Data Types

- The data types considered are all built-in, except for one which comes from the standard library.
- The only difference between built- in data types and library data types is that in the latter case, we must first im- port the relevant module and we must qualify the data type's name with the name of the module it comes from.

Identifiers and Keywords

- when we assign in Python, what really happens is that we bind an object reference to refer to the object in memory that holds the data.
- The names we give to our object references are called identifiers or just plain names.
- A valid Python identifier is a nonempty sequence of characters of any length that consists of a "start character" and zero or more "continuation characters".
- The start char- acter can be anything that Unicode considers to be a letter, including the ASCII letters ("a", "b", ⋯, "z", "A", "B", ⋯, "Z"), the underscore ("_"), as well as the let- ters from most non-English languages.
- Each continuation character can be any character that is permitted as a start character, or pretty well any non- whitespace character, including any character that Unicode considers to be a digit, such as ("0", "1", ⋯, "9"), or the Catalan character "·". Identifiers are case-sensitive.
- The precise set of characters that are permitted for the start and continuation are described in the documentation (Python language reference, Lexical analysis, Identifiers and keywords section), and in PEP 3131 (Supporting Non-ASCII Identifiers).★

- The second rule is that no identifier can have the same name as one of Python's keywords

Table 2.1 Python's Keywords

Column1	Column2	Column3	Column4	Column5	Column6	Column7
and	continue	except	global	lambda	pass	while
as	def	False	if	None	raise	with
assert	del	finally	import	nonlocal	return	yield
break	elif	for	in	not	True	
class	else	from	is	or	try	

- The first convention is: Don't use the names of any of Python's predefined iden- tifiers for your own identifiers.
- So, avoid using NotImplemented and Ellipsis, and the name of any of Python's built-in data types (such as int, float, list, str, and tuple), and any of Python's built-in functions or exceptions.

- How can we tell whether an identifier falls into one of these categories? Python has a built-in function called dir() that returns a list of an object's attributes.

```
>>> dir() # Python 3.1's list has an extra item, '__package__'
['__builtins__', '__doc__', '__name__']
```

- The **builtins** attribute is, in effect, a module that holds all of Python's built-in attributes.

```
>>> dir(__builtins__)
['ArithmeticError', 'AssertionError', 'AttributeError',
...
'sum', 'super', 'tuple', 'type', 'vars', 'zip']
```

> ★ A "PEP" is a Python Enhancement Proposal.
> ★ All the PEPs are accessible from www.python.org/dev/peps/.

- There are about 130 names in the list.
- Those that begin with a capital letter are the names of Python's built-in exceptions; the rest are function and data type names.
- The second convention concerns the use of underscores (_).
- Names that begin and end with two underscores (such as **It**) should not be used.
- Python defines various special methods and variables that use such names (and in the case of special methods, we can reimplement them, that is, make our own ver- sions of them), but we should not introduce new names of this kind ourselves.
- Names that begin with one or two lead- ing underscores (and that don't end with two underscores) are

treated specially in some contexts.

- A single underscore on its own can be used as an identifier, and inside an interactive interpreter or Python Shell, _ holds the result of the last expression that was evaluated.
- In a normal running program no _ exists, unless we use it explicitly in our code.
- Some programmers like to use _ in for ⋯ in loops when they don't care about the items being looped over.

```
for _ in (0, 1, 2, 3, 4, 5):
    print("Hello")
```

- Be aware, however, that those who write programs that are internationalized often use _ as the name of their translation function.
- They do this so that instead of writing gettext.gettext("Translate me"), they can write _("Translate me")

```
π = math.pi
ε = 0.0000001
nueva_área = π * radio * radio
if abs(nueva_área - vieja_área) < ε:
    print("las áreas han convergido")
```

- We are free to use accented characters and Greek letters for identifiers.
- We could just as easily create identifiers using Arabic, Chinese, Hebrew, Japanese, and Russian characters, or indeed characters from any other language supported by the Unicode character set.
- The easiest way to check whether something is a valid identifier is to try to assign to it in an interactive Python interpreter or in IDLE's Python Shell window.

```
>>> stretch-factor = 1
SyntaxError: can't assign to operator (...)
>>> 2miles = 2
SyntaxError: invalid syntax (...)
>>> str = 3 # Legal but BAD
>>> l'impôt31 = 4
SyntaxError: EOL while scanning single-quoted string (...)
>>> l_impôt31 = 5
>>>
```

- When an invalid identifier is used it causes a SyntaxError exception to be raised.
- The first assignment fails because "-" is not a Unicode letter, digit, or underscore.
- The second one fails because the start character is not a Unicode letter or underscore; only continuation characters can be digits.
- No exception is raised if we create an identifier that is valid—even if the identifier is the name of a built-

in data type, exception, or function—so the third assignment works, although it is ill-advised.
- The fourth fails because a quote is not a Unicode letter, digit, or underscore.
- The fifth is fine.

Integral Types

- Python provides two built-in integral types, int and bool.★ Both integers and Booleans are immutable.
- When used in Boolean expressions, 0 and False are False, and any other integer and True are True.
- When used in numerical expressions True evaluates to 1 and False to 0.
- This means that we can write some rather odd things—for example, we can increment an integer, i, using the expression i += True.
- Naturally, the correct way to do this is i += 1.

Integers

- The size of an integer is limited only by the machine's memory, so integers hundreds of digits long can easily be created and worked with—although they will be slower to use than integers that can be represented natively by the machine's processor.

> ★ The standard library also provides the fractions.Fraction type (unlimited precision rationals) which may be useful in some specialized mathematical and scientific contexts.

Table2.2 NumericOperatorsandFunctions

Syntax	Description
x + y	Adds number x and number y
x - y	Subtracts y from x
x * y	Multiplies x by y
x / y	Divides x by y; always produces a float (or a complex if x or y is complex)
x // y	Divides x by y; truncates any fractional part so always pro- duces an int result; see also the round() function
x % y	Produces the modulus (remainder) of dividing x by y
x ** y	Raises x to the power of y; see also the pow() functions
-x	Negates x; changes x's sign if nonzero, does nothing if zero
+x	Does nothing; is sometimes used to clarify code

abs(x)	Returns the absolute value of x
divmod(x, y)	Returns the quotient and remainder of dividing x by y as a tuple of two ints
pow(x, y)	Raises x to the power of y; the same as the ** operator
pow(x, y, z)	A faster alternative to (x ** y) % z
round(x, n)	Returns x rounded to n integral digits if n is a negative int or returns x rounded to n decimal places if n is a positive int; the returned value has the same type as x; see the text

Table2.3 IntegerConversionFunctions

Syntax	Description
bin(i)	Returns the binary representation of int i as a string, e.g., bin(1980) == '0b11110111100'
hex(i)	Returns the hexadecimal representation of i as a string, e.g., hex(1980) == '0x7bc'
int(x)	Converts object x to an integer; raises ValueError on failure—or TypeError if x's data type does not support integer conversion. If x is a floating-point number it is truncated.
int(s, base)	Converts str s to an integer; raises ValueError on failure. If the optional base argument is given it should be an integer between 2 and 36 inclusive.
oct(i)	Returns the octal representation of i as a string, e.g., oct(1980) == '0o3674'

- Integer literals are written using base 10 (decimal) by default, but other number bases can be used when this is convenient:

```
>>> 14600926     # decimal
14600926
>>> 0b110111101100101011011110         # binary
14600926
>>> 0o67545336             # octal
14600926
>>> 0xDECADE             # hexadecimal
14600926
```

- Binary numbers are written with a leading 0b, octal numbers with a leading 0o,★ and hexadecimal numbers with a leading 0x.

- Uppercase letters can also be used.
- Some of the functionality is provided by built-in functions like abs()—for example, abs(i) returns the

absolute value of integer i—and other functionality is provided by int operators—for example, i + j returns the sum of integers i and j.

- For floats, the round() function works in the expected way—for example, round(1.246, 2) produces 1.25—for ints, using a positive rounding value has no effect and results in the same number being returned, since there are no decimal digits to workon.
- Butwhen a negative rounding value is used a subtle and useful behavior is achieved—for example, round(13579, -3) produces 14000, and round(34.8, -1) produces 30.0.
- All the binary numeric operators (+, -, /, //, %, and **) have augmented assign- ment versions (+=, -=, /=, //=, %=, and **=) where x op= y is logically equivalent to x = x op y in the normal case when reading x's value has no side effects.
- Objects can be created by assigning literals to variables, for example, x = 17, or by calling the relevant data type as a function, for example, x = int(17).
- Some objects (e.g., those of type decimal.Decimal) can be created only by using the data type since they have no literal representation.
- When an object is created using its data type there are three possible use cases.
- The first use case is when a data type is called with no arguments. In this case an object with a default value is created—for example, x = int() creates an integer of value 0.
- All the built-in types can be called with no arguments.
- The second use case is when the data type is called with a single argument.
- If an argument of the same type is given, a new object which is a shallow copy of the original object is created.

> ★Users of C-style languages note that a single leading 0 is not sufficient to specify an octal number; 0o (zero, letter o) must be used in Python.

- If an argument of a different type is given, a conversion is attempted.
- If the argument is of a type that supports conversions to the given type and the conversion fails, a ValueError exception is raised; otherwise, the resultant object of the given type is returned.
- If the argument's data type does not support conversion to the given type a TypeError exception is raised.
- The built-in float and str types both provide integer conversions; it is also possible to provide integer and other conversions for our own custom data types.
- The third use case is where two or more arguments are given—not all types support this, and for those that do the argument types and their meanings vary.
- For the int type two arguments are permitted where the first is a string that represents an integer and the second is the number base of the string representation.
- All the binary bitwise operators (|, ^, &, <<, and >>) have augmented assignment versions (|=, ^=, &=, <<=, and >>=) where i op= j is logically equivalent to i = i op j in the normal case when reading i's value has no side effects.
- From Python 3.1, the int.bit_length() method is available.
- This returns the number of bits required to represent the int it is called on.
- (2145).bit_length() returns 12. (The parentheses are required if a literal inte- ger is used, but not if we use an integer variable.)

Table2.4 IntegerBitwiseOperators

Syntax	Description
i OR j	Bitwise OR of int i and int j; negative numbers are assumed to be represented using 2's complement
i ^ j	Bitwise XOR (exclusive or) of i and j
i & j	BitwiseAND of i andj
i << j	Shifts i left by j bits; like i *(2 * j)* without overflow checking
i >> j	Shifts i right by j bits; like i // (2 ** j) without overflow checking
~i	Inverts i's bits

Booleans

- There are two built-in Boolean objects: True and False.
- The bool data type can be called as a function—with no arguments it returns False, with a bool argument it returns a copy of the argument, and with any other argument it attempts to convert the given object to a bool.
- All the built-in and standard library data types can be converted to produce a Boolean value, and it is easy to provide Boolean conversions for custom data types.

```
>>> t = True
>>> f = False
>>> t and f
False
>>> t and True
True
```

- Python provides three logical operators: and, or, and not.
- Both and and or use short-circuit logic and return the operand that determined the result, whereas not always returns either True or False.
- Programmers who have been using older versions of Python sometimes use 1 and 0 instead of True and False; this almost always works fine, but new code should use the built-in Boolean objects when a Boolean value is required.

Floating-Point Types

- Python provides three kinds of floating-point values: the built-in float and complex types, and the decimal.Decimal type from the standard library.

- All three are immutable.
- Type float holds double-precision floating-point numbers whose range depends on the C (or C# or Java) compiler Python was built with; they have limited precision and cannot reliably be compared for equality.
- Numbers of type float are written with a decimal point, or using exponential notation, for example, 0.0, 4., 5.7, -2.5, -2e9, 8.9e-4.

- Computers natively represent floating-point numbers using base 2.

```
>>> 0.0, 5.4, -2.5, 8.9e-4
(0.0, 5.4000000000000004, -2.5, 0.00089999999999999995)
```

- Python 3.1 produces much more sensible-looking output:

```
>>> 0.0, 5.4, -2.5, 8.9e-4
(0.0, 5.4, -2.5, 0.00089)
```

- When Python 3.1 outputs a floating-point number, in most cases it uses David Gay's algorithm.

- This outputs the fewest possible digits without losing any accuracy.
- Although this produces nicer output, it doesn't change the fact that computers (no matter what computer language is used) effectively store floating-point numbers as approximations.

- Need really high precision there are two approaches.
    - One approach is to use ints—for example, working in terms of pennies or tenths of a penny or similar—and scale the numbers when necessary.
    - The other approach is to use Python's decimal.Decimal numbers from the decimal module. These perform calculations that are accurate to the level of precision we specify (by default, to 28 decimal places) and can represent periodic numbers like 0.1 exactly; but processing is a lot slower than with floats.
        - Because of their accuracy, decimal.Decimal numbers are suitable for financial calculations.

- Mixed mode arithmetic is supported such that using an int and a float produces a float, and using a float and a complex produces a complex.

- Because decimal.Decimals are of fixed precision they can be used only with other decimal.
- Decimals and with ints, in the latter case producing a decimal.Decimal result.
- If an operation is attempted using incompatible types, a TypeError exception is raised.

Floating-Point Numbers

- The float data type can be called as a function—with no arguments it returns 0.0, with a float argument it returns a copy of the argument, and with any other argument it attempts to convert the given object to a float.

- When used for conversions a string argument can be given, either using simple decimal notation or using exponential notation.
- It is possible that NaN (“not a number”) or “infinity” may be produced by a calculation involving floats—unfortunately the behavior is not consistent across implementations and may differ depending on the system’s underlying math library.

Table2.5 TheMathModule’sFunctionsandConstants#1

Syntax	Description
math.acos(x)	Returns the arc cosine of x in radians
math.acosh(x)	Returns the arc hyperbolic cosine of x in radians
math.asin(x)	Returns the arc sine of x in radians
math.asinh(x)	Returns the arc hyperbolic sine of x in radians
math.atan(x)	Returns the arc tangent of x in radians
math.atan2(y, x)	Returns the arc tangent of y / x in radians
math.atanh(x)	Returns the arc hyperbolic tangent of x in radians
math.ceil(x)	Returns ⌈x⌉, i.e., the smallest integer greater than or equal to x as an int; e.g., math.ceil(5.4) == 6
math.copysign(x,y)	Returns x with y’s sign
math.cos(x)	Returns the cosine of x in radians
math.cosh(x)	Returns the hyperbolic cosine of x in radians
math.degrees(r)	Converts float r from radians to degrees math.e The constant e; approximately 2.718 281 828 459 045 1
math.exp(x)	Returns ex, i.e., math.e ** x
math.fabs(x)	Returns x , i.e., the absolute value of x as a float
math.factorial(x)	Returns x!
math.floor(x)	Returns ⌊x⌋, i.e., the largest integer less than or equal to x as an int; e.g., math.floor(5.4) == 5
math.fmod(x, y)	Produces the modulus (remainder) of dividing x by y; this produces better results than % for floats
	Returns a 2-tuple with the mantissa (as a float) and the exponent (as an int) so, x

math.frexp(x)	= m × 2e; see math.ldexp()
math.fsum(i)	Returns the sum of the values in iterable i as a float
math.hypot(x, y)	Returns $\sqrt{}$x2 + y2
math.isinf(x)	Returns True if float x is ± inf (± ∞)
math.isnan(x)	Returns True if float x is nan (“not a number”)
math.ldexp(m, e)	Returns m × 2e; effectively the inverse of math.frexp()
math.log(x, b)	Returns logbx; b is optional and defaults to math.e
math.log10(x)	Returns log10x
math.log1p(x)	Returns loge(1 + x); accurate even when x is close to 0
math.modf(x)	Returns x’s fractional and whole parts as two floats
math.pi	The constant π; approximately 3.141 592 653 589 793 1
math.pow(x, y)	Returns xy as a float
math.radians(d)	Converts float d from degrees to radians
math.sin(x)	Returns the sine of x in radians
math.sinh(x)	Returns the hyperbolic sine of x in radians
math.sqrt(x)	Returns $\sqrt{}$x
math.tan(x)	Returns the tangent of x in radians
math.tanh(x)	Returns the hyperbolic tangent of x in radians
math.trunc(x)	Returns the whole part of x as an int; same as int(x)

```
def equal_float(a, b):
    return abs(a - b) <= sys.float_info.epsilon
```

- The sys.float_info object has many attributes; sys.float_info.epsilon is effectively the smallest difference that the machine can distinguish between two floating-point numbers.
- On one of the author’s 32-bit machines it is just over 0.000 000 000 000 000 2. (Epsilon is the traditional name for this number.)
- Python floats normally provide reliable accuracy for up to 17 significant digits.
- If you type sys.float_info into IDLE, all its attributes will be displayed; these include the minimum and maximum floating-point numbers the machine can represent.

- And typing help(sys.float_info) will print some information about the sys.float_info object.
- Floating-point numbers can be converted to integers using the int() func- tion which returns the whole part and throws away the fractional part, or using round() which accounts for the fractional part, or using math.floor() or math.ceil() which convert down to or up to the nearest integer.
- The float.is_integer() method returns True if a floating-point number's frac- tional part is 0, and a float's fractional representation can be obtained using the float.as_integer_ratio() method.
- For example, given x = 2.75, the call x.as_integer_ratio() returns (11, 4).
- Integers can be converted to floating-point numbers using float().

- Floating-point numbers can also be represented as strings in hexadecimal format using the float.hex() method. Such strings can be converted back to floating-point numbers using the float.fromhex() method.

```
s = 14.25.hex()          # str s == '0x1.c800000000000p+3'
f = float.fromhex(s)     # float f == 14.25
t = f.hex()                 # str t == '0x1.c800000000000p+3'
```

- The exponent is indicated using p ("power") rather than e since e is a valid hexadecimal digit.

- In addition to the built-in floating-point functionality, the math module provides many more functions that operate on floats.

```
>>> import math
>>> math.pi * (5 ** 2) # Python 3.1 outputs: 78.53981633974483
78.539816339744831
>>> math.hypot(5, 12)
13.0
>>> math.modf(13.732) # Python 3.1 outputs: (0.7319999999999993, 13.
0)
(0.73199999999999932, 13.0)
```

- The math module is very dependent on the underlying math library that Python was compiled against.

- This means that some error conditions and boundary cases may behave differently on different platforms.

Complex Numbers

- The complex data type is an immutable type that holds a pair of floats, one representing the real part and the other the imaginary part of a complex number.
- Literal complex numbers are written with the real and imaginary parts joined by a + or - sign, and with the imaginary part followed by a j.
- if the real part is 0, we can omit it entirely.

- The separate parts of a complex are available as attributes real and imag.

```
>>> z = -89.5+2.125j
>>> z.real, z.imag
(-89.5, 2.125)
```

- Except for //, %, divmod(), and the three-argument pow(), all the numeric operators and functions in Table 2.2 (55)➤ can be used with complex numbers, and so can the augmented assignment versions.

- method, conjugate(), which changes the sign of the imaginary part.

```
>>> z.conjugate()
(-89.5-2.125j)
>>> 3-4j.conjugate()
(3+4j)
```

- Python allows us to call methods or access attributes on any literal, as long as the literal's data type provides the called method or the attribute.

- The complex data type can be called as a function—with no arguments it returns 0j, with a complex argument it returns a copy of the argument, and with any other argument it attempts to convert the given object to a complex.
- When used for conversions complex() accepts either a single string argument, or one or two floats.
- If just one float is given, the imaginary part is taken to be 0j.
- The functions in the math module do not work with complex numbers.
- This is a deliberate design decision that ensures that users of the math module get exceptions rather than silently getting complex numbers in some situations.
- Users of complex numbers can import the cmath module, which provides complex number versions of most of the trigonometric and logarithmic functions that are in the math module, plus some complex number-specific functions such as cmath.phase(), cmath.polar(), and cmath.rect(), and also the cmath.pi and cmath.e constants which hold the same float values as their math module coun- terparts.

Decimal Numbers

- In many applications the numerical inaccuracies that can occur when using floats don't matter, and in any case are far outweighed by the speed of calcu- lation that floats offer.
- But in some cases we prefer the opposite trade-off, and want complete accuracy, even at the cost of speed.
- The decimal module provides immutable Decimal numbers that are as accurate as we specify.

- Calculations involving Decimals are slower than those involving floats, but whether this is noticeable will depend on the application.

```
>>> import decimal
>>> a = decimal.Decimal(9876)
>>> b = decimal.Decimal("54321.012345678987654321")
```

```
>>> a + b
Decimal('64197.0123456789876543 21')
```

- Decimal numbers are created using the decimal.Decimal() function.

- This function can take an integer or a string argument—but not a float, since floats are held inexactly whereas decimals are represented exactly.
- If a string is used it can use simple decimal notation or exponential notation.

- In addition to providing accuracy, the exact representation of decimal.Decimals means that they can be reliably compared for equality.

- From Python 3.1 it is possible to convert floats to decimals using the decimal.Decimal.from_float() function.

- This function takes a float as argument and returns the decimal.Decimal that is closest to the number the float approximates.
- All the numeric operators and functions listed in Table 2.2 (55)➤ , including the augmented assignment versions, can be used with decimal.Decimals, but with a couple of caveats.
- If the ** operator has a decimal.Decimal left-hand operand, its right-hand operand must be an integer.
- Similarly, if the pow() function's first argument is a decimal.Decimal, then its second and optional third arguments must be integers.
- The math and cmath modules are not suitable for use with decimal.Decimals, but some of the functions provided by the math module are provided as deci- mal.Decimal methods.
- For example, to calculate ex where x is a float, we write math.exp(x), but where x is a decimal.Decimal, we write x.exp().
- The decimal.Decimal data type also provides ln() which calculates the natural (base e) logarithm (just like math.log() with one argument), log10(), and sqrt(), along with many other methods specific to the decimal.Decimal data type.
- Numbers of type decimal.Decimal work within the scope of a context; the context is a collection of settings that affect how decimal.Decimals behave.
- The context specifies the precision that should be used (the default is 28 decimal places), the rounding technique, and some other details.

- In some situations the difference in accuracy between floats and decimal. Decimals becomes obvious:

```
>>> 23 / 1.05
21.904761904761905
>>> print(23 / 1.05)
21.9047619048
>>> print(decimal.Decimal(23) / decimal.Decimal("1.05"))
21.90476190476190476190476190
>>> decimal.Decimal(23) / decimal.Decimal("1.05")
Decimal('21.90476190476190476190476190')
```

- Although the division using decimal.Decimals is more accurate than the one involving floats, in this case (on a 32-bit machine) the difference only shows up in the fifteenth decimal place.

- When we call print() on the result of decimal.Decimal(23) / decimal.Decimal("1.05") the bare number is printed—this output is in string form.
- If we simply enter the expression we get a decimal.Decimal output—this output is in representational form.
- All Python objects have two output forms.
- String form is designed to be human-readable.
- Representational form is designed to produce output that if fed to a Python interpreter would (when possible) re- produce the represented object.
- Library Reference's decimal module documentation provides all the details.

Strings

- Strings are represented by the immutable str data type which holds a sequence of Unicode characters.
- The str data type can be called as a function to create string objects—with no arguments it returns an empty string, with a nonstring argument it returns the string form of the argument, and with a string argument it returns a copy of the string.
- The str() function can also be used as a conversion function, in which case the first argument should be a string or something convertable to a string, with up to two optional string arguments being passed, one specifying the encoding to use and the other specifying how to handle encoding errors.
- string literals are created using quotes, and that we are free to use single or double quotes providing we use the same at both ends.

- We can use a triple quoted string—this is Python-speak for a string that begins and ends with three quote characters.

```
text = """A triple quoted string like this can include 'quotes' and
 "quotes" without formality. We can also escape newlines \ so this p
articular string is actually only two lines long."""
```

Table2.7 Python'sStringEscapes

Escape	Meaning
\newline Escape	(i.e., ignore) the newline
\	Backslash ()
\'	Single quote (')
\"	Double quote (")
\a	ASCII bell (BEL)
\b	ASCII backspace (BS)

\f	ASCII formfeed (FF)
\n	ASCII linefeed (LF)
\N{name}	Unicode character with the given name
\ooo	Character with the given octal value
\r	ASCII carriage return (CR)
\t	ASCII tab (TAB)
\uhhhh	Unicode character with the given 16-bit hexadecimal value
\Uhhhhhhhh	Unicode character with the given 32-bit hexadecimal value
\v	ASCII vertical tab (VT)
\xhh	Character with the given 8-bit hexadecimal value

- If we want to use quotes inside a normal quoted string we can do so without formality if they are different from the delimiting quotes; otherwise, we must escape them:

```
a = "Single 'quotes' are fine; \"doubles\" must be escaped."
b = 'Single \'quotes\' must be escaped; "doubles" are fine.'
```

- Python uses newline as its statement terminator, except inside parentheses (()), square brackets ([]), braces ({}), or triple quoted strings.
- Newlines can be used without formality in triple quoted strings, and we can include newlines in any string literal using the \n escape sequence.
- In some situations—for example, when writing regular expressions—we need to create strings with lots of literal backslashes.
- This can be inconvenient since each one must be escaped:

```
import re
phone1 = re.compile("^((?:[(]\\d+[)])?\\s*\\d+(?:-\\d+)?)$")
```

- The solution is to use raw strings.
- These are quoted or triple quoted strings whose first quote is preceded by the letter r.
- Inside such strings all characters are taken to be literals, so no escaping is necessary.

```
phone2 = re.compile(r"^((?:[(]\d+[)])?\s*\d+(?:-\d+)?)$")
```

- If we want to write a long string literal spread over two or more lines but without using a triple quoted

string there are a couple of approaches we can take:

```
  t = "This is not the best way to join two long strings " + \ "togeth
er since it relies on ugly newline escaping"
  s = ("This is the nice way to join two long strings " "together; it
relies on string literal concatenation.")
```

- Notice that in the second case we must use parentheses to create a single expression—without them, s would be assigned only to the first string, and the second string would cause an IndentationError exception to be raised.

- The Python documentation's "Idioms and Anti-Idioms" HOWTO document recommends always using parentheses to spread statements of any kind over mul- tiple lines rather than escaping newlines.
- Since .py files default to using the UTF-8 Unicode encoding, we can write any Unicode characters in our string literals without formality.

- We can also put any Unicode characters inside strings using hexadecimal escape sequences or using Unicode names.

```
>>> euros = " \N{euro sign} \u20AC \U000020AC"
>>> print(euros)
```

- Note that Unicode character names are not case-sensitive, and spaces inside them are optional.

- Want to know the Unicode code point (the integer assigned to the charac- ter in the Unicode encoding) for a particular character in a string, we can use the built-in ord() function.

```
>>> ord(euros[0])
8364
>>> hex(ord(euros[0]))
'0x20ac'
```

- Similarly, we can convert any integer that represents a valid code point into the corresponding Unicode character using the built-in chr() function:

```
>>> s = "anarchists are " + chr(8734) + chr(0x23B7)
>>> s
'anarchists are ∞⎷'
>>> ascii(s)
"'anarchists are \u221e\u23b7'"
```

- If we enter s on its own in IDLE, it is output in its string form, which for strings means the characters are output enclosed in quotes.

- Want only ASCII characters, we can use the built-in ascii() function which returns the representational form of its argument using 7-bit ASCII characters where possible, and str.using the shortest form of

\xhh, \uhhhh, or \Uhhhhhhhh escape otherwise.

Comparing Strings

- Strings support the usual comparison operators <, <=, ==, !=, >, and >=. These operators compare strings byte by byte in memory.
- Two problems arise when performing comparisons, such as when sorting lists of strings.
- The first problem is that some Unicode characters can be represented by two or more different byte sequences.
- Character Å (Unicode code point 0x00C5) can be represented in UTF-8 encoded bytes in three different ways: [0xE2, 0x84, 0xAB], [0xC3, 0x85], and [0x41, 0xCC, 0x8A].
- Import the unicodedata module and call unicodedata.normalize() with “NFKC” as the first argument (this is a normalization method—three others are also available, “NFC”, “NFD”, and “NFKD”), and a string containing the Å character using any of its valid byte sequences, the function will return a string that when represented as UTF-8 encoded bytes will always be the byte sequence [0xC3, 0x85].
- The second problem is that the sorting of some characters is language-specific.
- One example is that in Swedish ä is sorted after z, whereas in German, ä is sort- ed as if though were spelled ae.
- In English we sort ø as if it were o, in Danish and Norwegian it is sorted after z.
- And sometimes strings are in a mixture of languages (e.g., some Spanish, others English), and some characters (such as arrows, dingbats, and mathematical symbols) don’t really have mean- ingful sort positions.
- As a matter of policy—to prevent subtle mistakes—Python does not make guesses.
- In the case of string comparisons, it compares using the strings’ in- memory byte representation. This gives a sort order based on Unicode code points which gives ASCII sorting for English.
- Lower or uppercasing all the strings compared produces a more natural English language ordering.
- The whole issue of sorting Unicode strings is explained in detail in the Unicode Collation Algorithm document (unicode.org/reports/tr10).

Slicing and Striding Strings

- Individual characters in a string, can be extracted using the item access operator ([]).
- This operator is much more versatile and can be used to extract not just one item or character, but an entire slice (subsequence) of items or characters, in which context it is referred to as the slice operator.
- Index positions into a string begin at 0 and go up to the length of the string minus 1.

- But it is also possible to use negative index positions—these count from the last character back toward the first.

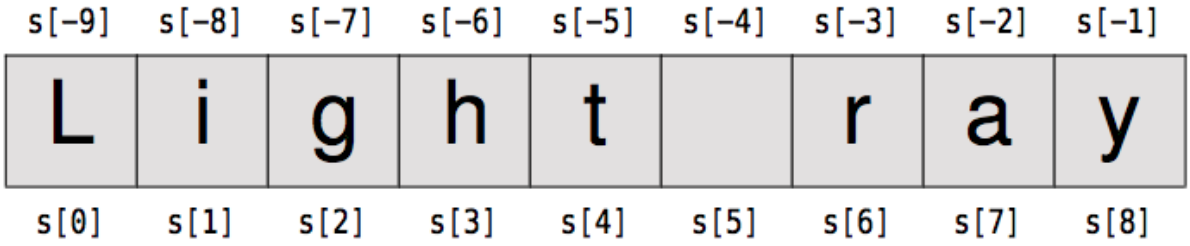

Figure 2.1 *String index positions*

- Negative indexes -1 which always gives us the last character in a string. Accessing an out-of-range index (or any index in an empty string) will cause an IndexError exception to be raised.
- The slice operator has three syntaxes:

```
seq[start]
seq[start:end]
seq[start:end:step]
```

- The seq can be any sequence, such as a list, string, or tuple.
- The start, end, and step values must all be integers (or variables holding integers).
- second syntax extracts a slice from and including the start-th item, up to and excluding the end-th item.
- If we use the second (one colon) syntax, we can omit either of the integer indexes.
- If we omit the start index, it will default to 0.
- If we omit the end index, it will default to len(seq).
- This means that if we omit both indexes, for example, s[:], it is the same as writing s[0:len(s)], and extracts—that is, copies—the entire sequence.

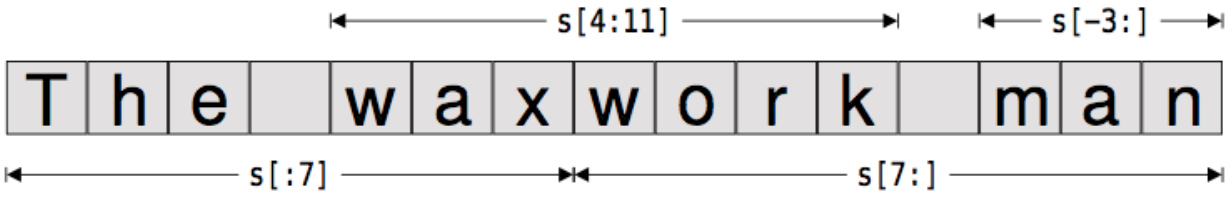

Figure 2.2 *Sequence slicing*

- One way of inserting a substring inside a string is to mix slicing with concatenation.

```
>>> s = s[:12] + "wo" + s[12:]
>>> s
'The waxwork woman'
```

- Using + to concatenate and += to append is not particularly efficient when many strings are involved.
- For joining lots of strings it is usually best to use the str.join() method.
- The third (two colon) slice syntax is like the second, only instead of extracting ➤71 every character,

every step-th character is taken.

- And like the second syntax, we can omit either of the index integers.
- If we omit the start index, it will default to 0—unless a negative step is given, in which case the start index defaults to -1.
- If we omit the end index, it will default to len(seq)—unless a negative step is given, in which case the end index effectively defaults to before the beginning of the string.
- If we use two colons but omit the step size, it will default to 1.
- But there is no point using the two colon syntax with a step size of 1, since that's the default anyway.

- Also, a step size of zero isn't allowed.

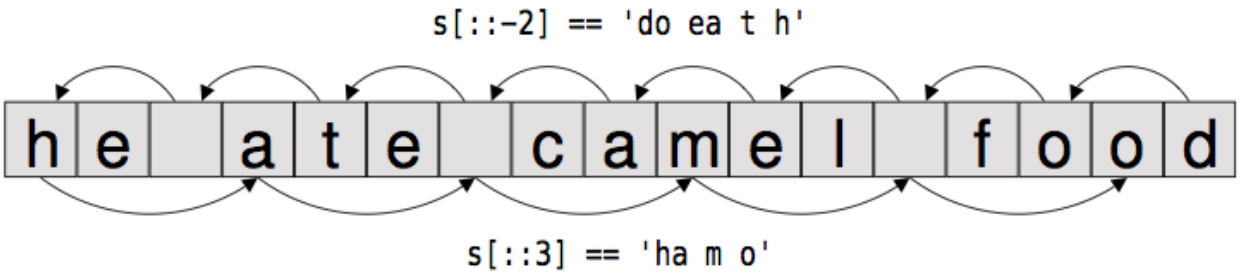

Figure 2.3 *Sequence striding*

- Possible to combine slicing indexes with striding.

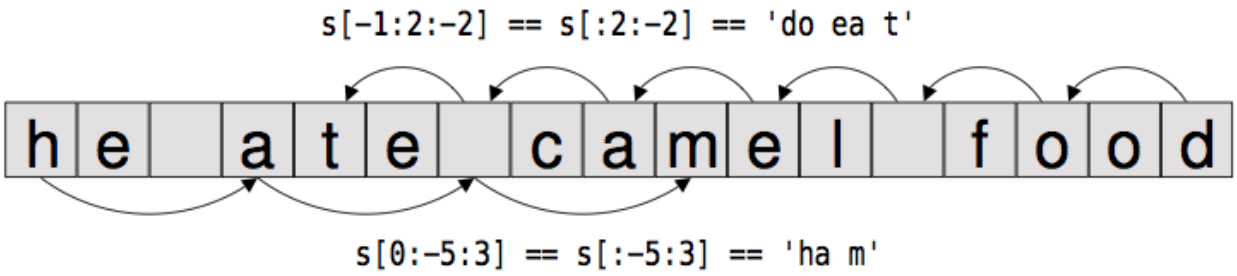

Figure 2.4 *Sequence slicing and striding*

String Operators and Methods

- Strings are immutable sequences, all the functionality that can be used with immutable sequences can be used with strings.
- This includes member- ship testing with in, concatenation with +, appending with +=, replication with *, and augmented assignment replication with *=.
- As strings are sequences they are "sized" objects, and therefore we can call len() with a string as the argument.
- The length returned is the number of characters in the string (zero for an empty string).
- - operator is overloaded to provide string concatenation.
- In cases where we want to concatenate lots of strings the str.join() method.

- The method takes a sequence as an argument (e.g., a list or tuple of strings), and joins them together into a single string with the string the method was called on between each one.

```
>>> treatises = ["Arithmetica", "Conics", "Elements"]
>>> " ".join(treatises)
'Arithmetica Conics Elements'
>>> "-<>-".join(treatises)
'Arithmetica-<>-Conics-<>-Elements'
>>> "".join(treatises)
'ArithmeticaConicsElements'
```

- The str.join() method can also be used with the built-in reversed() function, to reverse a string, “”.join(reversed(s)), although the same result can be achieved more concisely by striding, for example, s[::-1].

- The * operator provides string replication:

```
>>> s = "=" * 5
>>> print(s)
=====
>>> s *= 10
>>> print(s)
==================================================
```

- Can also use the augmented assignment version of the replication operator.★

- To find the position of one string inside another, we have two methods to choose from.
- One is the str.index() method; this returns the index position of the substring, or raises a ValueError exception on failure.
- The other is the str.find() method; this returns the index position of the substring, or -1 on failure.
- Both methods take the string to find as their first argument, and can accept a couple of optional arguments.
- The second argument is the start position in the string being searched, and the third argument is the end position in the string being searched.

> ★Strings also support the % operator for formatting. This operator is deprecated and provided only to ease conversion from Python 2 to Python 3.

Table2.8 StringMethods#1

Syntax	Description
s.capitalize()	Returns a copy of str s with the first letter capitalized; see also the str.title() method
s.center(width, char)	Returns a copy of s centered in a string of length width padded with spaces or optionally with char (a string of length 1); see str.ljust(), str.rjust(), and str.format()
s.count(t, start:end)	Returns the number of occurrences of str t in str s (or in start, end slice of s)

s.encode(encoding, err)	Returns a bytes object that represents the string using the default encoding or using the specified encoding and handling errors according to the optional err argument
s.endswith(x, start, end)	Returns True if s (or the start:end slice of s) ends with str x or with any of the strings in tuplex; otherwise, returns False. See also str.startswith().
s.expandtabs(size)	Returns a copy of s with tabs replaced with spaces in multiples of 8 or of size if specified
s.find(t, start, end)	Returns the leftmost position of t in s (or in the start:end slice of s) or -1 if not found. Use str.rfind() to find the rightmost position. See also str.index().
s.format(⋯)	Returns a copy of s formatted according to the given arguments. This method and its arguments are covered in the next subsection.
s.index(t, start, end)	Returns the leftmost position of t in s (or in the start:end slice of s) or raises ValueError if not found. Use
str.rindex()	to search from the right. See str.find().
s.isalnum()	Returns True if s is nonempty and every character in s is alphanumeric
s.isalpha()	Returns True if s is nonempty and every character in s is alphabetic
s.isdecimal()	Returns True if s is nonempty and every character in s is a Unicode base 10 digit
s.isdigit()	Returns True if s is nonempty and every character in s is an ASCII digit
s.isidentifier()	Returns True if s is nonempty and is a valid identifier
s.islower()	Returns True if s has at least one lowercaseable charac- ter and all its lowercaseable characters are lowercase; see also str.isupper()
s.isnumeric()	Returns True if s is nonempty and every character in s is a numeric Unicode character such as a digit or fraction
s.isprintable()	Returns True if s is empty or if every character in s is con- sidered to be printable, including space, but not newline
s.isspace()	Returns True if s is nonempty and every character in s is a whitespace character
s.istitle()	Returns True if s is a nonempty title-cased string; see also str.title()
s.isupper()	Returns True if str s has at least one uppercaseable char- acter and all its uppercaseable characters are uppercase; see also str.islower()
s.join(seq)	Returns the concatenation of every item in the sequence seq, with str s (which may

	be empty) between each one
s.ljust(width, char)	Returns a copy of s left-aligned in a string of length width padded with spaces or optionally with char (a string of) length 1). Use str.rjust() to right-align and str.center() to center. See also str.format().
s.lower()	Returns a lowercased copy of s; see also str.upper()
s.maketrans()	Companion of str.translate(); see text for details
s.partition(t)	Returns a tuple of three strings—the part of str s before the leftmost str t, t, and the part of s after t; or if t isn't in s returns s and two empty strings. Use str.rpartition() to partition on the rightmost occurrence of t.
s.replace(t, u, n)	Returns a copy of s with every (or a maximum of n if given) occurrences of str t replaced with str u
s.split(t, n)	Returns a list of strings splitting at most n times on str t; if n isn't given, splits as many times as possible; if t isn't given, splits on whitespace. Use str.rsplit() to split from the right—this makes a difference only if n is given and is less than the maximum number of splits possible.
s.splitlines(f)	terminators, stripping the terminators unless f is TrueReturns the list of lines produced by splitting s on line.
s.startswith(x, start, end)	Returns True if s (or the start:end slice of s) starts with str x or with any of the strings in tuple x; otherwise, returns False. See also str.endswith().
s.strip(chars)	Returns a copy of s with leading and trailing whitespace (or the characters in str chars) removed; str.lstrip() strips only at the start, and str.rstrip() strips only at the end
s.swapcase()	Returns a copy of s with uppercase characters lowercased and lowercase characters uppercased; see also str.lower() and str.upper()
s.title()	Returns a copy of s where the first letter of each word is uppercased and all other letters are lowercased; see str.istitle()
s.translate()	Companion of str.maketrans(); see text for details
s.upper()	Returns an uppercased copy of s; see also str.lower()
s.zfill(w)	Returns a copy of s, which if shorter than w is padded with leading zeros to make it w characters long

- if we are looking for multiple index positions, using the str.index() method often produces cleaner

code,

```
def extract_from_tag(tag, line):
    opener = "<" + tag + ">"
    closer = "</" + tag + ">"
    try:
        i = line.index(opener)
        start = i + len(opener)
        j = line.index(closer, start)
        return line[start:j]
    except ValueError:
        return None
```

```
def extract_from_tag(tag, line):
    opener = "<" + tag + ">"
    closer = "</" + tag + ">"
    i = line.find(opener)
    if i != -1:
        start = i + len(opener)
        j = line.find(closer, start)
        if j != -1:
            return line[start:j]
    return None
```

- Both versions of the extract_from_tag() function have exactly the same be- havior.
- For example, extract_from_tag(“red”, “what a rose this is”) returns the string “rose”.
- The exception-handling version on the left separates out the code that does what we want from the code that handles errors, and the error return value version on the right intersperses what we want with error handling.
- The methods str.count(), str.endswith(), str.find(), str.rfind(), str.index(), str.rindex(), and str.startswith() all accept up to two optional arguments: a start position and an end position.

```
s.count("m", 6) == s[6:].count("m")
s.count("m", 5, -3) == s[5:-3].count("m")
```

- The string methods that accept start and end indexes operate on the slice of the string specified by those indexes.

```
result = s.rpartition("/")
```

```
i = s.rfind("/")
if i == -1:
```

```
    result = "", "", s
else:
    result = s[:i], s[i], s[i + 1:]
```

- The left and right-hand code snippets are not quite equivalent because the one on the right also creates a new variable, i.

- Notice that we can assign tuples without formality, and that in both cases we looked for the rightmost occurrence of /.
- If s is the string "/usr/local/bin/firefox", both snippets produce the same result: ('/usr/local/bin', '/', 'firefox').

- Can use str.endswith() (and str.startswith()) with a single string argument, for example, s.startswith("From:"), or with a tuple of strings.

```
if filename.lower().endswith((".jpg", ".jpeg")):
    print(filename, "is a JPEG image")
```

- The is*() methods such as isalpha() and isspace() return True if the string they are called on has at least one character, and every character in the string meets the criterion.

```
>>> "917.5".isdigit(), "".isdigit(), "-2".isdigit(), "203".isdigit()

(False, False, False, True)
```

- The is*() methods work on the basis of Unicode character classifications, str.isdigit() on the strings "\N{circled digit two}03" and "②03" returns True for both of them.

- For this reason we cannot assume that a string can be converted to an integer when isdigit() returns True.
- Can strip whitespace from the left using str.lstrip(), from the right using str.rstrip(), or from both ends using str.strip().

- We can also give a string as an argument to the strip methods, in which case every occurrence of every character given will be stripped from the appropriate end or ends.

```
>>> s = "\t no parking "
>>> s.lstrip(), s.rstrip(), s.strip()
('no parking ', '\t no parking', 'no parking')
>>> "<[unbracketed]>".strip("[](){}<>")
'unbracketed'
```

- Can also replace strings within strings using the str.replace() method.

- This method takes two string arguments, and returns a copy of the string it is called on with every occurrence of the first string replaced with the second.

- If the second argument is an empty string the effect is to delete every occurrence of the first string.

```
>>> record = "Leo Tolstoy*1828-8-28*1910-11-20"
>>> fields = record.split("*")
>>> fields
['Leo Tolstoy', '1828-8-28', '1910-11-20']
```

```
>>> born = fields[1].split("-")
>>> born
['1828', '8', '28']
>>> died = fields[2].split("-")
>>> print("lived about", int(died[0]) - int(born[0]), "years")
lived about 82 years
```

- The str.maketrans() method is used to create a translation table which maps characters to characters.
- (two argument) call where the first argument is a string containing characters to translate from and the second argument is a string containing the characters to translate to.
- Both arguments must be the same length.
- The str.translate() method takes a translation table as an argument and returns a copy of its string with the characters translated according to the translation table.

```
table = "".maketrans("\N{bengali digit zero}"
    "\N{bengali digit one}\N{bengali digit two}"
    "\N{bengali digit three}\N{bengali digit four}"
    "\N{bengali digit five}\N{bengali digit six}"
    "\N{bengali digit seven}\N{bengali digit eight}"
    "\N{bengali digit nine}", "0123456789")
print("20749".translate(table)) # prints: 20749
print("\N{bengali digit two}07\N{bengali digit four}"
      "\N{bengali digit nine}".translate(table)) # prints: 20749
```

- Notice that we have taken advantage of Python's string literal concatenation inside the str.maketrans() call and inside the second print() call to spread strings over multiple lines without having to escape newlines or use explicit concatenation.
- We called str.maketrans() on an empty string because it doesn't matter what string it is called on; it simply processes its arguments and returns a transla- tion table.
- More sophisticated character translations are required, we could create a custom codec—see the codecs module.
- difflib which can be used to show differences between files or between strings, the io module's io.StringIO class which allows us to read from or write to strings as though they were files, and the textwrap module which provides facilities for wrapping and filling strings.
- There is also a string module that has a few useful constants such as ascii_letters and

ascii_lowercase.

String Formatting with the str.format() Method

- The str.format() method provides a very flexible and powerful way of creating strings.
- The str.format() method returns a new string with the replacement fields in its string replaced with its arguments suitably formatted.

```
>>> "The novel '{0}' was published in {1}".format("Hard Times", 1854
)
"The novel 'Hard Times' was published in 1854"
```

- Each replacement field is identified by a field name in braces. If the field name is a simple integer, it is taken to be the index position of one of the arguments passed to str.format().
- If we need to include braces inside format strings, we can do so by doubling them up.

```
>>> "{{{0}}} {1} ;-}}".format("I'm in braces", "I'm not")
"{I'm in braces} I'm not ;-}"
```

- If we try to concatenate a string and a number, Python will quite rightly raise a TypeError.

```
>>> "{0}{1}".format("The amount due is $", 200)
'The amount due is $200'
```

- We can also concatenate strings using str.format() (although the str.join() method is best for this):

```
>>> x = "three"
>>> s ="{0} {1} {2}"
>>> s = s.format("The", x, "tops")
>>> s
'The three tops'
```

- The replacement field can have any of the following general syntaxes:

```
{field_name}
{field_name!conversion}
{field_name:format_specification}
{field_name!conversion:format_specification}
```

- One other point to note is that replacement fields can contain replacement fields. Nested replacement fields cannot have any formatting

Field Names

- A field name can be either an integer corresponding to one of the str.format() method's arguments, or the name of one of the method's keyword arguments.

```
>>> "{who} turned {age} this year".format(who="She", age=88)
'She turned 88 this year'
>>> "The {who} was {0} last week".format(12, who="boy")
'The boy was 12 last week'
```

- First example uses two keyword arguments, who and age, and the second example uses one positional argument (the only kind we have used up to now) and one keyword argument.
- We can make use of any arguments in any order inside the format string.
- Field names may refer to collection data types.
- In such cases we can include an index (not a slice!) to identify a particular item:

```
>>> stock = ["paper", "envelopes", "notepads", "pens", "paper clips"
]
>>> "We have {0[1]} and {0[2]} in stock".format(stock)
'We have envelopes and notepads in stock'
```

- Python dictionaries. These store key–value items, and since they can be used with str.format().

```
>>> d = dict(animal="elephant", weight=12000)
>>> "The {0[animal]} weighs {0[weight]}kg".format(d)
'The elephant weighs 12000kg'
```

- Just as we access list and tuple items using an integer position index, we access dictionary items using a key.
- We can also access named attributes.

```
>>> "math.pi=={0.pi} sys.maxunicode=={1.maxunicode}".format(math, sy
s)
'math.pi==3.14159265359 sys.maxunicode==65535'
```

- The field name syntax allows us to refer to positional and key- word arguments that are passed to the str.format() method.
- If the arguments are collection data types like lists or dictionaries, or have attributes, we can ac- cess the part we want using [] or . notation

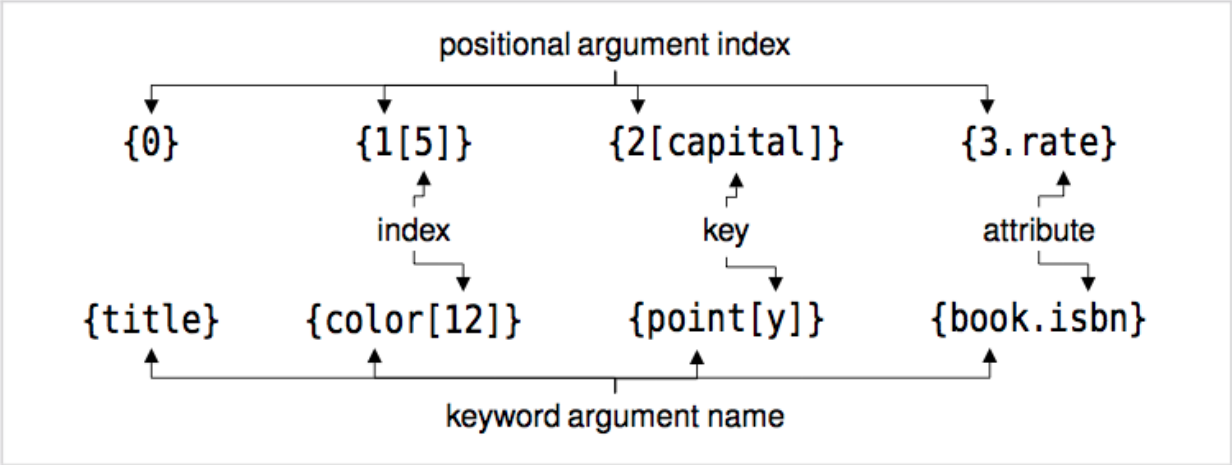

Figure 2.5 *Annotated format specifier field name examples*

- From Python 3.1 it is possible to omit field names, in which case Python will in effect put them in for us, using numbers starting from 0.

```
>>> "{} {} {}".format("Python", "can", "count")
'Python can count'
```

- The local variables that are currently in scope are available from the built-in locals() function.
- This function returns a dictionary whose keys are local Map-ping unpack- ing variable names and whose values are references to the variables' values.
- Can use mapping unpacking to feed this dictionary into the str.format() method.
- The mapping unpacking operator is ** and it can be applied to a mapping (such as a dictionary) to produce a key–value list suitable for passing to a function.

```
>>> element = "Silver"
>>> number = 47
>>> "Element {number} is {element}".format(**locals())
'Element 47 is Silver'
```

- We can use variable names in format strings and leave Python to fill in their values simply by unpacking the dictionary returned by locals()—or some other dictionary—into the str.format() method.

```
>>> "The {animal} weighs {weight}kg".format(**d)
'The elephant weighs 12000kg'
```

- Unpacking a dictionary into the str.format() method allows us to use the dictionary's keys as field names.
- This makes string formats much easier to understand, and also easier to maintain, since they are not dependent on the order of the arguments.
- Note, however, that if we want to pass more than one argument to str.format(), only the last one can

use mapping unpacking.

- Decimal.Decimal numbers we noticed that such numbers Parameter unpacking are output in one of two ways.

```
>>> decimal.Decimal("3.4084")
Decimal('3.4084')
>>> print(decimal.Decimal("3.4084"))
3.4084
```

- The first way that the decimal.Decimal is shown is in its representational form.

- The purpose of this form is to provide a string which if interpreted by Python would re-create the object it represents.
- Not all objects can provide a reproducing representation, in which case they provide a string enclosed in angle brackets.
- For example, the repre- sentational form of the sys module is the string "\".
- The second way that decimal.Decimal is shown is in its string form.
- This form is aimed at human readers, so the concern is to show something that makes sense to people.
- If a data type doesn't have a string form and a string is required, Python will use the representational form.
- Python's built-in data types know about str.format(), and when passed as an argument to this method they return a suitable string to display themselves.
- In addition, it is possible to override the data type's normal behavior and force it to provide either its string or its representational form.
- This is done by adding a conversion specifier to the field.
- Currently there are three such specifiers: s to force string form, r to force representational form, eval() and a to force representational form but only using ASCII characters.
- Want to avoid non-ASCII characters we can use either ascii(movie) or "{0!a}".format(movie).

Format Specifications

- For strings, the things that we can control are the fill character, the alignment within the field, and the minimum and maximum field widths.
- A string format specification is introduced with a colon (:) and this is followed by an optional pair of characters—a fill character (which may not be }) and an alignment character (< for left align, ^ for center, > for right align).
- Then comes an optional minimum width integer, and if we want to specify a maximum width, this comes last as a period followed by an integer.
- Note that if we specify a fill character we must also specify an alignment.

- We omit the sign and type parts of the format specification because they have no effect on strings.

```
>>> s = "The sword of truth"
>>> "{0}".format(s)      # default formatting
```

```
'The sword of truth'
```

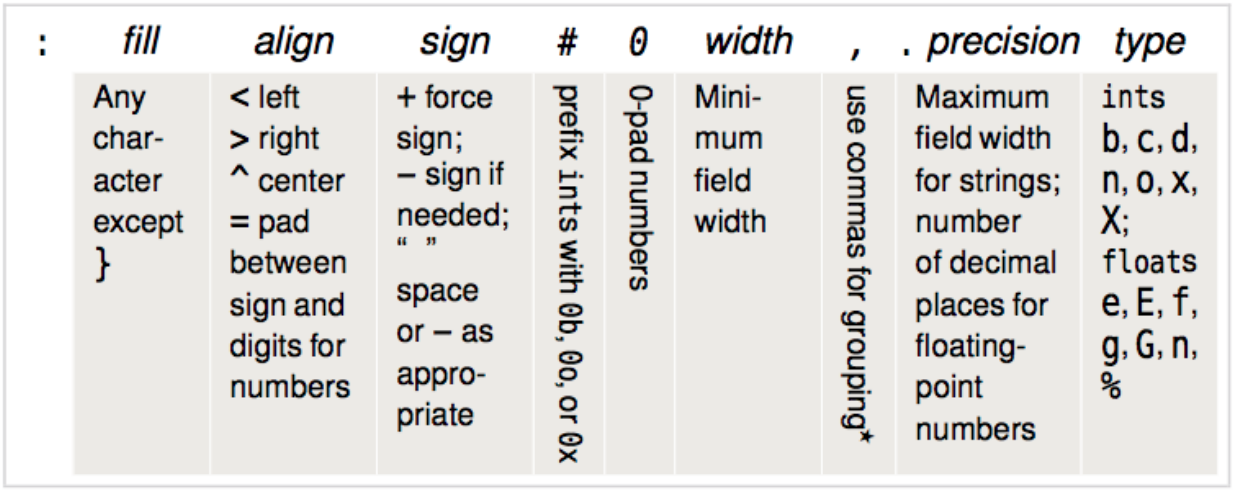

Figure 2.6 *The general form of a format specification*

```
>>> "{0:25}".format(s) # minimum width 25
'The sword of truth '
>>> "{0:>25}".format(s) # right align, minimum width 25
' The sword of truth'
>>> "{0:^25}".format(s) # center align, minimum width 25
' The sword of truth '
>>> "{0:-^25}".format(s) # - fill, center align, minimum width 25
'---The sword of truth----'
>>> "{0:.<25}".format(s) # . fill, left align, minimum width 25
'The sword of truth.......'
>>> "{0:.10}".format(s) # maximum width 10
'The sword '
```

- We had to specify the left alignment (even though this is the default).
- If we left out the <, we would have :.25, and this simply means a maximum field width of 25 characters.
- Two ways of setting a string's maximum width using a maxwidth variable:

```
>>> maxwidth = 12
>>> "{0}".format(s[:maxwidth])
'The sword of'
>>> "{0:.{1}}".format(s, maxwidth)
'The sword of'
```

- The first approach uses standard string slicing; the second uses an inner replacement field.
- For integers, the format specification allows us to control the fill character, the alignment within the field, the sign, whether to use a nonlocale-aware comma separator to group digits (from Python 3.1),

the minimum field width, and the number base.

- An integer format specification begins with a colon, after which we can have an optional pair of characters—a fill character (which may not be }) and an alignment character (< for left align, ^ for center, > for right align, and = for the filling to be done between the sign and the number).
- Next is an optional sign character: + forces the output of the sign, - outputs the sign only for negative numbers, and a space outputs a space for positive numbers and a - sign for negative numbers.
- Then comes an optional minimum width integer—this can be preceded by a # character to get the base prefix output (for binary, octal, and hexadecimal numbers), and by a 0 to get 0-padding.
- Then, from Python 3.1, comes an optional comma—if present this will cause the number's digits to be grouped into threes with a comma separating each group.
- If we want the output in a base other than decimal we must add a type character—b for binary, o for octal, x for lowercase hexadecimal, and X for uppercase hexadecimal, al- though for completeness, d for decimal integer is also allowed.
- There are two other type characters: c, which means that the Unicode character corresponding to the integer should be output, and n, which outputs numbers in a locale- sensitive way.

- Get 0-padding in two different ways:

```
>>> "{0:0=12}".format(8749203)          # 0 fill, minimum width 12
'000008749203'
>>> "{0:0=12}".format(-8749203)          # 0 fill, minimum width 12
'-00008749203'
>>> "{0:012}".format(8749203)          # 0-pad and minimum width 12
'000008749203'
>>> "{0:012}".format(-8749203)          # 0-pad and minimum width 12
'-00008749203'
```

- The first two examples have a fill character of 0 and fill between the sign and the number itself (=).

- The second two examples have a minimum width of 12 and 0-padding.

```
>>> "{0:*<15}".format(18340427) # * fill, left align, min width 15
'18340427*******'
>>> "{0:*>15}".format(18340427) # * fill, right align, min width 15
'*******18340427'
>>> "{0:*^15}".format(18340427) # * fill, center align, min width 15

'***18340427****'
>>> "{0:*^15}".format(-18340427) # * fill, center align, min width 1
5
'***-18340427***'
```

- Effects of the sign characters:

```
>>> "{0:b} {0:o} {0:x} {0:X}".format(14613198)
```

```
'110111101111101011001110 67575316 deface DEFACE'
>>> "{0:#b} {0:#o} {0:#x} {0:#X}".format(14613198)
'0b110111101111101011001110 0o67575316 0xdeface 0XDEFACE'
```

- It is not possible to specify a maximum field width for integers.
- This is because doing so might require digits to be chopped off, thereby rendering the integer meaningless.
- Python 3.1 and use a comma in the format specification, the integer will use commas for grouping.

```
 >>> "{0:,}  {0:*>13,}".format(int(2.39432185e6))
'2,394,321  ****2,394,321'
```

- This is very convenient for many scientific and financial programs, but it does not take into account the current locale.
- For example, many Continental Europeans would expect the thousands separator to be . and the decimal separator to be ,.
- Format character available for integers (and which is also available for floating-point numbers) is n.
- This has the same effect as d when given an integer and the same effect as g when given a floating-point number.
- What makes n special is that it respects the current locale, and will use the locale-specific decimal separator and grouping separator in the output it produces.
- The default locale is called the C locale, and for this the decimal and grouping characters are a period and an empty string.
- We can respect the user's locale by starting our programs with the following two lines as the first executable statements:

```
import locale
locale.setlocale(locale.LC_ALL, "")
```

- Passing an empty string as the locale tells Python to try to automatically determine the user's locale (e.g., by examining the LANG environment variable), with a fallback of the C locale.

```
 x, y = (1234567890, 1234.56)
locale.setlocale(locale.LC_ALL, "C")
c = "{0:n} {1:n}".format(x, y) # c == "1234567890 1234.56"
locale.setlocale(locale.LC_ALL, "en_US.UTF-8")
en = "{0:n} {1:n}".format(x, y) # en == "1,234,567,890 1,234.56"
locale.setlocale(locale.LC_ALL, "de_DE.UTF-8")
de = "{0:n} {1:n}".format(x, y) # de == "1.234.567.890 1.234,56"
```

- For floating-point numbers, the format specification gives us control over the fill character, the alignment within the field, the sign, whether to use a nonlocale aware comma separator to group digits

(from Python 3.1), the minimum field width, the number of digits after the decimal place, and whether to present the number in standard or exponential form, or as a percentage.

- The format specification for floating-point numbers is the same as for integers, except for two differences at the end.
- After the optional minimum width—from Python 3.1, after the optional grouping comma—we can specify the number of digits after the decimal place by writing a period followed by an integer.
- Wecan also add a type character at the end: e for exponential form with a lowercase e, E for exponential form with an uppercase E, f for standard floating-point form, g for "general" form—this is the same as f unless the number is very large.

- Also available is %—this results in the number being multiplied by 100 with the resultant number output in f format with a % symbol appended.

```
>>> amount = (10 ** 3) * math.pi
>>> "[{0:12.2e}] [{0:12.2f}]".format(amount)
'[    3.14e+03] [     3141.59]'
>>> "[{0:*>12.2e}] [{0:*>12.2f}]".format(amount)
'[****3.14e+03] [*****3141.59]'
>>> "[{0:*>+12.2e}] [{0:*>+12.2f}]".format(amount)
'[***+3.14e+03] [****+3141.59]'
```

- In Python 3.0, decimal.Decimal numbers are treated by str.format() as strings rather than as numbers.

- From Python 3.1, decimal.Decimal numbers can be formatted as floats, in- cluding support for , to get comma-separated groups.

- Here is an example—we have omitted the field name since we don't need it for Python 3.1:

```
 >>> "{:,.6f}".format(decimal.Decimal("1234567890.1234567890"))
'1,234,567,890.123457'
```

- If we omitted the f format character (or used the g format character), the number would be formatted as '1.23457E+9'.

- Python 3.0 does not provide any direct support for formatting complex numbers—support was added with Python 3.1.

- However, we can easily solve this by formatting the real and imaginary parts as individual floating-point numbers.

```
>>> "{0.real:.3f}{0.imag:+.3f}j".format(4.75917+1.2042j)
'4.759+1.204j'
>>> "{0.real:.3f}{0.imag:+.3f}j".format(4.75917-1.2042j)
'4.759-1.204j'
```

- We access each attribute of the complex number individually, and format them both as floating-point numbers.
- Python 3.1 supports formatting complex numbers using the same syntax as for floats:

```
>>> "{:,.4f}".format(3.59284e6-8.984327843e6j)
'3,592,840.0000-8,984,327.8430j'
```

- One slight drawback of this approach is that exactly the same formatting is applied to both the real and the imaginary parts; but we can always use the Python 3.0 technique of accessing the complex number's attributes individually if we want to format each one differently.

Example: print_unicode.py

- print_unicode.py spoked

decimal	hex	chr	name
10018	2722	✢	Four Teardrop-Spoked Asterisk
10019	2723	✣	Four Balloon-Spoked Asterisk
10020	2724	✤	Heavy Four Balloon-Spoked Asterisk
10021	2725	✥	Four Club-Spoked Asterisk
10035	2733	✳	Eight Spoked Asterisk
10043	273B	✻	Teardrop-Spoked Asterisk
10044	273C	✼	Open Centre Teardrop-Spoked Asterisk
10045	273D	✽	Heavy Teardrop-Spoked Asterisk
10051	2743	❃	Heavy Teardrop-Spoked Pinwheel Asterisk
10057	2749	❉	Balloon-Spoked Asterisk
10058	274A	❊	Eight Teardrop-Spoked Propeller Asterisk
10059	274B	❋	Heavy Eight Teardrop-Spoked Propeller Asterisk

- If run with no arguments, the program produces a table of every Unicode character, starting from the space character and going up to the character with the highest available code point.
- If an argument is given, as in the example, only those rows in the table where the lowercased Unicode character name contains the argument are printed.

```
word = None
if len(sys.argv) > 1:
```

```
        if sys.argv[1] in ("-h", "--help"):
            print("usage: {0} [string]".format(sys.argv[0]))
            word = 0
        else:
            word = sys.argv[1].lower()
    if word != 0:
        print_unicode_table(word)
```

> ★ This program assumes that the console uses the Unicode UTF-8 encoding. Unfortunate- ly, the Windows console has poor UTF-8 support. As a workaround, the examples include print_unicode_uni.py, a version of the program that writes its output to a file which can then be opened using a UTF-8-savvy editor, such as IDLE.

- Code variable, initializing it to the code point for a space (0x20).
- Set the end variable to be the highest Unicode code point available—this will vary depending on whether Python was compiled to use the UCS-2 or the UCS-4 character encoding.
- We get the Unicode character that corresponds to the code point using the chr() function.
- The unicodedata.name() function returns the Unicode character name for the given Unicode character; its optional second argument is the name to use if no character name is defined.

Character Encodings

- In the U.S. and Western Europe the Latin-1 encoding was often used; its characters in the range 0x20–0x7E are the same as the corresponding characters in 7-bit ASCII, with those in the range 0xA0–0xFF used for accented characters and other symbols needed by those using non-English Latin alphabets.
- One solution that has been almost universally adopted is the Unicode encoding.
- Unicode assigns every charac- ter to an integer—called a code point in Unicode-speak.
- Unicode is not limited to using one byte per character, and is therefore able to represent every character in every language in a single encod-ing, so unlike other encodings, Unicode can handle characters from a mixture of languages, rather than just one.
- Currently, slightly more than 100000 Unicode characters are defined, so even using signed numbers, a 32-bit integer is more than adequate to store any Unicode code point.
- So the simplest way to store Unicode characters is as a sequence of 32-bit integers, one integer per character.
- However, in practice things aren't so simple, since some Unicode characters can be represented by one or by two code points—for example, é can be represented by the single code point 0xE9 or by two code points, 0x65 and 0x301 (e and a combining acute accent).
- Unicode is usually stored both on disk and in memory using UTF- 8, UTF-16, or UTF-32.
- UTF-8, is backward compatible with 7-bit ASCII since its first 128 code points are represented by single-byte values that are the same as the 7-bit ASCII character values.
- To represent all the other Unicode characters, UTF-8 uses two, three, or more bytes per character.
- This makes UTF-8 very compact for representing text that is all or mostly English.
- UTF-8 is becoming the de facto standard format for storing Unicode text in files—for example, UTF-8 is the default format for XML, and many web pages these days use UTF-8.

- Java, uses UCS-2 (which in modern form is the same as UTF-16). This representation uses two or four bytes per character, with the most common characters represented by two bytes.
- The UTF-32 rep- resentation (also called UCS-4) uses four bytes per character.
- Using UTF-16 or UTF-32 for storing Unicode in files or for sending over a network connection has a potential pitfall: If the data is sent as integers then the endianness matters.
- One solution to this is to precede the data with a byte order mark so that readers can adapt accordingly.
- This problem doesn't arise with UTF-8, which is another reason why it is so popular.
- Python represents Unicode using either UCS-2 (UTF-16) format, or UCS-4 (UTF-32) format.
- In fact, when using UCS-2, Python uses a slightly simplified version that always uses two bytes per character and so can only represent code points up to 0xFFFF.
- When using UCS-4, Python can represent all the Unicode code points. The maximum code point is stored in the read-only sys.maxunicode attribute—if its value is 65535.

- The str.encode() method returns a sequence of bytes—actually a bytes object.

```
>>> artist = "Tage Åsén"
>>> artist.encode("Latin1")
b'Tage \xc5s\xe9n'
>>> artist.encode("CP850")
b'Tage \x8fs\x82n'
>>> artist.encode("utf8")
b'Tage \xc3\x85s\xc3\xa9n'
>>> artist.encode("utf16")
b'\xff\xfeT\x00a\x00g\x00e\x00 \x00\xc5\x00s\x00\xe9\x00n\x00'
```

- A b before an opening quote signifies a bytes literal rather than a string literal.

- As a convenience, when creating bytes literals we can use a mixture of printable ASCII characters and hexadecimal escapes.
- We cannot encode Tage Åsén's name using the ASCII encoding because it does not have the Å character or any accented characters, so attempting to do so will result in a UnicodeEncodeError exception being raised.
- For UTF-16, the first two bytes are the byte order mark—these are used by the decoding function to detect whether the data is big or little-endian so that it can adapt accordingly.
- It is worth noting a couple more points about the str.encode() method.
- The first argument (the encoding name) is case-insensitive, and hyphens and underscores in the name are treated as equivalent, so "us-ascii" and "US_ASCII" are considered the same.
- The method can also accept an optional second argument which is used to tell it how to handle errors.
- Can encode any string into ASCII if we pass a second argument of "ignore" or "replace"—at the price of losing data, of course—or losslessly if we use "backslashreplace" which replaces non-ASCII characters with \x, \u, and \U escapes.

- The complement of str.encode() is bytes.decode() (and bytearray.decode()) which returns a string with the bytes decoded using the given encoding.

```
>>> print(b"Tage \xc3\x85s\xc3\xa9n".decode("utf8"))
Tage Åsén
>>> print(b"Tage \xc5s\xe9n".decode("latin1"))
Tage Åsén
```

- Python .py files use UTF-8, so Python always knows the encoding to use with string literals.
- This means that we can type any Unicode characters into our strings—providing our editor supports this.★
- When Python reads data from external sources such as sockets, it cannot know what encoding is used, so it returns bytes which we can then decode accordingly.
- For text files Python takes as ofter approach, using the local encoding unless we specify an encoding explicitly.
- Fortunately, some file formats specify their encoding. For example, we can assume that an XML file uses UTF-8, unless the <?xml?> directive explicitly specifies a different encoding.