Basics of Python

Contents

- Values and Types
- Numbers
- Strings
- Keyboard Input
- Indentation and Line Joining

Values and Types

- Values belong to different types

```
>>> type('Hello, World!')
<type 'str'>
>>> type(17)
<type 'int'>
>>> type(3.2)
<type 'float'>
>>> type('17')
<type 'str'>
>>> type((1, 0, 0))
>>> <type 'tuple'>
>>> 1,000,000
(1, 0, 0)
```

  – Python interprets 1,000,000 as a sequence of integers separated by commas → an example of a semantic error

Numbers

- Integers:

`2` `34` `-34`

- Floats (floating point numbers):

`34.` `3.23` `52.3E-4` (= 52.3×10^{-4})

- The built-in functions **`abs`**, **`int`**, and **`round`**:

Expression	Value	Expression	Value	Expression	Value
`abs(3)`	`3`	`int(2.7)`	`2`	`round(2.7)`	`3`
`abs(0)`	`0`	`int(3)`	`3`	`round(2.317,2)`	`2.32`
`abs(-3)`	`3`	`int(-2.7)`	`-2`	`round(2.317,1)`	`2.3`

Variables

- Variables are just parts of our computer’s memory where you store some information
  - We need some method of accessing these variables and hence we give them names
- An assignment statement creates new variable and gives them values

```
speed = 50
timeElapsed = 14
distance = speed * timeElapsed
print(distance)
```

[RUN]

```
700
```

Variables

- Naming rule:
  - Case sensitive: `myname` and `myName` are different
  - Must begin with a letter or an underscore (_)
  - The rest can be any of the alphabet, underscores, or digits
  - Descriptive variable names help others (and yourself) easily recall what the variable represents
- Examples of invalid identifier names:

```
2things
```

```
this is spaced out
```

```
my-name
```

```
>a1b2_c3
```

Variables

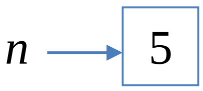
- Python's keywords cannot be used as variable names
  - Python 2 has 31 keywords

`and`	`del`	`from`	`not`	`while`
`as`	`elif`	`global`	`or`	`with`
`assert`	`else`	`if`	`pass`	`yield`
`break`	`except`	`import`	`print`	
`class`	`exec`	`in`	`raise`	
`continue`	`finally`	`is`	`return`	
`def`	`for`	`lambda`	`try`	

Variables

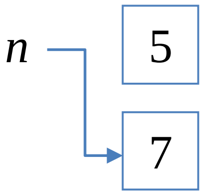
- Numeric objects in Memory:
  - Consider the following lines of code

```
n = 5
n = 7
```



✓ A portion of memory is set aside to hold 5

✓ The variable *n* is set to reference (or point to) 5 in the memory location



✓ A new memory location is set aside to hold 7

✓ The variable *n* is redirected to point to the new memory location

  - The number 5 in memory is said to be orphaned or abandoned
  - Python will eventually remove the orphaned number from memory with a process called garbage collection

Strings

- Strings are immutable
  - Cannot be changed once created
- A string is a sequence of characters
  - Single quotes: `'Quote me on this'`
  - Double quotes: `"What’s your name?"`
  - Triple quotes (`"""` or `'''`) can be used to specify multi-line strings in which single quotes and double quotes can be used freely

Strings

```
>>> x = '''This is the first line.
This is the second line.
"What's your name?," I asked.
He said "Bond, James Bond."'''
>>> x
'This is the first line.\nThis is the second line.\n"What\'s
your name?," I asked.\nHe said "Bond, James Bond."'
>>> print(x)
This is the first line.
This is the second line.
"What's your name?," I asked.
He said "Bond, James Bond."
```

Concatenation and Repetition

- Two strings can be concatenated by using the **+** operator
- A string can be repeatedly concatenated by using the ***** operator

```
>>> 'good' + 'bye'
'goodbye'
>>> ('a' + 'b') * 3
'ababab'
>>> 'a' * 3 + 'b' * 3
'aaabbb'
>>> 'ha' * 3
'hahaha'
>>> ("cha-" * 2) + "cha"
'cha-cha-cha'
```

- When a string expression appears in an assignment statement or a print statement, the string expression is evaluated before being assigned or displayed

String Functions and Methods

```
>>> len('string')
6
>>> int('23')
23
>>> float('23')
23.0
>>> eval('23')
23
>>> eval('23.5')
23.5
>>> x = 5
>>> eval('23 + (2 * x)')
33
```

– The **`eval`** function evaluates the expression to an integer or floating-point number as appropriate

String Functions and Methods

- The **`exec`** function takes a string consisting of Python code and executes it

```
>>> exec('x = 2')
>>> x
2
>>> exec('y = 3')
>>> y
3
>>> eval('x + y')
5
>>> exec('x + y')
>>>
```

String Functions and Methods

- The **int** and **float** functions can also be applied to numeric expressions

Example	Value	Example	Value
int(4.8)	4	float(4.67)	4.67
int(-4.8)	-4	float(-4)	-4.0
int(4)	4	float(0)	0.0

- The **str** function converts a number to its string representation

```
>>> str(5.6)
'5.6'
>>> str(5)
'5'
>>> str(5.)
'5.0'
>>> x = 10
>>> str(x) + '%'
'10%'
```

String Functions and Methods

- A string method is a process that performs a task on a string
  - The general form of an expression applying a method is

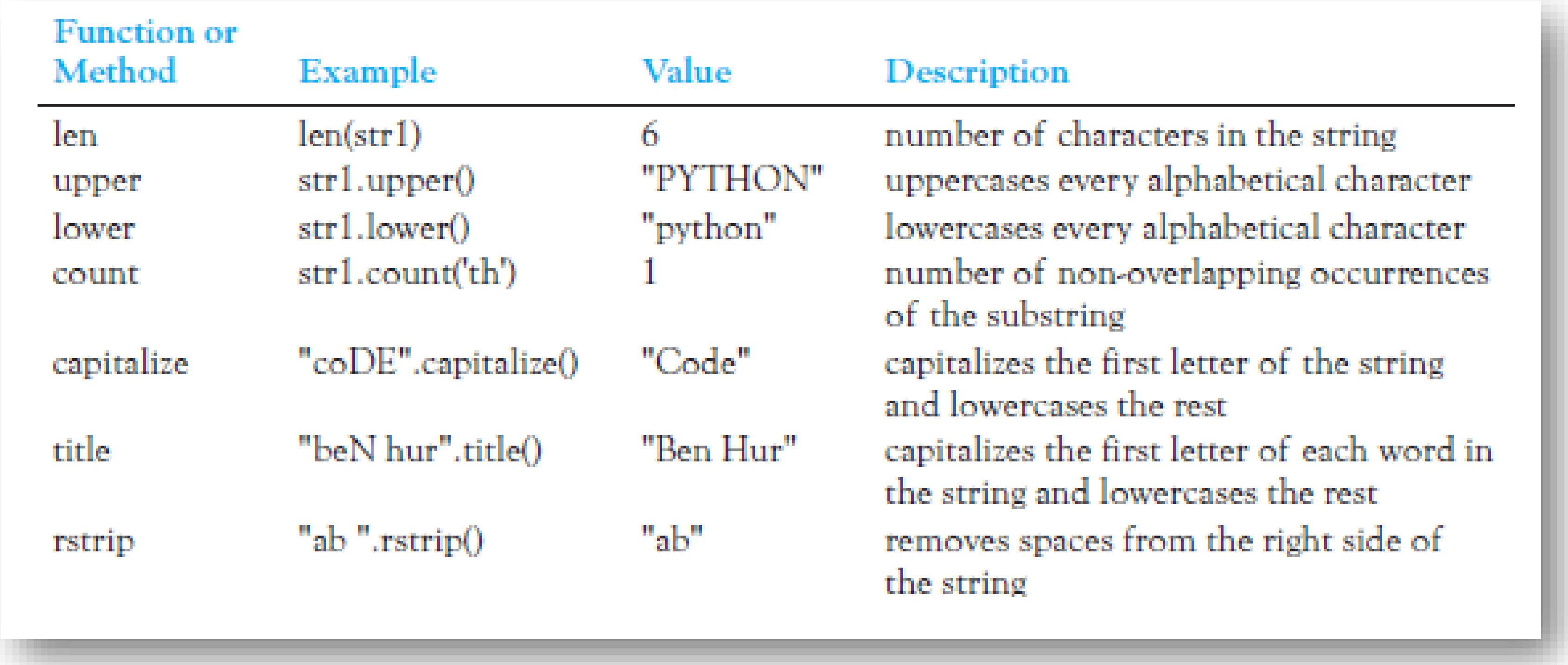
    `stringName.methodName()`

    where the parentheses might contain values

```
>>> str1 = "Python"
>>> str1.upper()
'PYTHON'
>>> str1.lower()
'python'
>>> str1.count('th')
1
>>> 'coDE'.capitalize()
'Code'
>>> "beN hur".title()
'Ben Hur'
>>> 'ab   '.rstrip()  # removes spaces from the right side
'ab'
```

String Functions and Methods

- String operations (`str1 = "Python"`)



Function or Method	Example	Value	Description
len	len(str1)	6	number of characters in the string
upper	str1.upper()	"PYTHON"	uppercases every alphabetical character
lower	str1.lower()	"python"	lowercases every alphabetical character
count	str1.count('th')	1	number of non-overlapping occurrences of the substring
capitalize	"coDE".capitalize()	"Code"	capitalizes the first letter of the string and lowercases the rest
title	"beN hur".title()	"Ben Hur"	capitalizes the first letter of each word in the string and lowercases the rest
rstrip	"ab ".rstrip()	"ab"	removes spaces from the right side of the string

String Functions and Methods

- Chained methods:

```
>>> praise = "Good Doggie".upper()
>>> numberOfGees = praise.count('G')
>>> print(numberOfGees)
3
```

  - These two lines can be combined into a single line by chaining the two methods

```
>>> numberOfGees = "Good Doggie".upper().count('G')
>>> print(numberOfGees)
3
```

  - Chained methods are executed from left to right
  - Chaining often produces clearer code since it eliminates temporary variables, such as the variable **`praise`** above

Indices and Slices

- If `str1` is a string variable, then `str1[i]` is the character of the string having index *i* (the index starts from 0)
- A slice of a string is a sequence of consecutive characters from the string
  - `str1[m:n]` is the substring beginning at position *m* and ending at position *n* – 1
  - `str1[m:n]` will be the empty string (`""`) if *m* ≥ *n*
- Given another string `subStr`, the methods `str1.find(subStr)` and `str1.rfind(subStr)` return the positive index from the left and right, respectively, of the first appearance of `subStr` in `str1`
  - –1 is returned if `subStr` does not appear in `str1`

Indices and Slices

```
print('Python')
print('Python'[1], 'Python'[5], 'Python'[2:4])
str1 = 'Hello World!'
print(str1.find('W'))
print(str1.find('x'))
print(str1.rfind('l'))     # finds the rightmost ‘l’
```

[RUN]

```
Python
y n th
6
-1
9
```

Indices and Slices

- Python allows strings to be indexed by their position from the right side of the string by using negative numbers for indices

```
print('Python')
print('Python'[-1], 'Python'[-4], 'Python'[-5:-2])
str1 = 'spam & eggs'
print(str1[-2])
print(str1[-8:-3])
print(str1[0:-1])
```

[RUN]

```
Python
n t yth
g
m & e
spam & egg
```

Indices and Slices

- One or both of the bounds in **`str1[m:n]`** can be omitted
  - *m* defaults to 0
  - *n* defaults to the length of the string

```
print('Python'[2:], 'Python'[:4], 'Python'[:])
print('Python'[-3:], 'Python'[:-3])
```

[RUN]

```
thon Pyth Python
hon Pyt
```

Optional print Arguments

- We can optionally change the separator with **`sep`** argument:

```
>>> x = 5; y = 7
>>> print(x, y, sep='*')
5*7
>>> print("Hello", "World", sep="")
HelloWorld
>>> print('1', 'two', 3, sep='   ')
1   two   3
```

- **`print`** always ends with an invisible special character "new line" (**`\n`** or **`₩n`**) so that repeated calls to print will all appear on a separate new line each
- We can optionally change the ending operation with **`end`** argument:

```
print("Hello", end=" ")
print("World")
```

[RUN]

```
Hello World
```

The format Method

- Strings can be constructed from other information

```
age = 20
name = 'Swaroop'

print('''{0} was {1} years old when he wrote the book
A Byte of Python.'''.format(name, age))
print('Why is {0} playing with that python?'.format(name))
```

[RUN]

```
Swaroop was 20 years old when he wrote the book
A Byte of Python.
Why is Swaroop playing with that Python?
```

– Python substitutes each of the format argument value into the place of the corresponding specification in the string

– Note that Python starts counting from 0

The format Method

- Also note that the numbers in the specifications are optional
  - The following code gives exactly the same output as the previous code

```
age = 20
name = 'Swaroop'

print('''{} was {} years old when he wrote the book
A Byte of Python.'''.format(name, age))
print('Why is {} playing with that python?'.format(name))
```

[RUN]

```
Swaroop was 20 years old when he wrote the book
A Byte of Python.
Why is Swaroop playing with that Python?
```

The format Method

- The symbols <, ^, and > that precede the width of each field instruct the print function to left-justify, center, and right-justify, respectively

```
## Demonstrate justification of output.
print("0123456789012345678901234567")
print("{0:^5}{1:<20}{2:>3}".format("Rank", "Player", "HR"))
print("{0:^5}{1:<20}{2:>3}".format(1, "Barry Bonds", 762))
print("{0:^5}{1:<20}{2:>3}".format(2, "Hank Aaron", 755))
print("{0:^5}{1:<20}{2:>3}".format(3, "Babe Ruth", 714))
```

[RUN]

```
0123456789012345678901234567
Rank Player               HR
  1  Barry Bonds         762
  2  Hank Aaron          755
  3  Babe Ruth           714
```

  - When none of the symbols <, ^, or > are present, the number (string) will be displayed left-justified (right-justified) by default

The format Method

- `f` and `%` are used after the field-width number to display a floating-point number or a number in percentages, respectively
  - They should be preceded by a period and a number indicating the decimal precision
  - A comma can be inserted after the field-width number if we want thousands separators

Statement	Outcome
`print('{0:10.2f}'.format(1234.5678))`	`1234.57`
`print('{0:10,.2f}'.format(1234.5678))`	`1,234.57`
`print('{0:10,.3f}'.format(1234.5678))`	`1,234.568`
`print('{0:10,.2%}'.format(1234.5678))`	`123,456.78%`
`print('{0:10,.3%}'.format(1234.5678))`	`123,456.780%`
`print('{0:10,}'.format(12345678))`	`12,345,678`

The format Method

- More on the format method

```
# decimal (.) precision of 3 for a float
print('{0:.3f}'.format(1.0/3))
# fill with underscores (_) with the text centered (^)
# to the width of 11
print('{0:_^11}'.format('hello'))
# keyword-based specifications
print('{name} wrote {book}'.format(name = 'Swaroop',
                                   book = 'A Byte of Python'))
```

[RUN]

```
0.333
___hello___
Swaroop wrote A Byte of Python
```

  - Note: `1.0/3` is a float division

    `1/3` is an integer division resulting in `0`

Escape Sequences

- How can you specify a string that has a single quote in it?

```
>>> print('What's your name?')

SyntaxError: invalid syntax
```

```
print("What's your name?")
print('What\'s your name?')
print('He said, "Bond, James Bond."')
print("He said, \"Bond, James Bond.\"")
```

[RUN]

```
What's your name?
What's your name?
He said, "Bond, James Bond."
He said, "Bond, James Bond."
```

* '\' actually appears as '₩' in Python windows

Escape Sequences

- Escape sequences are short sequences that are placed in strings to permit some special characters to be printed
  - The first character is always a backslash (\)

- A backslash itself can be specified by using an additional backslash

```
print('How can you prevent \\n from being printed?')
print('How can you prevent \n from being printed?')
print('A backslash at the end of the line \
indicates line continuation')
```

[RUN]

```
How can you prevent \n from being printed?
How can you prevent
 from being printed?
A backslash at the end of the line indicates line continuation
```

Escape Sequences

- To specify some strings where no special processing such as escape sequences are handled
  - Specify a raw string by prefixing **`r`** or **`R`** to the string

```
print("New lines are indicated by \n.")
print(r"New lines are indicated by \n.")

[RUN]

New lines are indicated by
.
New lines are indicated by \n.
```

Keyboard Input

- When a built-in function called **`input`** is called, the program stops and waits for the user to type something
  - When the user presses *Enter*, the program resumes and **`input`** returns what the user typed as a string

```
>>> text = input()
What are you waiting for?
>>> print(text)
What are you waiting for?
```

- If you want to print a prompt telling the user what to input, you can give the prompt to **`input`** as an argument

```
>>> name = input('What is your name? ')
What is your name? Allen
>>> print(name)
Allen
>>> name
'Allen'
```

Keyboard Input

- If you expect the user to type an integer, you can try to convert the return value to **`int`**

```
>>> prompt = 'What is the airspeed velocity of a swallow?\n'
>>> speed = int(input(prompt))
What is the airspeed velocity of a swallow?
17
>>> speed
17
```

  – The user’s input appears below the prompt because the new line character **`\n`** at the end of the prompt causes a line break

Keyboard Input

```
fullName = input('Enter a full name: ')
n = fullName.rfind(' ')
print('Last name:', fullName[n+1:])
print('First name(s):', fullName[:n])
```

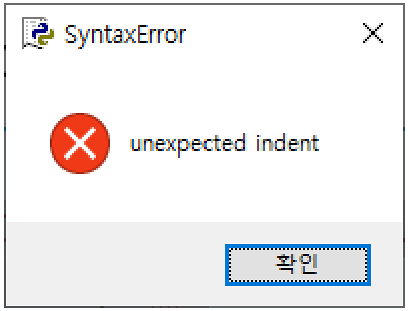
[RUN]

```
Enter a full name: Franklin Delano Roosevelt
Last name: Roosevelt
First name(s): Franklin Delano
```

Indentation and Line Joining

- Indentation is semantically meaningful in Python
  - Statements that go together (called a block) must have the same indentation
  - An indentation must have four spaces
- Wrong indentation gives rise to errors

```
i = 5
# Error below! Notice a single space at the start of the line
 print 'Value is ', i
print 'I repeat, the value is ', i
```



unexpected indent

Indentation and Line Joining

- Explicit line joining
  - A long logical line can be broken down to multiple physical lines by using the backslash

```
print('The area of {0} is {1:,} square miles.'
      .format('Texas', 268820))
str1 = 'The population of {0} is {1:.2%} of \ # continue to
the U.S. population.`                         # the next line
print(str1.format('Texas', 26448000. / 309000000))
```

[RUN]

```
The area of Texas is 268,820 square miles.
The population of Texas is 8.56% of the U.S. population.
```

Indentation and Line Joining

- Implicit line joining
  - Backslash is not needed when the logical line has a starting parentheses, starting square brackets, or a starting curly braces but not an ending one

```
quotation = ('Well written code is its own ' +
             'best documentation.')
print(quotation)
```

[RUN]

```
Well written code is its own best documentation.
```