < str()` in the console.

Relations on Mixed–Types: To perform any relational operator on two operands that do not have the same type, the lower type operand is promoted to the type of the higher type operand (as for arithmetic operators): so, in `3 < 5.0` the integer value `3` is promoted to the floating–point value `3.0`, which is then —satisfying one of the prototypes for `<`— compared to `5.0`, which returns the result `True`. Note that there is no implicit conversion to or from the string type.

Inclusion: The inclusion operators (`in`, and `not in`) work on any sequence type in Python. They are opposites: when one returns `True` the other returns `False`. For now, the only sequence type we know is `str` (which are sequences of characters), so that is how we will explore these operators. We will generalize these operators to other sequence types when we learn them.

The `in` operator determines if all the characters in its left operand appear in the same order, adjacent, in it right operand. In the simplest case, the left operand has just one character. So, assuming `s` is bound to a `str` object, `'a' in s` returns whether or not the character `a` appears anywhere in `s`. Likewise, `'acg' in s` returns whether or not the character sequence `cgt` appears anywhere in `s`: it returns `True` if `s` is bound to `'acgtta'`, but returns `False` if `s` is bound to `'attgca'`.

Also, according to this definition, for every string bound to `s`, `'' in s` returns `True`: because there is no character in the empty string that is not in `s`, because the empty string has no characters for which this test can fail.

The `in` operator is related to the `builtins.str.find` function discussed in Section 4.2, which returns the index of a substring in a string, or `0` if the substring is not in the string. The function call `str.find(a,b)` returns a positive value exactly when `a in b` returns `True`.

Finally, we must learn to think of an operator like `<` just like `+`: both have two operands and both compute a result (in the former case a boolean; in the later case some numeric type). So we say `3 < 5` computes the result `True` just as we would say `3 + 5` computes the result `8`. We must learn to think of all Python operators as computing a value, regardless of their operand and result types.

5.2.5 Logical Operators

Python's logical operators mostly use boolean operands to produce boolean results. Each of these operators is written as an identifier.

`Conjuction/And`: The `and` operator in Python can be used only in the binary form, which means conjunction, returning a result of `True` only if both of its operands are `True`. Its boolean prototype is `and (bool,bool) -> bool`.

`Disjunction/Or`: The `or` operator in Python can be used only in the binary form, which means disjunction, returning a result of `True` if either or both of its operands are `True`. Its boolean prototype is `or (bool,bool) -> bool`.

`Negation`: The `not` operator in Python can be used only in the unary form, which means negation, returning the a result that is the opposite of its operand. Its boolean prototype is `not (bool) -> bool`.

The following table summarizes the semantics of these operators for relevant boolean operands denoted a and `b`. Note that the `not` operator is unary, so it looks a bit different in the truth table (and see the *).

a	b	a `and` b	a `or` b	`not` b	a `!=` b (exclusive or)
`False`	`False`	`False`	`False`	`True`	`False`
`False`	`True`	`False`	`True`	`False`	`True`
`True`	`False`	`False`	`True`	*	`True`
`True`	`True`	`True`	`True`	*	`False`

*These rows would duplicate the top two rows, where b has the same values.

Programmers must memorize these truth–tables to be able to analyze complicated expressions that use logical operators. Here are two short-cuts. The result returned by the `and` operator is `True` only when both of its operands are `True`; more intuitively, if I say to my son, "You may play if your homework is done **and** your room is clean" the only time he can play is is both are `True`. The result returned by the `or` operator is `False` only when both of its operands are `False`; more intuitively, if I say to my son, "You may play if your homework is done **or** your room is clean" he cannot play if both are `False`, but can play either or both are `True`: this is called "inclusive or". The "exclusive or" is `True` if one operands is `True` and the other operands is `False`: there is no exclusive or

operator, but `!=` on booleans has the same semantics (shown in the last column of the table above).

In fact, the semantics of these these operators are a bit more complicated, because we can apply them to operands of any types, not just booleans. Recall the semantics of the `bool` conversion function discussed in Section 5.2.2.

When evaluating the `and` operator, Python first evaluates its left operand: if calling the `bool` conversion function with this value is `False`, the result returned is the value of the left operand; otherwise, the result returned is the value of the right operand. Notice that for boolean left and right operands, these semantics match the semantics of the table shown above; check all four examples.

When evaluating the `or` operator, Python first evaluates its left operand: if calling the `bool` conversion function with this value is `True`, the result returned is the value of the left operand; otherwise, the result returned is the value of the right operand. Again, for boolean left and right operands, these semantics match the semantics of the table shown above; check all four examples.

Experiment with the Python interpreter to explore these logical operators for both boolean and non–boolean operands.

5.2.6 Bit–wise Operators

Bit–wise operations (`& | ~ ^ << >>`) compute their results on the individual bits in the binary representations of their integer operands. We can illustrate all four possible combinations of left and right bits using the operands `3` (the binary literal `0b0011`) and `3` (the binary literal `0b0101`).

Bit–wise	Prototype	Semantics (integers shown as bits)
and	`& (int,int) -> int`	`0011 & 0101` returns the result `0001`
inclusive or	`| (int,int) -> int`	`0011 & 0101` returns the result `0111`
not	`~ (int) -> int`	`~01` returns the result `10`
exclusive or	`^ (int,int) -> int`	`0011 & 0101` returns the result `0110`
shift left	`<< (int,int) -> int`	`101 << 2` returns the result `10100` (similar to `5 // 2**2`)
shift right	`>> (int,int) -> int`	`101 >> 2` returns the result `1` (similar to `5 * 2**2`)

The last two operators shift the left binary operand either left or right by the number of bits specified by the right operand. For shifting left, `0s` are added on the right; for shifting right, bits are removed from the right. Shifting positive numbers left/right is like multiplying/dividing by 2; this is similar to how decimal numbers are shifted left/right when they are multiplied/divided by `10`. Because of the way negative binary numbers are typically stored (twos–complement), shifting negative numbers right does not work as expected: e.g., `-1>>2` returns a result of `-1`!

Note that the function `builtins.bin(x : int) -> str` returns a result that is the binary string equivalent of `x`: e.g., `bin(5)` returns a result of `'0b101'`. So `bin(3 ^ 5 )` returns the result `'0b110'`. For any name `x` bound to an integer `int(bin(x),base=0)` returns the same result as `x`; where `base=0` means to interpret the first argument as string specifying a integer literal (which might start with `0` and a base indicator.

In most simple programs we will not use bit–wise operators. But, in some

advanced programs, in which we must understand the binary representation of data, we may use bit–wise operators.

5.2.7 Delimiters: Sequence Operators and Reassignment

Python uses the delimiters `[]` (often with `:`) as sequence operators to index an element or subsequence of any sequence type. Again, because the only sequence type we know is `str`, we will use that type (as we did in Section 5.2.4) to explore these operators; we will generalize these operators to other sequence types when we learn them. These operators are called "distfix" (as opposed to infix or prefix) because they are written distributed around their operands. Assuming the names `s` is bound to a string, there are three forms of this operator, each described in the box below and followed by many examples.

But first, to understand the semantics of these operators, we must focus on two other useful pieces of information about `str` (and all sequence types). First, the sequence of characters in strings are indexed by integer values, with the first character indexed by `0`. Second, the function `builtins.len(str) -> int` returns a result that is the number of characters in the string (generally it works on all sequence types, returning the number of values in the sequence): e.g., `len('')` returns a result of `0` and `len('acgtta')` returns a result of `6`. Because the first character of string `s` is indexed by `0`, the index of the last character is `len(s) - 1`.

Form	Syntax	Semantics
index	`s[int1] -> str`	Returns a one–character string at index `int` if `int` $\leq$ `len(s)`; otherwise Python raises the `IndexError` exception
slice	`s[int1:int2] -> str`	Returns a substring with those characters in `s` at indexes `i` such that `int1` $\leq$ `i` $<$ `int2`; if `int1` is omitted/$>$`len(s)`, its value is `0`/`len(s)`; if `int2` is omitted/$>$`len(s)`, its value is `len(s)`; negative values for either `int` specify an index that is relative to the **end** of the string, and equivalent to `len(s) + int`: e.g., `-1` for an `int` is equivalent to `len(s) - 1`
slice/step	`s[int1:int2:int3] -> str`	Returns a result similar to slice, but with the first character from index `int1`, the second from index `int1+int3`, the third from index `int1+2*int3`, ... until the index reaches or exceeds `int2`; for negative step values, if `int1` is omitted/$\geq$`len(s)` its value is `len(s) - 1` and if `int2` is omitted its value is `0`; if `int3` is omitted, its value is `1`, and if it is `0` Python raises the `ValueError` exception

To understand these semantics better, let's explore some examples of each form. When analyzing sequence operators, it is useful to illustrate the strings they operate on as follows (here for the string `'acgtta'`):

string	'acgtta'					
index	0	1	2	3	4	5
character	'a'	'c'	'g'	't'	't'	'a'

Now, here are the examples that illustrate most of the interesting semantics of these operators using string `s`. Unfortunately, it is confusing when the first character in this string is at index `0` and last character is at index `len(s)-1`; programmers must learn to overcome this confusion. Experiment in the Python interpreter with other strings and integer operands.

Operation	int_1	int_2	int_3	used indexes	Result	Comments
`s[0]`	`0`	NA	NA	`0`	`'a'`	First character
`s[1]`	`1`	NA	NA	`1`	`'c'`	Second character
`s[-1]`	`5 (6-1)`	NA	NA	`5`	`'a'`	Last character
`s[6]`	`6`	NA	NA	`6`	NA	raises `IndexError` exception
`s[0:6]`	`0`	`6`	`1`	`0, 1, 2, 3, 4, 5`	`'acgtta'`	All characters
`s[:]`	`0`	`6`	`1`	`0, 1, 2, 3, 4, 5`	`'acgtta'`	All characters
`s[1:3]`	`1`	`3`	`1`	`1, 2`	`'cg'`	Second and third characters
`s[-4:10]`	`2 (6-4)`	`6 (len(s))`	`1`	`3, 4, 5`	`'gtta'`	Fourth from last to the end
`s[::2]`	`0`	`6`	`2`	`0, 2, 4`	`'agt'`	Every other character forward from index `0`
`s[1:5:2]`	`1`	`5`	`2`	`1, 3`	`'ct'`	2 characters, every other forward from index `1`
`s[::-2]`	`5`	`0`	`-2`	`5, 3, 1`	`'atc'`	Every other character backward from last index

problems: standard operators: no conversion; e.g., 1 + 'a' TypeError: unsupported operand type(s) for +: 'int' and 'str' boolean mixed /2**N bool('False'): strange; see eval 0*4 is 0; what string when replicated 4 ties is the same string 11 ¡ 5 vs str(11) ¡ str(5), how does " compare

5.3 Expressions

Structure Evaluation

5.3.1 Oval Diagrams

5.3.2 Operator Precedence and Associativity

Operator Precedence

Dictionary/Set + their Comprehensions [] List + its Comprehension () Tuple + its Comprehension x.attr Attribute reference x(...) Call: function, method x[i:j:k] Slicing x[i] Indexing x**y Power (exponentiation) x Bitwise not -x,+x Negation, Identity * + - Add/catenation, subtract/set-difference x¡¡y x¿¿y Shift x left/right by y bits & Bitwise AND/set-intersection Ĵbitwise NOT/set-symmetric-difference(XOR) — Bitwise OR/set-union ¡ ¡= ¿ ¿= Ordered comparison/set-subset-superset == != Value equality/inequality is, is not Object identity tests in, not in Membership tests (on interables and Strings) not Logical negation and Logical and (short-circuit) or Logical not (short-circuit) x if b else y Ternary selection operator lambda a : e Un-named(anonymous) function generation (lambda x,y,z : ...) yield Generator function send protocol

Finally, we can "chain" relational operators together. a r-op b r-op c ... a r-op b and b r-op c and .... chaining: a¡b¡c = a ¡ b and b ¡ c (expressions covered later)

All operators are left associative except exponentiation (right associative) ordered comparisons (which are chained)

Operators are automatically converted up: bool -¿ integer -¿ floating-point -¿ complex

5.3.3 Common Mistakes

Problems: what if 'not in' not included in Python: not (a in b)

5.4 Operators as Method/Function Calls

How operators get new meanings

5.5 The `eval` function

eval('False')

5.6 Functions abstracting Expressions

Simplest aspect def header : return expression

body of function defintion is statement or statement sequence that can refer to parameters; executed in namespace of function object with parameters bound to their arguments

now cover/use return is statement: much more in next chapter

binds name in header to a function object of parameters and body that returns the result specified by the body See distance function See predicate module: `def non_negative(x : int) -> boolean : x >= 0` ...?

Chapter Summary

Summary here

Chapter Exercises

1. First

   Answer:

use prototypes x+y = x.__add__(y) overloading later precedence

later show how + =¿ __add__ so a + b =¿ a.__add__(b) =¿ type(a).__add__(a,b) =¿ int.__add__(a,b) finding the __add__ method in the int class; what if a is a float?