# String Format, Regular Expression

講者：Isaac

# Outline

- String Formatting
- Regular Expression

# String Formatting

# String Formatting

- The Formatter class implementation as the built-in format() method.
- Format strings contain "replacement fields" surrounded by curly braces {}.
  - Anything that is not contained in braces is considered literal text, which is copied unchanged to the output.
  - If you need to include a brace character in the literal text, it can be escaped by doubling: {{ and }}.
- Example:

```
>>> a = 100
>>> b = 8.9
>>> c = 'hello'
>>> '{} {} {}'.format(a, b, c)
'100 8.9 hello'
```

# Replacement field Syntax:

- **format_spec** ::= [[fill]align][sign][#][0][width][grouping_option][.precision][type]

| Option Name | Description & Meaning |
|---|---|
| fill | Fill with characters |
| align | "<" Forces the field to be left-aligned within the available space |
| | ">" Forces the field to be right-aligned within the available |
| | "=" Forces the padding to be placed after the sign (if any) but before the digits. This is used for printing fields in the form '+000000120'. This alignment option is only valid for numeric types. It becomes the default when '0' immediately precedes the field width. |
| | "^" Forces the field to be centered within the available space. |

# Replacement field Syntax:

- **format_spec** ::= [[fill]align][sign][#][0][width][grouping_option][.precision][type]

| Option Name | Description & Meaning |
|---|---|
| sign | "+" indicates that a sign should be used for both positive as well as negative numbers. |
| | "-" indicates that a sign should be used only for negative numbers (this is the default behavior). |
| | " " indicates that a leading space should be used on positive numbers, and a minus sign on negative numbers. |
| "#" | Add prefix respective "0b" to binary integers. |
| | Add prefix respective "0o" to octal integers. |
| | Add prefix respective "0x" to hexadecimal integers. |
| 0 | Add zeros to prefix. |

# Replacement field Syntax:

- **format_spec** ::= [[fill]align][sign][#][0][width][grouping_option][.precision][type]

| Option Name | Description & Meaning |
|---|---|
| width | a decimal integer defining the minimum total field width, including any prefixes, separators, and other formatting characters. If not specified, then the field width will be determined by the content. |
| grouping_option | “,” signals the use of a comma for a thousands separator. |
| | “_” signals the use of an underscore for a thousands separator for floating point presentation types and for integer presentation type 'd'. For integer presentation types 'b', 'o', 'x', and 'X', underscores will be inserted every 4 digits. |
| .precision | a decimal number indicating how many digits should be displayed after the decimal point for a floating point value formatted with 'f' and 'F', or before and after the decimal point for a floating point value formatted with 'g' or 'G'. |

# Replacement field Syntax:

- **format_spec** ::= [[fill]align][sign][#][0][width][grouping_option][.precision][type]
  - type: determines how the data should be presented.
  - String presentation types:

| Type | Meaning |
|---|---|
| 's' | String format. This is the default type for strings and may be omitted. |
| Name | The same as "s". |

  - Integer presentation types:

| Type | Meaning |
|---|---|
| 'b' | Binary format. Outputs the number in base 2. |
| 'd' | Decimal Integer. Outputs the number in base 10. |

# Replacement field Syntax:

- Integer presentation types:

| Type | Meaning |
|---|---|
| 'o' | Octal format. Outputs the number in base 8. |
| 'x' | Hex format. Outputs the number in base 16, using lower-case letters for the digits above 9. |
| 'X' | Hex format. Outputs the number in base 16, using upper-case letters for the digits above 9. |
| 'n' | Number. This is the same as 'd', except that it uses the current locale setting to insert the appropriate number separator characters. |
| 'c' | Character. Converts the integer to the corresponding unicode character before printing. |
| None | The same as 'd'. |

# Replacement field Syntax:

- Floating point presentation types:

| Type | Meaning |
|---|---|
| 'e' | Exponent notation. Prints the number in scientific notation using the letter ‘e’ to indicate the exponent. The default precision is 6. |
| 'E' | Exponent notation. Same as 'e' except it uses an upper case ‘E’ as the separator character. |
| 'f' | Fixed-point notation. Displays the number as a fixed-point number. The default precision is 6. |
| 'F' | Fixed-point notation. Same as 'f', but converts nan to NAN and inf to INF. |
| '%' | Percentage. Multiplies the number by 100 and displays in fixed ('f') format, followed by a percent sign. |

# Replacement field Syntax:

- Floating point presentation types:

| Type | Meaning |
|---|---|
| 'g' | General format. For a given precision p >= 1, this rounds the number to p significant digits and then formats the result in either fixed-point format or in scientific notation, depending on its magnitude.<br>The default precision is 6. |
| 'G' | General format. Same as 'g' except switches to 'E' if the number gets too large. The representations of infinity and NaN are uppercased, too. |
| 'n' | Number. This is the same as 'g', except that it uses the current locale setting to insert the appropriate number separator characters. |
| None | Similar to 'g', except that fixed-point notation, when used, has at least one digit past the decimal point. The default precision is as high as needed to represent the particular value. The overall effect is to match the output of str() as altered by the other format modifiers. |

# Formatting Examples

- Accessing arguments by position:

```
>>> a = 100
>>> b = 8.9
>>> c = 'hello'
```

```
>>> '{} {} {}'.format(a, b, c)
'100 8.9 hello'
```

```
>>> '{0} {1} {2}'.format(a, b, c)
'100 8.9 hello'
>>> '{2} {1} {0}'.format(a, b, c)
'hello 8.9 100'
```

# Formatting Examples

- Accessing arguments by name:

```
>>> 'Coordinates: {latitude}, {longitude}'.format(latitude='25.019N', longitude='121.54E')
'Coordinates: 25.019N, 121.54E'
>>> coord = {'latitude': '25.019N', 'longitude': '121.54E'}
>>> 'Coordinates: {latitude}, {longitude}'.format(**coord)
'Coordinates: 25.019N, 121.54E'
```

- Accessing arguments' attributes:

```
>>> ('the complex number {0} contains real part {0.real} and imaginary part {0.imag}').format(a)
'the complex number (1+2j) contains real part 1.0 and imaginary part 2.0'
```

- Accessing arguments' items:

```
>>> coord = (8, 9)
>>> 'X: {0[0]}; Y: {0[1]}'.format(coord)
'X: 8; Y: 9'
```

# Formatting Examples

- Aligning: a=100, b=8.9, c='hello'

Set width to 10

```
>>> '{0:10d} {1:10f} {2:10s}'.format(a, b, c)
'       100   8.900000 hello     '
```

Align to the left  Float format

```
>>> '{0:>10d} {1:>10f} {2:>10s}'.format(a, b, c)
'       100   8.900000      hello'
```

Align to the right  String format

```
>>> '{0:<10d} {1:<10f} {2:<10s}'.format(a, b, c)
'100        8.900000   hello     '
```

Integer format  Align to center

```
>>> '{0:^10d} {1:^10f} {2:^10s}'.format(a, b, c)
'   100     8.900000    hello    '
```

# Formatting Examples

- Converting to different bases :

```
>>> 'int: {0:d}; bin: {0:b}; oct: {0:o}; hex: {0:x}'.format(99)
'int: 99; bin: 1100011; oct: 143; hex: 63'
```

**Add 0x/0b/0o as prefixes**

```
>>> 'int: {0:#d}; bin: {0:#b}; oct: {0:#o}; hex: {0:#x}'.format(99)
'int: 99; bin: 0b1100011; oct: 0o143; hex: 0x63'
```

- Using the comma as thousand separator

```
>>> '{:,}'.format(1234567890)
'1,234,567,890'
```

# Formatting Examples

- Showing the floating points after decimal numbers:

```
>>> a = 33
>>> b = 19
>>> 'percentage = {:.2%}'.format(b/a)
'percentage = 57.58%'
```

- Type-Specific formatting:

```
>>> import datetime
>>> d = datetime.datetime(2020, 1, 1, 12, 33, 43)
>>> '{:%Y-%m-%d %H:%M:%S}'.format(d)
'2020-01-01 12:33:43'
```

# Regular Expression

# Regular Expression

- A regular expression (or RegEx/RE) specifies a set of strings that matches it; the functions in this module let you check if a particular string matches a given regular expression.
- Syntax Example:
  - matchObject = re.search(pattern, string)

```
>>> import re
>>> txt = "Go Go PowerRanger!"
>>> x = re.search('Go', txt)
>>> x
<re.Match object; span=(0, 2), match='Go'>
>>> type(x)
<class 're.Match'>
```

# RegEx Syntax

- Metacharacter

| Char | Description |
|---|---|
| ? | One character or none |
| * | Arbitrary number of character or none |
| + | At least one or more characters |
| {N} | N characters |
| {N, } | At least N characters |
| {N, M} | At least N characters to at most M characters |
| \ | Escape char |
| . | Any characters |
| […] | Any characters included of the string. |
| [^…] | Any characters excluded of the string. |
| ^… | Start with specify character |
| …$ | End with specify character |
| […] | a set of characters you wish to match |

# RegEx Syntax

- Metacharacter

| Char | Description |
|---|---|
| \d | Digits = [0-9] |
| \D | Non-digits = [^0-9] |
| \w | Digits, characters, underline = [a-zA-Z0-9_] |
| \W | Not \w, = [^a-zA-Z0-9_] |
| \s | Space character = [\f\n\r\t\v] |
| \S | Not space character = [^ \f\n\r\t\v] |
| \f | Form-feed |
| \n | newline |
| \r | Carriage return |
| \t | tab |
| \v | Vertical tab |

# RegEx Syntax

- Metacharacter Examples:
  - Square brackets [ ] specifies a set of characters.

| Expression | Input String | Matches |
|---|---|---|
| [abc] | a | 1 match |
| | bc | 2 matches |
| | Hello | No matches |
| | Abc wx bo | 4 matches |

# RegEx Syntax

- Metacharacter Examples:
  - A period . matches any single character (except newline '\n').

| Expression | Input String | Matches |
|---|---|---|
| .. | a | No match |
| | bc | 1 match |
| | Hello | 2 matches |
| | Abc wx bo | 3 matches |

# RegEx Syntax

- Metacharacter Examples:
  - The caret symbol ^ is used to check if a string **starts with** a certain character.

| Expression | Input String | Matches |
|---|---|---|
| ^a | a | 1 match |
| | bc | No match |
| | Halo | No match |
| | abc ax bo | 2 matches |

# RegEx Syntax

- Metacharacter Examples:
  - The star symbol * matches **zero or more occurrences** of the pattern left to it.

| Expression | Input String | Matches |
|---|---|---|
| ma*n | mn | No match |
| | man | 1 match |
| | maan | 1 match |
| | main | No match |

# RegEx Syntax

- Metacharacter Examples:
  - The plus symbol + matches **one or more occurrences** of the pattern left to it.

| Expression | Input String | Matches |
|---|---|---|
| ma+n | mn | No match |
| | man | 1 match |
| | maan | 1 match |
| | main | No match |

# RegEx Syntax

- Metacharacter Examples:
  - The question mark symbol ? matches **zero or one occurrence** of the pattern left to it.

| Expression | Input String | Matches |
|---|---|---|
| ma?n | mn | No match |
| | man | 1 match |
| | maan | 1 match |
| | main | No match |

# RegEx Syntax

- Metacharacter Examples:
  - {} Consider this code: {n,m}. This means at least n, and at most m repetitions of the pattern left to it.

| Expression | Input String | Matches |
|---|---|---|
| a{2,4} | abcda | No match |
| | abcdaa | 1 match |
| | abcdaaa | 1 match |
| [0-9]{2,3} | ab123cde | 1 match |
| | 1234a345 | 2 matches |

# RegEx Syntax

- Metacharacter Examples:
  - Vertical bar | is used for alternation (or operator).
  - Parentheses () is used to group sub-patterns.

| Expression | Input String | Matches |
|---|---|---|
| a\|b | abc | 2 matches |
| | cde | No match |
| | abcde | 2 matches |
| (a\|b)c | abc | 1 match |
| | cde | No match |
| | abcde | 1 match |

# Python3 re module

- Python has a module named re to work with regular expressions.
- Use *import re* to load module.
- The module defines several functions and constants to work with RegEx.
  - match()
  - search()
  - findall()

```
>>> import re
>>> txt = "Go Go PowerRanger!"
>>> x = re.search('Go', txt)
>>> x
<re.Match object; span=(0, 2), match='Go'>
>>> type(x)
<class 're.Match'>
```

# Python3 re module Methods

- re.match()
  - two arguments: a pattern and a string
  - start matching with the first character
  - successful match: return a match object
  - Otherwise: return None
- Example:

```
>>> import re
>>> txt = 'hello, hey, hi, Salaheyo'
>>> x = re.match('\w*', txt)
>>> x
<re.Match object; span=(0, 5), match='hello'>
>>> x = re.match('\w{2}', txt)
>>> x
<re.Match object; span=(0, 2), match='he'>
>>> x = re.match('\w{6}', txt)
>>> x None
```

Different match

# Example

- re.match()
  - Phone number

```
:
```

```python
import re

phonenumber = "02-222-1234, 03-333-1234"
regex = "(\w{2})-\w{3}-\w{4}"
match_obj = re.match(regex, phonenumber)

if match_obj:
    print(match_obj)
    print("Valid phone number")
else:
    print("Invalid phone number")
```

```
<re.Match object; span=(0, 11), match='02-222-1234'>
Valid phone number
```

```
:
```

```python
import re

phonenumber = "(02)-222-1234, 03-333-1234"
regex = "(\w{2})-\w{3}-\w{4}"
match_obj = re.match(regex, phonenumber)

if match_obj:
    print(match_obj)
    print("Valid phone number")
else:
    print("Invalid phone number")
```

```
Invalid phone number
```

# Python3 re module Methods

- re.search()
  - two arguments: a pattern and a string.
  - looks for the first location matches the RegEx pattern
  - successful search: returns a match object
  - if not, it returns None.
- Example:

```
>>> import re
>>> txt = "Go Go PowerRanger!"
>>> x = re.search('Go', txt)
>>> x
<re.Match object; span=(0, 2), match='Go'>
>>> type(x)
<class 're.Match'>
```

pattern string

# Example

- re.search()
  - Phone Number

```
:
```

```python
import re

phonenumber = "(02)-222-1234, 03-333-1234"
regex = "\w{2}-\w{3}-\w{4}"
search_obj = re.search(regex, phonenumber)

if search_obj:
    print(search_obj)
    print("Valid phone number")
else:
    print("Invalid phone number")
```

```
<re.Match object; span=(15, 26), match='03-333-1234'>
Valid phone number
```

# Python3 re module Methods

- re.findall()
  - returns a list of strings containing all matches.
- Example:

```
>>> import re
>>> txt = 'hello! 23 hi 11. how are 889.'
>>> pattern = '\d+'
>>> x = re.findall(pattern, txt)
>>> x
['23', '11', '889']
```

# Example

- re.findall()
  - Phone number

```
:  1  import re
   2
   3  phonenumber = "02-222-1234, 03-333-1234"
   4  regex = r"(\d{2}-\d{3}-\d{4})"
   5  obj = re.findall(regex, phonenumber)
   6
   7  if obj:
   8      print(obj)
   9      print("Valid phone number")
  10  else:
  11      print("Invalid phone number")
```

```
['02-222-1234', '03-333-1234']
Valid phone number
```

# Example

- re.findall()
  - ID

```
1  import re
2
3  id_card = "A123456789, B223456789, C333456789"
4  regex = r"(\w{1}[1|2]\d{8})"
5  obj = re.findall(regex, id_card)
6
7  if obj:
8      print(obj)
9      print("Valid id number")
10 else:
11     print("Invalid id number")
```

```
['A123456789', 'B223456789']
Valid id number
```

# RegEx Examples

- How modify the following example to include cat?

```
:
1 import re
2 animals = '1.cat, 2 dog, 03 pig, 4 duck'
3 regex = r"\d+\s\w+"
4 obj = re.findall(regex, animals)
5
6 if obj:
7     print(obj)
```

```
['2 dog', '03 pig', '4 duck']
```

- Answer

```
1 import re
2 animals = '1.cat, 2 dog, 03 pig, 4 duck'
3 regex = r"\d+\.?\s?\w+"
4 obj = re.findall(regex, animals)
5
6 if obj:
7     print(obj)
```

```
['1.cat', '2 dog', '03 pig', '4 duck']
```