Chapter 2

Tokens and Python's Lexical Structure

The first step towards wisdom is calling things by their right names.
Chinese Proverb

Chapter Objectives

- Learn the syntax and semantics of Python's five lexical categories
- Learn how Python joins lines and processes indentation
- Learn how to translate Python code into tokens
- Learn technical terms and EBNF rules concerning to lexical analysis

2.1 Introduction

Python's lexical structure comprises five lexical categories

We begin our study of Python by learning about its lexical structure and the rules Python uses to translate code into symbols and punctuation. We primarily use EBNF descriptions to specify the syntax of Python's five lexical categories, which are overviewed in Table 2.1. As we continue to explore Python, we will learn that all its more complex language features are built from these same lexical categories.

Python translates characters into tokens, each corresponding to one lexical category in Python

In fact, the first phase of the Python interpreter reads code as a sequence of characters and translates them into a sequence of tokens, classifying each by its lexical category; this operation is called "tokenization". By the end of this chapter we will know how to analyze a complete Python program lexically, by identifying and categorizing all its tokens.

Table 2.1: Python's Lexical Categories

Identifier	Names that the programmer defines
Operators	Symbols that operate on data and produce results
Delimiters	Grouping, punctuation, and assignment/binding symbols
Literals	Values classified by types: e.g., numbers, truth values, text
Comments	Documentation for programmers reading code

Programmers read programs in many contexts: while learning a new programming language, while studying programming style, while understanding algorithms —but mostly programmers read their own programs while writing, correcting, improving, and extending them. To understand a program, we must learn to see it the same way as Python does. As we read more Python programs, we will become more familiar with their lexical categories, and tokenization will occur almost subconsciously, as it does when we read a natural language.

When we read programs, we need to be able to see them as Python sees them

The first step towards mastering a technical discipline is learning its vocabulary. So, this chapter introduces many new technical terms and their related EBNF rules. It is meant to be both informative now and useful as a reference later. Read it now to become familiar with these terms, which appear repeatedly in this book; the more we study Python the better we will understand these terms. And, we can always return here to reread this material.

If you want to master a new discipline, it is important to learn and understand its technical terms

2.1.1 Python's Character Set

Before studying Python's lexical categories, we first examine the characters that appear in Python programs. It is convenient to group these characters using the EBNF rules below. There, the *white_space* rule specifies special symbols for non printable characters: ␣ for space; → for tab; and ↩ for newline,which ends one line, and starts another.

We use simple EBNF rules to group all Python characters

White–space separates tokens. Generally, adding white–space to a program changes its appearance but not its meaning; the only exception —and it is a critical one— is that Python has indentation rules for white–space at the start of a line; section 2.7.2 discusses indentation in detail. So programmers mostly use white-space for stylistic purposes: to make programs easier for people to read and understand. A skilled comedian knows where to pause when telling a joke; a skilled programmer knows where to put white–space when writing code.

White–space separates tokens and indents statements

EBNF Description: Character Set

lower ⇐ a|b|c|d|e|f|g|h|i|j|k|l|m|n|o|p|q|r|s|t|u|v|w|x|y|z

upper ⇐ A|B|C|D|E|F|G|H|I|J|K|L|M|N|O|P|Q|R|S|T|U|V|W|X|Y|Z

digit ⇐ 0|1|2|3|4|5|6|7|8|9

ordinary ⇐ _|(|)|$\boxed{\texttt{[}}$|$\boxed{\texttt{]}}$|$\boxed{\texttt{\{}}$|$\boxed{\texttt{\}}}$|+|-|*|/|%|!|&|$\boxed{\texttt{|}}$|~|^|<|=|>|,|.|:|;|$|?|#

graphic ⇐ *lower* | *upper* | *digit* | *ordinary*

special ⇐ ' | " | \

white_space ⇐ ␣ | → | ↩ (space, tab, or newline)

Python encodes characters using Unicode, which includes over 100,000 different characters from 100 languages —including natural and artificial languages like mathematics. The Python examples in this book use only characters in the American Standard Code for Information Interchange (ASCII, rhymes with "ask me") character set, which includes all the characters in the EBNF above.

Although Python can use the Unicode character set, this book uses only ASCII, a small subset of Unicode

Section Review Exercises

1. Which of the following mathematical symbols are part of the Python character set? $+$, $-$, $\times$, $\div$, $=$, $\neq$, $<$, or $\leq$.

   Answer: Only `+`, `-`, `=`, and `<`. In Python, the multiply operator is `*`, divide is `/`, not equal is `!=`, and less than or equal is `<=`. See Section 5.2.

2.2 Identifiers

We use identifiers in Python to define the names of objects. We use these names to refer to their objects, much as we use the names in EBNF rules to refer to their descriptions. In Python we can name objects that represent modules, values, functions, and classes, which are all language features that are built from tokens. We define identifiers in Python by two simple EBNF rules.

Identifiers are names that we define to refer to objects

EBNF Description: *identifier* (Python Identifiers)

id_start ⇐ *lower* | *upper* | _
identifier ⇐ *id_start*{*id_start* | *digit*}

There are also three semantic rules concerning Python identifiers.

Identifier Semantics

- Identifiers are case-sensitive: identifiers differing in the case (lower or upper) of their characters are different identifiers: e.g., `mark` and `Mark` are different identifiers.
- Underscores are meaningful: identifiers differing by only underscores are different identifiers: e.g., `pack_age` and `package` are different identifiers.
- An identifier that starts with an underscore has a special meaning in Python; we will discuss the exact nature of this specialness later.

When we read and write code we should think carefully about how identifiers are chosen. Specifically, here are some useful guidelines.

Identifier Pragmatics

- Choose descriptive identifiers, starting with lower–case letters (upper–case for classes), whose words are separated by underscores.
- Follow the Goldilocks principle for identifiers: they should neither be too short (confusing abbreviations), nor too long (unwieldy to type and read), but should be just the right size to be clear and concise.
- When programmers think about identifiers, some visualize them, while others hear their pronunciation. Therefore, , avoid using identifiers that are homophones, homoglyphs, or mirror images.

  Homophones are identifiers that are similar in pronunciation e.g., `a2d_convertor` and `a_to_d_convertor`. Homoglyphs are identifiers that are similar in appearance: e.g., `all_0s` and `allOs` —0 (zero) vs. upper–case O; same for the digit `1` and the lower–case letter `l`. Mirror images are identifiers that use the same words but reversed: e.g., `item_count` and `count_item`.

2.2.1 Keywords: Predefined Identifiers

Keywords are identifiers that have predefined meanings in Python. Most keywords start (or appear in) Python statements, although some specify operators and others literals. We cannot change the meaning of a keyword by using it to refer to a new object. Table 2.2 presents all 33 of Python's keywords. The first three are grouped together because they all start with upper–case letters.

Keywords are special identifiers with predefined meanings that cannot change

Keywords should be easy to locate in code: they act as guideposts for reading and understanding Python programs. This book presents Python code using bold–faced keywords; the editors in most Integrated Development Environments (IDEs) also highlight keywords: in Eclipse they are colored blue.

Keywords should stand out in code: they act as guideposts for reading and understanding programs

Table 2.2: Python's Keywords

`False`	`class`	`finally`	`is`	`return`
`None`	`continue`	`for`	`lambda`	`try`
`True`	`def`	`from`	`nonlocal`	`while`
`and`	`del`	`global`	`not`	`with`
`as`	`elif`	`if`	`or`	`yield`
`assert`	`else`	`import`	`pass`	
`break`	`except`	`in`	`raise`	

Section Review Exercises

1. Classify each of the following as a legal or illegal identifier. If it is legal, indicate whether it is a keyword, and if not a keyword whether it is written in the standard identifier style; if it is illegal, propose a similar legal identifier —a homophone or homoglyph.

a. `alpha`	g. `__main__`	m. `2lips`
b. `raise%`	h. `sumOfSquares`	n. `global`
c. `none`	i. `u235`	o. `%_owed`
d. `non_local`	j. `sum of squares`	p. `Length`
e. `x_1`	k. `hint_`	q. `re_turn`
f. `XVI`	l. `sdraw_kcab`	r. `_0_0_7`

Answer:

a. Legal	g. Legal (special: starts with _)	m.Illegal: `tulips` or `two_lips`
b. Illegal: `raise_percent`	h. Legal: `sum_of_squares`	n. Keyword
c. Legal (not keyword `None`)	i. Legal	o. Illegal: `percent_owed`
d. Legal (not keyword `nonlocal`)	j. Illegal (3 tokens; use h.)	p. Legal: `length`
e. Legal	k. Legal	q. Legal (not keyword `return`)
f. Legal: `xvi`	l. Legal	r. Legal (special: starts with _)

2.3 Operators

Operators compute a result based on the value(s) of their operands: e.g., `+` is the addition operator. Table 2.3 presents all 24 of Python's operators, followed by a quick classification of these operators. Most operators are written as special symbols comprising one or two *ordinary* characters; but some relational and logical operators are instead written as keywords (see the second and third lines of the table). We will discuss the syntax and semantics of most of these operators in Section 5.2.

Operators compute a result based on the value(s) of their operand(s); we primarily classify keywords that are relation and logical operators as operators

Table 2.3: Python's Operators

`+`	`-`	`*`	`/`	`//`	`%`	`**`		arithmetic operators
`==`	`!=`	`<`	`>`	`<=`	`>=`	`is`	`in`	relational operators
`and`	`not`	`or`						logical operators
`&`	`|`	`~`	`^`	`<<`	`>>`			bit–wise operators

We can also write one large `operator` EBNF rule using these alternatives.

EBNF Description: *operator* (Python Operators)

operator ⇐ +|-|*|/|//|%-|**|=|!=|<|>| <=|>=|&|$\boxed{|}$|~|^|<<|>|and|in|is|not|or

2.4 Delimiters

Delimiters are symbols that perform three special roles in Python: grouping, punctuation, and assignment/binding of objects to names. Table 2.4 presents all 24 of Python's delimiters, followed by a quick classification of their roles, with the *delimiter* EBNF rule following. Grouping and punctuation delimiters are all written as one character symbols. The assignment/binding delimiters include `=` and any multi–character delimiter that ends with an `=` sign. We will discuss the syntax and semantics of most of these delimiters later in this book.

Delimiters are either grouping symbols, punctuation symbols, or symbols that assign/bind objects to names

Table 2.4: Python's Delimiters

`(`	`)`	`[`	`]`	`{`	`}`			grouping
`.`	`,`	`:`	`;`	`@`				punctuation
`=`	`+=`	`-=`	`*=`	`/=`	`//=`	`%=`	`**=`	arithmetic assignment/binding
`&=`	`|=`	`^=`	`<<=`	`>>=`				bit–wise assignment/binding

EBNF Description: *delimiter* (Python Delimiters)

delimiter $\Leftarrow$ `(|)|`$\boxed{\texttt{[}}|\boxed{\texttt{]}}|\boxed{\texttt{\{}}|\boxed{\texttt{\}}}$`|.|,|:|;|@|=|+=|-=|*=|/=|//=|%=|**=|&=|`$\boxed{\texttt{|}}$`=|^=|<<=|>>=`

2.4.1 Python builds the longest legal token

Python constructs the longest legal token possible from the characters that it reads, but white–space often forces the end of a token. We can explore this rule using the lexical categories that we have learned. For example, Python tokenizes `import` as one identifier/keyword; it tokenizes `im␣port` as two identifiers: `im` and `port`: for clarity, the space appears here as the visible ␣ character.

When Python tokenizes code, it attempts to create the longest tokens possible

Python tokenizes `<<=` as one delimiter, not (1) as the operator `<<` followed by the delimiter `=`; nor (2) as the operator `<` followed by the operator `<=`; nor (3) as the operator `<` followed by the operator `<` followed by the delimiter `=`. But, we can always use white–space to force Python to create these three tokenizations by writing (1) `<<␣=`, (2) `<␣<=`, and (3) `<␣<␣=`. Similarly, if we write `<<~` Python recognizes the first token as the operator `<<` but since there are no operators or delimiters with all three characters, Python knows that this operator token is complete, and then Python finds that the next token is is the operator `~`.

We can use white–space to force Python to end a token, or let Python end the token naturally

We will write a token by enclosing its characters inside a box, followed by a superscript indicating the lexical category of the token: *i*dentifier (/*k*eyword), *o*perator, *d*elimiter, *l*iteral, and *c*omment. For `<<=` we write the delimiter token $\boxed{\texttt{<<=}}^d$; for `<<~` we write the two operator tokens $\boxed{\texttt{<<}}^o\boxed{\texttt{\~{}}}^o$. Likewise, for `<␣<␣=` we write the three operator tokens $\boxed{\texttt{<}}^o\boxed{\texttt{<}}^o\boxed{\texttt{=}}^o$. Finally, for the simple statement `x=y` we write the identifier, delimiter, and identifier tokens $\boxed{\texttt{x}}^i\boxed{\texttt{=}}^d\boxed{\texttt{y}}^i$.

We write tokens as characters in a box followed by a superscript indicating the category of the token: e.g., $\boxed{\texttt{sum}}^i$ for `sum` which is an identifier

Section Review Exercises

1. How would Python tokenize a. `<=` and b. `=<` ?

   Answer: a. `<=` is tokenized as the less–than–or–equal operator: $\boxed{\texttt{<=}}^o$. b. `=<` is tokenized as the equal delimiter followed by the less–than operator: $\boxed{\texttt{=}}^d\boxed{\texttt{<}}^o$. There is no equal–to–or-less–than operator in Python.

2. a. Tokenize the function definition `def␣gm(x,y):␣return␣math.sqrt(x*y)`
   b. Of the spaces, which can we omit and still produce the same tokens?

ANSWER: a. $\boxed{\texttt{def}}^{i/k}\boxed{\texttt{gm}}^{i}\boxed{\texttt{(}}^{d}\boxed{\texttt{x}}^{i}\boxed{\texttt{,}}^{d}\boxed{\texttt{y}}^{i}\boxed{\texttt{)}}^{d}\boxed{\texttt{:}}^{d}\boxed{\texttt{return}}^{i/k}\boxed{\texttt{math}}^{i}\boxed{\texttt{.}}^{d}\boxed{\texttt{sqrt}}^{i}\boxed{\texttt{(}}^{d}\boxed{\texttt{x}}^{i}\boxed{\texttt{*}}^{o}\boxed{\texttt{y}}^{i}\boxed{\texttt{)}}^{d}$
b. Omitting the second space produces the same tokens; omitting the first space, leads to $\boxed{\texttt{defgm}}^{i}$; omitting the last space, leads to $\boxed{\texttt{returnmath}}^{i}$.

2.5 Literals and their Types

Computers store and manipulate all information (numbers, text, audio, video) digitally: encoded as **b**inary dig**its** (bits) —sequences of zeroes and ones. The "type" of a value determines how Python interprets its binary information. The concept of types is critically import to our study of Python, and we will explore many different facets of this concept, starting here with literal values.

Computers store all information digitally, as bits; the type of the information determines how these bits are interpreted

In this section we learn how to read and write literals: each specifies a value belonging to one of Python's seven builtin types: numeric literals (types `int`, `float`, and `imaginary`), logical literals (type `bool`), text literals (types `str` and `bytes`), and one special literal of the type `NoneType`. Table 2.5 previews these types with examples of their literals.

We write values as literals in Python; each literal belongs to exactly one of Python's builtin types

Table 2.5: A Preview of Types and their Literals

Type	Used for	Example Literals
`int`	integral/integer values	`0   1024   0B111001011`
`float`	measurements/real value	`9.80665   9.10938188E-31`
`imaginary`	part of complex numbers	`5j   5.34j`
`bool`	logical/boolean	`True   False`
`str`	Unicode text/string	`""   '?'   'Mark said, "Boo!"\n'`
`bytes`	ASCII text/string	`b"pattis@ics.uci.edu"`
`NoneType`	no–value value	`None` (and no others)

We will learn how to import and use other types, and how to create ans use our own types, but only these seven builtin types have literal values.

We can import and create other types —but none have literals

2.5.1 Numeric Literals: `int`, `float`, and `imaginary`

Integer values (the `int` type in Python) represent quantities that are integral, countable, and discrete. We can write these literals using bases 2 (binary), 8 (octal), 10 (decimal) and 16 (hexadecimal) according to the following EBNF rules. In this book we will write every *int_literal* as a *decimal_literal*.

The type `int` represents quantities that are integral, countable, and discrete

EBNF Description: *int_literal* (Integer Literals: bases 2, 8, 10, and 16)

binary_digit	⇐ `0` | `1`
octal_digit	⇐ `0` | `1` | `2` | `3` | `4` | `5` | `6` | `7`
hex_digit	⇐ *digit* | `a` | `A` | `b` | `B` | `c` | `C` | `d` | `D` | `e` | `E` | `f` | `F`
non_0_digit	⇐ `1` | `2` | `3` | `4` | `5` | `6` | `7` | `8` | `9`
decimal_literal	⇐ `0`{`0`} | *non_0_digit*{*digit*}
octal_literal	⇐ `0b`*binary_digit*{*binary_digit*} | `0B`*binary_digit*{*binary_digit*}
octal_literal	⇐ `0O`*octal_digit*{*octal_digit*} | `0o`*octal_digit*{*octal_digit*}
hex_literal	⇐ `0X`*hex_digit*{*hex_digit*} | `0x`*hex_digit*{*hex_digit*}
int_literal	⇐ *decimal_literal* | *binary_literal* | *octal_literal* | *hex_literal*

Python places no restriction on the number of digits in an `int_literal`. There are no negative literals in Python: it interprets `-1` as $\boxed{\texttt{-}}^{o}\boxed{\texttt{1}}^{l/i}$ ($^{l/i}$ means a literal/`int`), which computes the value negative one; a distinction without a difference. Integers print in base 10 by default in Python.

Semantics of Integers

Floating point values (the `float` type in Python) represent quantities that are measurable, like the "real" numbers used in mathematics and the sciences. We write these literals according to the following EBNF rules.

Floating point quantities are like real numbers in mathematics and the sciences

EBNF Description: *float_literal* (Floating Point literals)

digits ⇐ *digit*{*digit*}
mantissa ⇐ *digits*`.`{*digit*} | `.`*digits*
exponent ⇐ `e`[`+`|`-`]`digits` | `E`[`+`|`-`]*digits*
float_literal ⇐ *mantissa*[*exponent*] | *digits exponent*

Observe from these rules that a *float_literal* is written in base 10; its `mantissa` can include digits before and/or after the decimal point and its *exponent* is option; an `exponent` starts with the letter `e` or `E` and has an optional sign followed one or more digits. Writing `E3` means $\times 10^3$. Finally, an *exponent* following *digits* without a decimal point is a *float_literal*: so, `1E3` is a *float_literal* (that we can write equivalently as `1000.`) but `1000` is an `int_literal`.

Floating point literals are written in base 10; `E3` means $\times 10^3$

Unlike integers, there are range bounds for floating point numbers. A mantissa contains about 15 digits and an exponent has a magnitude of about `300`.[1] Python prints floating point values without using E–notation when it makes sense, but switches to E–notation to print very large and small values.

Semantics of Floating points

Imaginary values represent the imaginary parts of complex numbers, and are written according to the following EBNF rule.

Complex numbers have real and imaginary parts

EBNF Description: *imaginary_literal* (Imaginary part of Complex numbers)

imag_ind ⇐ `j` | `J`
imaginary_literal ⇐ *digits imag_ind* | *float_literal imag_ind*

When we write a complex like `5.4+.2j` Python prints it as `(5.4+.2j)`.

Unlike mathematics (where integer is a subset of real which is a subset of complex) in Python integer, floating point, and complex literals are disjoint: every numeric literal matches the syntax of exactly one of these types of literals.

Types in Python are disjoint; a literal belongs to exactly one type

2.5.2 Logical/Boolean Literals: `bool`

Logical values (the `bool`[2] type in Python) represent truth values, which are use to represent true/false, yes/no, on/off, present/absent, etc. There are two literal values of the `bool` type, defined according to the following EBNF rule.

`True` and `False` are keywords and literals; we primarily classify them as a literals.

EBNF Description: *bool_literal* (Logical/Boolean literals)

bool_literal ⇐ `True` | `False` (both are Python keywords)

[1] The biggest floating point value in Python is `1.7976931348623157e+308` and the smallest (closest to zero) is `2.2250738585072014e-308`. The 64 bits allocated to floating point values as follows: 53 bits for the mantissa and its sign, and 11 bits for the exponent and its sign.

[2] This name honors George Boole, a 19*th*–century English mathematician who revolutionized the study of logic by making it more like arithmetic. He invented a method for calculating with truth values and an algebra for reasoning about these calculations. Boole's method are used extensively in hardware and software systems. Boole rhymes with tool.

2.5.3 Text/String Literals: `str` and `bytes`

Text/String values (primarily the `str` type in Python) represent sequences of characters.[3] Input/Output —via the keyboard, console screen, and most files— processes strings. The `str` type in Python can store Unicode characters; the `bytes` type in Python (byte strings) stores only ASCII characters.[4] Beginning programmers use literals of the `str` type much more than the `bytes` type; the EBNF rules for both types of literals appear below.

Text/Strings are sequences of characters strung together, used primarily for input/output

EBNF Description: *str_literal* (String and Byte String literals)

```
text_chars          ⇐ graphic | white_space | " | '                                    (all but the \ character)
esc_a               ⇐ \text_chars | \o octal_digit^3 | \h hex_digit^2                  (3 octal or 2 hex digits)
esc_u               ⇐ esc_a | \n{unicode_name} | \u hex_digit^4 | \U hex_digit^8       (4 or 8 hex digits)
raw_opt             ⇐ r | R
bytes_ind           ⇐ b | B

single_quoted_str   ⇐ "{graphic | esc_u | ␣ | → | '}"   |   '{graphic | esc_u | ␣ | → | "}'
triple_quoted_str   ⇐ """{text_chars | esc_u}"""   |   '''{text_chars | esc_u"}'''
str_literal         ⇐ [raw_opt]single_quoted_str | [raw_opt]triple_quoted_str

single_quoted_bytes ⇐ "{graphic | esc_a | ␣ | → | '}"   |   '{graphic | esc_a | ␣ | → | "}'
triple_quoted_bytes ⇐ """{text_chars | esc_u}"""   |   '''{text_chars | esc_u}'''
bytes_literal       ⇐ bytes_opt[raw_opt]single_quoted_bytes | bytes_opt[raw_opt]triple_quoted_bytes
```

There are two kinds of string/byte string literals in Python: single– and triple–quoted; each must start and end with the same kind of quotation mark: a double quote `"` or a single quote `'`. Single–quoted strings/byte strings start and end on the same line, and can contain all characters but ↩ and `\`. Triple–quoted strings/byte strings can span multiple lines, and can contain all characters but `\`. Note that inside strings/byte strings white–space does not separate tokens, it becomes part of the string. Finally, empty strings specify a repetition of zero times, so the quotation marks can be adjacent: e.g., `""`.

Single–quoted string literals must appear on one line; triple–quoted string literals can span multiple lines

What are **esc**ape characters? What is the difference between *str* and *bytes* literals? Why do strings and byte strings allow different escape characters? How does the optional `raw_opt`, if included, affect these literals? We will discuss all these topics in later chapters, when we examine string and byte string processing in detail. In this chapter, we are interested only in how string and byte string literals are written, so we can recognize them in Python programs.

Escape characters and byte strings are discussed in later chapters

Finally, there are also special places in Python code where strings can appear, and be read and processed by other programs. PyDoc is a Python utility that reads such strings to produce special web–pages that document the code; DocTest is a testing utility that reads such strings and uses them as test cases for checking that the code behaves correctly.

Strings appearing in special parts of Python programs can be read and processed by Python utility programs

2.5.4 A Literal About Nothing: `NoneType`

Python has a special type that has just one literal value. The type is `NoneType` and its single literal is `None`, defined according to the following EBNF rule.

`None` is a keyword and literal; we primarily classify it as a literal.

[3]Strings are characters "strung" together, one after another, preserving their order.

[4]A byte stores 8 bits of information, allowing 256 different ASCII characters.

EBNF Description: *none_literal* (`NoneType` literal: meaning "no value")

none_literal $\Leftarrow$ `None` (a Python keyword)

Why does Python include the `NoneType` and its `None` literal value? As one example, all functions in Python must return a value; but some functions are commands (done for effect) not queries (done to compute a value), so they don't have any value to return: these functions, because they must return a value, return `None`. We will learn a few different uses for `None` in Python.

We use `None` whenever a Python language feature requires a value, but there is no sensible value to use

SECTION REVIEW EXERCISES

1. Classify each of the following as a legal or illegal literal; if it is legal, indicate it type; if it is a *float_literal* with an exponent, write it as an equivalent *float_literal* without an exponent. If it is illegal, propose a legal literal that has a similar value and type.

a. `22`	j. `E2`	s. `""""""`
b. `00`	k. `5E2`	t. `"True"`
c. `007`	l. `9.193818e+400`	u. `"SSN'`
d. `1,024`	m. `0b01`	v. `'caveat emptor'`
e. `0o128`	n. `0b101E1`	w. `br"Hello world\n"`
f. `5.0`	o. `002.99792758E09`	x. `"""`
g. `5.`	p. `007j`	`gm(x,y) -> float`
h. `5`	q. `true`	`Compute the geometric mean`
i `000.5`	r. `yes`	`"""`

ANSWER:

a. Legal: `int`	j. Illegal: `1E2`	s. Legal: `str`
b. Legal: `int`	k. Legal: `float 500.`	t. Legal: `str`
c. Illegal: `7`	l. Illegal: `float` too big	u. Illegal: `'SSN'` or `"SSN"`
d. Illegal: `1024`	m. Legal: `int`	v. Legal: `str`
e. Illegal: `0o127`	n. Illegal: `0b110010`	w. Legal: `bytes`
f. Legal	o. Legal: `float 2997927580.`	x. Legal: `str`
g. Legal	p. Legal: `imaginary`	
h. Legal: `int`	q. Illegal: `True`	
i. Legal: `float`	r. Illegal `True`	

2. b. Tokenize the code `tails␣=␣sum(throw_coin(100,.55))` writing the superscripts $^{i,i/k,o,d,l/x}$, where x in l/x can be *i*nteger, *f*loat, *j* imaginary, *b*oolean, *s*tring, *bs* byte string, or *n*one, indicating the type of the literal.

ANSWER: a. $\boxed{\texttt{tails}}^{i}\boxed{\texttt{=}}^{d}\boxed{\texttt{sum}}^{i}\boxed{\texttt{(}}^{d}\boxed{\texttt{throw\_coin}}^{i}\boxed{\texttt{(}}^{d}\boxed{\texttt{100}}^{l/i}\boxed{\texttt{,}}^{d}\boxed{\texttt{.55}}^{l/f}\boxed{\texttt{)}}^{d}\boxed{\texttt{)}}^{d}$

2.6 Comments

Programmers embed comments in their Python programs as documentation, according to the following syntax. Comments start with the special `#` character and end with a "new line" $\hookleftarrow$ character, with any characters between; inside comments white–space does not separate tokens, it is part of the comment.

Programmers document their code with comments

EBNF Description: *comment* (Python Comments)

comment $\Leftarrow$ `#`$\{$*text_chars* $|$ `\`$\}$ $\hookleftarrow$

Semantically, once Python tokenizes a comment it discards that token, excluding it from from further processing. So, a program has the same meaning in

Comment Semantics

Python with or without its comments.

Comments embody information that cannot be expressed in code

But good documentation is an important part of programming. Comments help programmers capture aspects of their code that they cannot express directly in the Python language: e.g., goals, specifications, design decisions, time/space trade-offs, historical information, advice for using/modifying their code. Comments are anything that can be typed on the keyboard: English, mathematics, even low-resolution pictures. Programmers intensely study their own code (or the code of others) when writing and maintaining programs. Good commenting make these tasks much easier.

2.7 Tokenizing Python Programs

How do Line–joining and indentation affect the lexical analysis of Python code

This section completes our analysis of the lexical structure of Python. We first discuss the concepts of physical and logical lines (and line–joining), and then examine how Python actively tokenizes indentation and its opposite: dedentation. Finally, we will see how Python uses all the information covered in this chapter to analyze the lexical structure of a program by fully tokenizing it.

2.7.1 Physical and Logical Lines: Line–Joining with \

The backslash \ character forces Python to join physical lines into longer logical lines

The backslash character (\) is also known as the "line–joining" character: it joins physical lines to create longer logical lines. A physical line in Python always ends in a ↩. But if the line–joining character appears as the last one on a physical line (i.e., appears right before the ↩) Python creates a logical line that includes the current physical line (with its \↩ replaced by ␣) followed by the characters on the next line. This process can repeat, if subsequent physical lines also end in \↩. So, it is possible to create a very long logical line from any number of physical lines; that final logical line ends with a ↩.

An example and explanation of line–joining

In the following code (with the ↩ character explicitly shown) Python translates the lines on the left...

```
code1↩
code2\↩
code3\↩
code4↩
code5↩
```

...into the equivalent lines on the right. Notice how each \↩ pair at the end of a line is replaced by ␣ and then that line is joined with the next line.

```
code1↩
code2␣code3␣code4↩
code5↩
```

There are two places where \ does not mean the line-joining character

The backslash character can appear in three contexts: in a string/byte string (starting an escape character), in a comment as itself, or at the end of a line (as the line–joining character). If it appears anywhere else, Python reports a lexical error.

Line–joining is useful, because style rules limit the length of lines in programs

How is line–joining used in practice? Style rules specify the maximum length of a line of code; the number should be chosen so the programmer's editor can show every character in a line of code, without horizontal scrolling. A common choice is 80 characters. So, we break a longer line into shorter ones, using line–joining to satisfy this style rule. Most code in this book will not require line–joining, but it is convenient to present this material here, for reference.

2.7.2 Indentation: The $\boxed{\texttt{INDENT}}$ and $\boxed{\texttt{DEDENT}}$ Tokens

Python uses indentation (the amount of white–space at the beginning of a line) to determine statement nesting

Indentation is the amount of white–space that occurs at the front of each line in a program; if a line is all white-space, Python ignores it. Python is one of just a few programming languages that uses indentation to specify how statements nest (accomplished with "blocks" in most other languages). We will examine the meaning and use of nested statements when we cover statements themselves; for now we explore only how Python tokenizes indentation in programs.

Python produces special tokens for indentation and dedentation

Python tokenizes indentation using the following algorithm, which produces special $\boxed{\texttt{INDENT}}$ and $\boxed{\texttt{DEDENT}}$ tokens, using a list that grows and shrinks while storing relevant prior indentations. We present this algorithm here for reference. Don't memorize it, but understand and be able to follow/apply it.

Python's indentation algorithm

```
I. Ensure that the first line has no indentation (0 white-space characters); if
   it doesn't, report an error. If it does, initialize the list with the value 0.
II. For each logical line (after line–joining)
    A. If the current line's indentation is > the indentation at the list's end
       1. Add the current line's indentation to the end of the list.
       2. Produce an INDENT token.
    B. If the current line's indentation is < the indentation at the list's end
       1. For each value at the end of the list that is unequal to the current
          line's indentation (if it is not in the list, report a lexical error).
          a. Remove the value from the end of the list.
          b. Produce a DEDENT token.
    C. Tokenize the current line.
III. For every indentation on the list except 0, produce a DEDENT token.
```

A simple example of Python's indentation algorithm

We apply this algorithm below to compute all the tokens (including the indentation tokens) for the following simplified code: each line is numbered, shows its white–space, and contains just one identifier token; the actual tokens on a line are irrelevant to the indentation algorithm. The indentation pattern for these lines is 0, 2, 2, 4, 6, 6, 2, 4, and 2.

```
1 a
2 ␣␣b
3 ␣␣c
4 ␣␣␣␣d
5 ␣␣␣␣␣␣e
6 ␣␣␣␣␣␣f
7 ␣␣g
8 ␣␣␣␣h
9 ␣␣i
```

$\boxed{\texttt{a}}^i$ $\boxed{\texttt{INDENT}}$ $\boxed{\texttt{b}}^i$ $\boxed{\texttt{c}}^i$ $\boxed{\texttt{INDENT}}$ $\boxed{\texttt{d}}^i$ $\boxed{\texttt{INDENT}}$ $\boxed{\texttt{e}}^i$ $\boxed{\texttt{f}}^i$ $\boxed{\texttt{DEDENT}}$ $\boxed{\texttt{DEDENT}}$ $\boxed{\texttt{g}}^i$
$\boxed{\texttt{INDENT}}$ $\boxed{\texttt{H}}^i$ $\boxed{\texttt{DEDENT}}$ $\boxed{\texttt{i}}^i$ $\boxed{\texttt{DEDENT}}$

A trace of the indentation algorithm

Table 2.6 is a trace of the indentation algorithm running on the simplified code above. It explains how the algorithm produces the following tokens.

Table 2.6: Tokenizing with the Indentation Algorithm

Line	List	CLI	Algorithm Step(s) Followed (CLI is Current Line Indentation)
1	None	0	I. The first line has no indentation (0 spaces); initialize the List with 0
1	0	0	IIC. produce $\boxed{\text{a}}^i$ token
2	0	2	IIA1. add 2 to List; IIA2. produce the $\boxed{\text{INDENT}}$ token; IIC. produce $\boxed{\text{b}}^i$ token
3	0, 2	2	IIC. produce $\boxed{\text{c}}^i$ token
4	0, 2	4	IIA1. add 4 to List; IIA2. produce the $\boxed{\text{INDENT}}$ token; IIC. produce $\boxed{\text{d}}^i$ token
5	0, 2, 4	6	IIA1. add 2 to List; IIA2. produce the $\boxed{\text{INDENT}}$ token; IIC. produce $\boxed{\text{e}}^i$ token
6	0, 2, 4, 6	6	IIC. produce $\boxed{\text{f}}^i$ token
7	0, 2, 4, 6	2	IIB1. remove 6 from List; IIB2. produce the $\boxed{\text{DEDENT}}$ token; IIB1. remove 4 from List; IIB2. produce the $\boxed{\text{DEDENT}}$ token; IIC. produce $\boxed{\text{g}}^i$ token; stop because indentation is 2
8	0, 2	4	IIA1. add 4 to List; IIA2. produce the $\boxed{\text{INDENT}}$ token; IIC. produce $\boxed{\text{h}}^i$ token
9	0, 2, 4	2	IIB1. remove 4 from List; IIB2. produce the $\boxed{\text{DEDENT}}$ token
End	0, 2		III. remove 2 from List; produce the $\boxed{\text{DEDENT}}$ token; stop when indentation is 0

2.7.3 Complete Tokenization of a Python Program

Python's complete tokenization algorithm, including indentation

We have now learned all the rules that Python uses to tokenize programs, which we will summarize in the following algorithm. Again, we present this algorithm here for reference. Don't memorize it, but understand and be able to follow/apply it.

I. Scan the code from left–to–right and top–to–bottom using line-joining and using white–space to separate tokens (where appropriate).

   A. At the start of a line, tokenize the indentation; beyond the first token, skip any white–space (continuing to a new line after a $\hookleftarrow$)

   B. If the first character is a *lower/upper* letter or underscore, tokenize the appropriate identifier, keyword, or literal: `bool`, `str`, `bytes`, or `NoneType`; recall `str` and `bytes` literals can start with combinations of `r`*s* and `b`*s*.

   C. If the first character is an *ordinary* character, tokenize an operator, delimiter, or comment...

      1. ...except if the character is period immediately followed by a `digit`, in which case tokenize a `float` literal.

   D. If the first character is a *digit*, tokenize the appropriate numeric literal.

   E. If the first character is either kind of quote, tokenize a `str` literal.

A complex example of Python's indentation algorithm

Below is a complex example of a Python function and the tokens Python produces for it, including the standard superscripts: *i*dentifier (with /*k* for keyword), *o*perator, *d*elimiter, *l*iteral (with /*x* identifying type *x*), and *c*omment label these tokens. The lines these tokens appear on have no significance.

```
def␣collatz(n):
␣␣"""
␣␣Computes␣the␣number␣of␣iterations␣needed␣to␣reduce␣n␣to␣1
␣␣"""
␣␣count␣=␣0
␣␣while␣True:
␣␣␣␣if␣n␣==␣1:
␣␣␣␣␣␣break
␣␣␣␣count␣+=␣1
␣␣␣␣if␣n%2␣==␣0:␣#check␣n
␣␣␣␣␣␣n␣=␣n//2␣␣␣#n␣was␣even
␣␣␣␣else:
␣␣␣␣␣␣n␣=␣3*n+1␣␣#␣was␣odd
␣␣return␣count
```

$\boxed{\texttt{def}}^{i/k}$ $\boxed{\texttt{collatz}}^{i}$ $\boxed{\texttt{(}}^{d}$ $\boxed{\texttt{n}}^{i}$ $\boxed{\texttt{)}}^{d}$ $\boxed{\texttt{:}}^{d}$ $\boxed{\texttt{INDENT}}$
$\boxed{\texttt{"""}\hookleftarrow\texttt{␣␣Computes the number of iterations needed to reduce n to 1}\hookleftarrow\texttt{␣␣"""}}^{l//s}$ $\boxed{\texttt{count}}^{i}$ $\boxed{\texttt{=}}^{d}$
$\boxed{\texttt{0}}^{l/i}$ $\boxed{\texttt{while}}^{i/k}$ $\boxed{\texttt{True}}^{l/b}$ $\boxed{\texttt{:}}^{d}$ $\boxed{\texttt{INDENT}}$ $\boxed{\texttt{if}}^{i/k}$ $\boxed{\texttt{n}}^{i}$ $\boxed{\texttt{==}}^{o}$ $\boxed{\texttt{1}}^{l/i}$ $\boxed{\texttt{:}}^{d}$ $\boxed{\texttt{INDENT}}$ $\boxed{\texttt{break}}^{i/k}$ $\boxed{\texttt{DEDENT}}$
$\boxed{\texttt{count}}^{i}$ $\boxed{\texttt{+=}}^{d}$ $\boxed{\texttt{1}}^{l/i}$ $\boxed{\texttt{if}}^{i/k}$ $\boxed{\texttt{n}}^{i}$ $\boxed{\texttt{\%}}^{o}$ $\boxed{\texttt{2}}^{l/i}$ $\boxed{\texttt{==}}^{o}$ $\boxed{\texttt{0}}^{l/i}$ $\boxed{\texttt{:}}^{d}$ $\boxed{\texttt{\#check n}}^{c}$ $\boxed{\texttt{INDENT}}$ $\boxed{\texttt{n}}^{i}$ $\boxed{\texttt{=}}^{d}$ $\boxed{\texttt{n}}^{i}$ $\boxed{\texttt{//}}^{o}$
$\boxed{\texttt{2}}^{l/i}$ $\boxed{\texttt{\#n was even:}}^{c}$ $\boxed{\texttt{DEDENT}}$ $\boxed{\texttt{else}}^{i/k}$ $\boxed{\texttt{:}}^{d}$ $\boxed{\texttt{INDENT}}$ $\boxed{\texttt{n}}^{i}$ $\boxed{\texttt{=}}^{d}$ $\boxed{\texttt{3}}^{l/i}$ $\boxed{\texttt{*}}^{o}$ $\boxed{\texttt{n}}^{i}$ $\boxed{\texttt{+}}^{o}$ $\boxed{\texttt{1}}^{l/i}$ $\boxed{\texttt{\#n was odd}}^{c}$
$\boxed{\texttt{DEDENT}}$ $\boxed{\texttt{DEDENT}}$ $\boxed{\texttt{return}}^{i/k}$ $\boxed{\texttt{count}}^{i}$ $\boxed{\texttt{DEDENT}}$

Carefully examine the triple-quoted string; note the spaces (␣) and newlines ($\hookleftarrow$) that are allowed inside it, all according to the *triple_quoted_str* EBNF rule.

Examine the contents of the triple–quoted string literal

Section Review Exercises

1. Identify the errors Python reports when it analyzes the lexical structure of the left and right code below.

```
1  def␣f(x):
2  ␣␣return␣\#use␣line-joining
3  ␣␣x
```

```
1  def␣sum(x):
2  ␣␣s␣=␣0
3  ␣␣for␣i␣in␣range(x):
4  ␣␣␣␣␣␣s␣+=␣i
5  ␣␣␣␣return␣s
```

Answer: **Left**: Python reports a lexical error on line 2 because when a line–joining character is present, it must appear last on the line, right before $\hookleftarrow$. If Python joined these lines as described, the identifier `x` would become part of the comment at the end of line 2! **Right**: Python reports a lexical error on line 5 because the indentation pattern (0, 2, 2, 6, 4) is illegal. When Python processes line 5, the list contains 0, 2, 6; the 4–space indentation on line 5 does not match anything in this list. So, rule IIB1 in the algorithm causes Python to report of an error.

2.8 Seeing Programs

Adriaan de Groot, a Dutch psychologist, did research on chess expertise in the 1940s, by performing the following experiment: he sat down chess experts in front of an empty chessboard, all the pieces from a chess set, and a curtain. Behind the curtain was a chessboard with its pieces arranged about 35 moves into a game. The curtain was briefly raised and then lowered. The experts were

What do chess experts see when viewing a mid–game chess board?

then asked to reconstruct the game board. In most cases, the experts were able to completely reconstruct the game board they saw. He then performed the same experiment with chess novices; they did much worse. These results could be interpreted as, "Chess experts have much better memories than novices."

American psychologists Herb Simon and Bill Chase repeated DeGroot's experiment and extended it in the 1960s. In a second experiment, the board behind the curtain had the same number of chess pieces, but they were randomly placed on the board; they did not represent an ongoing game. In this experiment, when asked to reconstruct the chess board, the experts did only marginally better than the novices. Their conclusion: "Experts saw the board differently than novices: they performed much better in the first experiment because they were recognizing the structure of the game, not just seeing pieces on a board."

What do chess experts see when viewing a chess board with pieces placed randomly on it?

This chapter is trying to teach us how to see programs as a programmer (and Python) sees them: not just as a sequence of characters, but recognizing the tokens using Python's lexical structures; in later chapters we will learn to see programs at even a higher level, in terms of their syntactic structures.

See programs as tokens instead of as seeing them characters

Chapter Summary

This chapter examined the lexical structure of Python programs. It started with a discussion of Python's character set, and then examined Python's five lexical categories in detail, describing each using EBNF rules. Some categories used very simple EBNF descriptions, while others described more complex structures for their tokens. We learned how to box characters to denote tokens, labeling each token with a superscript describing the token's lexical category. In the process, we learned how Python uses the "longest legal token" rule when tokenizing characters. The biggest lexical category we examined was literals, where we began our study of the concept of types in programming languages. We then learned how Python joins physical lines into logical lines, and how Python tokenizes indentation: white–space at the beginning of a line. Then, we distilled all this knowledge into two algorithms that describe how Python tokenizes a program, including line-joining and the tokenization of indentation. A last example showed how Python translates a sequence of characters into a sequence of tokens (including INDENT and DEDENT tokens). Finally, we discussed how knowing how to tokenize programs allowed us to see them more as a programmer (or Python) sees them.

Chapter Exercises

1. White–space frequently separates tokens in Python. When can white–space be included as part of a token? Discuss ␣, →, and ↩ separately.

2. Tokenize each of the following, including the appropriate superscript(s).
   a. `six` b. `vi` c. `6` d. `6.` e. `6E0` f. `"six"` g. `b"six"` h. `b␣"six"`

3. a. List five keywords that are also operators. b. List three that are also literals. c. How do we categorize these tokens?

4. Tokenize each of the following (caution: how many tokens are in each?)

a.
```
#Here print("\n\n") prints two blank lines
```
b.
```
"""
␣␣I put the comment #KLUDGE on a line
␣␣to mark the code for further review
"""
```

5. Answer and justify: a. How many 1–character identifiers are there in Python? b. 2–character identifiers? c. n–character identifiers? (write a formula) d. 5–character identifiers? (use the formula)

6. a. Is is possible to have two consecutive `INDENT` tokens? b. Is is possible to have two consecutive `DEDENT` tokens? If so, cite examples from this book.

7. Tokenize the code below. What is its indentation pattern? Is it legal?

```
def␣happy(times,lines):
␣␣for␣l␣in␣range(lines):
␣␣␣␣␣␣␣print("I'm␣Happy",end='')
␣␣␣␣␣␣␣for␣t␣in␣range(times-1):
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣print(',␣Happy',end='')
␣␣␣␣␣␣␣print('␣to␣know␣you.')
␣␣print('And␣I␣hope␣you␣are␣happy␣to␣know␣me␣too.')
```

8. This chapter discussed two lexical errors that Python reports. When do they occur?

9. Select an appropriate type to represent each of the following pieces of information: a. the number of characters in a file; b. elapsed time of day since midnight (accurate to 1 second); c. whether or not the left mouse button is currently pushed; d. the temperature of a blast furnace; e. an indication of whether one quantity is less than, equal to, or greater than another; d. the position of a rotary switch (with 5 positions); h. the name of a company.