**Using Python as a Calculator**

Let’s try some simple Python commands. Start the interpreter and wait for the primary prompt, >>>. (It

shouldn’t take long.)

**Numbers**

The interpreter acts as a simple calculator: you can type an expression at it and it will write the value.

Expression syntax is straightforward: the operators +, -, \* and / work just like in most other languages (for

example, Pascal or C); parentheses (()) can be used for grouping. For example:

**>>>** 2 + 2

4

**>>>** 50 - 5\*6 20

**>>>** (50 - 5\*6) / 4

5.0

**>>>** 8 / 5 *# division always returns a floating point number*

1.6

The integer numbers (e.g. 2, 4, 20) have type int, the ones with a fractional part (e.g. 5.0, 1.6) have type

float. We will see more about numeric types later in the tutorial.

Division (/) always returns a ﬂoat. To do *ﬂoor division* and get an integer result (discarding any fractional result) you can use the // operator; to calculate the remainder you can use :

**>>>** 17 / 3 *# classic division returns a float*

5.666666666666667

>>>

**>>>** 17 // 3 *# floor division discards the fractional part*

5

**>>>** 17 3 *# the operator returns the remainder of the division*

2

**>>>** 5 \* 3 + 2 *# result \* divisor + remainder*

17

With Python, it is possible to use the \*\* operator to calculate powers1:

**>>>** 5 \*\* 2 *# 5 squared*

25

**>>>** 2 \*\* 7 *# 2 to the power of 7*

128

The equal sign (=) is used to assign a value to a variable. Afterwards, no result is displayed before the next interactive prompt:

**>>>** width = 20

**>>>** height = 5 \* 9

**>>>** width \* height 900

If a variable is not “deﬁned” (assigned a value), trying to use it will give you an error:

**>>>** n *# try to access an undefined variable*

Traceback (most recent call last): File "<stdin>", line 1, in <module>

NameError: name 'n' is not defined

There is full support for ﬂoating point; operators with mixed type operands convert the integer operand to ﬂoating point:

**>>>** 4 \* 3.75 - 1

14.0

In interactive mode, the last printed expression is assigned to the variable \_. This means that when you are using Python as a desk calculator, it is somewhat easier to continue calculations, for example:

**>>>** tax = 12.5 / 100

**>>>** price = 100.50

**>>>** price \* tax 12.5625

**>>>** price + \_ 113.0625

**>>>** round(\_, 2)

113.06

This variable should be treated as read-only by the user. Don’t explicitly assign a value to it — you would create an independent local variable with the same name masking the built-in variable with its magic behavior.

In addition to int and float, Python supports other types of numbers, such as Decimal and Fraction. Python also has built-in support for complex numbers, and uses the j or J suﬃx to indicate the imaginary part (e.g. 3+5j).

**Strings**

Besides numbers, Python can also manipulate strings, which can be expressed in several ways. They can be enclosed in single quotes ('...') or double quotes ("...") with the same result2. \ can be used to escape quotes:

**>>>** 'spam eggs' *# single quotes*

'spam eggs'

**>>>** 'doesn**\'**t' *# use \' to escape the single quote...*

"doesn't"

**>>>** "doesn't" *# ...or use double quotes instead*

"doesn't"

**>>>** '"Yes," they said.' '"Yes," they said.'

**>>>** "**\"**Yes,**\"** they said." '"Yes," they said.'

**>>>** '"Isn**\'**t," they said.'

'"Isn\'t," they said.'

In the interactive interpreter, the output string is enclosed in quotes and special characters are escaped with backslashes. While this might sometimes look diﬀerent from the input (the enclosing quotes could change), the two strings are equivalent. The string is enclosed in double quotes if the string contains a single quote and no double quotes, otherwise it is enclosed in single quotes. The print() function produces a more readable output, by omitting the enclosing quotes and by printing escaped and special characters:

**>>>** '"Isn**\'**t," they said.'

'"Isn\'t," they said.'

**>>>** print('"Isn**\'**t," they said.') "Isn't," they said.

**>>>** s = 'First line.**\n**Second line.' *# \n means newline*

**>>>** s *# without print(), \n is included in the output*

'First line.\nSecond line.'

**>>>** print(s) *# with print(), \n produces a new line*

First line.

Second line.

If you don’t want characters prefaced by \ to be interpreted as special characters, you can use *raw strings*

by adding an r before the ﬁrst quote:

**>>>** print('C:\some**\n**ame') *# here \n means newline!*

C:\some ame

**>>>** print(r'C:\some\name') *# note the r before the quote*

C:\some\name

String literals can span multiple lines. One way is using triple-quotes: """...""" or '''...'''. End of lines are automatically included in the string, but it’s possible to prevent this by adding a \ at the end of the line. The following example:

print("""**\**

Usage: thingy [OPTIONS]

-h

-H hostname

""")

Display this usage message Hostname to connect to

produces the following output (note that the initial newline is not included):

Usage: thingy [OPTIONS]

-h

-H hostname

Display this usage message Hostname to connect to

Strings can be concatenated (glued together) with the + operator, and repeated with \*:

**>>>** *# 3 times 'un', followed by 'ium'*

**>>>** 3 \* 'un' + 'ium' 'unununium'

Two or more *string literals* (i.e. the ones enclosed between quotes) next to each other are automatically concatenated.

**>>>** 'Py' 'thon' 'Python'

This feature is particularly useful when you want to break long strings:

**>>>** text = ('Put several strings within parentheses '

**...** 'to have them joined together.')

**>>>** text

'Put several strings within parentheses to have them joined together.'

This only works with two literals though, not with variables or expressions:

**>>>** prefix = 'Py'

**>>>** prefix 'thon' *# can't concatenate a variable and a string literal*

...

SyntaxError: invalid syntax

**>>>** ('un' \* 3) 'ium'

...

SyntaxError: invalid syntax

If you want to concatenate variables or a variable and a literal, use +:

**>>>** prefix + 'thon' 'Python'

Strings can be *indexed* (subscripted), with the ﬁrst character having index 0. There is no separate character type; a character is simply a string of size one:

**>>>** word = 'Python'

**>>>** word[0] *# character in position 0*

'P'

**>>>** word[5] *# character in position 5*

'n'

Indices may also be negative numbers, to start counting from the right:

**>>>** word[-1] *# last character*

'n'

**>>>** word[-2] *# second-last character*

'o'

**>>>** word[-6] 'P'

Note that since -0 is the same as 0, negative indices start from -1.

In addition to indexing, *slicing* is also supported. While indexing is used to obtain individual characters,

*slicing* allows you to obtain substring:

**>>>** word[0:2] *# characters from position 0 (included) to 2 (excluded)*

'Py'

**>>>** word[2:5] *# characters from position 2 (included) to 5 (excluded)*

'tho'

Note how the start is always included, and the end always excluded. This makes sure that s[:i] + s[i:]

is always equal to s:

**>>>** word[:2] + word[2:] 'Python'

**>>>** word[:4] + word[4:] 'Python'

Slice indices have useful defaults; an omitted ﬁrst index defaults to zero, an omitted second index defaults to the size of the string being sliced.

**>>>** word[:2] *# character from the beginning to position 2 (excluded)*

'Py'

**>>>** word[4:] *# characters from position 4 (included) to the end*

'on'

**>>>** word[-2:] *# characters from the second-last (included) to the end*

'on'

One way to remember how slices work is to think of the indices as pointing *between* characters, with the left edge of the ﬁrst character numbered 0. Then the right edge of the last character of a string of *n* characters has index *n*, for example:

+---+---+---+---+---+---+

| P | y | t | h | o | n |

+---+---+---+---+---+---+ 0 1 2 3 4 5 6

-6 -5 -4 -3 -2 -1

The ﬁrst row of numbers gives the position of the indices 0…6 in the string; the second row gives the corresponding negative indices. The slice from *i* to *j* consists of all characters between the edges labeled *i* and *j*, respectively.

For non-negative indices, the length of a slice is the diﬀerence of the indices, if both are within bounds. For example, the length of word[1:3] is 2.