The Python Language

Tran Giang Son, tran-giang.son@usth.edu.vn

ICT Department, USTH

Expressions

Interactive vs Script

- Interactive
  - Type command
  - Execute
  - Wait for response
- Script
  - All-in-one long sequences of statements
  - `python script.py`
  - Shebang `#!` works

Constants

- What
  - Fixed values
  - Value does not change over time
- Examples
  - Numeric constants
  - String constants
    - Single quotes '
    - Double quotes "
- Why: everywhere

Constants

- How

```
>>> print(123)
123
>>> print(98.6)
98.6
>>> print('Hello world')
Hello world
```

Variables

- What
  - Named place in the memory to store data
  - Access it later using name
  - Modifiable at runtime
- Why: store temporary changable values

Variables

- Variable name rules
  - Letters, numbers, or underscores
  - CaSe sEnSiTiVe
  - Not allowed: starting with number
- Examples
  - Good: `spam`, `eggs`, `spam23`, `_speed`
  - Bad: `23spam`, `#sign`, `var.12`
  - Different: `spam`, `Spam`, `SPAM`

Variables

- Reserved words

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

Statements

- What: combination of operator and its operand(s)
  - Operator: symbol indicating a calculation
  - One or more operands
- Numeric expression
  - + Addition
  - - Subtraction
  - * Multiplication
  - / Division
  - ** Power
  - % Remainder

Statements

- Numeric expression

```
>>> x = 2
>>> x = x + 2
>>> print(x)
4
>>> y = 440 * 12
>>> print(y)
5280
>>> z = y / 1000
>>> print(z)
5
```

```
>>> j = 23
>>> k = j % 5
>>> print(k)
3
>>> print(4 ** 3)
64
```

Statements

- Mixing Integer and Floats: convert everything to float.

```
>>> print(99 / 100)
0
>>> print(99 / 100.0)
0.99
>>> print(99.0 / 100)
0.99
>>> print(1 + 2 * 3 / 4.0 - 5)
-2.5
>>>
```

Data Types

What

- Variables, literals, and constants have a “data type”

Type	Examples
Integer	0, 12, 5, -5
Float	4.5, 3.99, 0.1
String	“Hi”, “Hello”, “Hi there!”"
Boolean	True, False
List	[“hi”, “there”, “you”]
Tuple	(4, 2, 7, 3)

What: Boolean

- `bool`
- 2 possible values: `True`, `False`

What: Integer

- `int`
- Unbounded.

```
>>> i=10**100
>>> type(i)
<class 'int'>
>>> i
1000000000000000000000000000000000000000000000000000000000
```

What: Float

- `float`
- Digits and Exponents

```
>>> 2.5
>>> 2e4
>>> 0.00001
>>> 1000020000300004
```

What: Strings

- `str`
- Series of Unicode characters
- Character: String of length 1
- Enclosed by a pair of single or double quotes
- Multiline: triple quote
  - `'''`
  - `"""`

```
>>> s="""This is
... a Multiline string
... for example"""
>>> s
'This is\na Multiline string\nfor example'
```

Dynamically typing

- Dynamically typed variables
- Types are automatically managed

C, Java

```
int a;
float b;
a = 5;
b = 0.43;
```

Python

```
a = 5
a = 0.43
a = "Hello"
```

Number Conversion

- int()
- float()

```
>>> print(float(99) / 100)
0.99
>>> i = 42
>>> type(i)
<class 'int'>
>>> f = float(i)
>>> print(f)
42.0
>>> type(f)
<class 'float'>
>>> print(1 + 2 * float(3) / 4 - 5)
-2.5
>>>
```

Number Conversion

- Also works with strings!

```
>>> sval = '123'
>>> type(sval)
<class 'str'>
>>> print(sval + 1)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module
TypeError: can only concatenate str
>>> ival = int(sval)
>>> type(ival)
<class 'int'>
>>> print(ival + 1)
124
>>> nsv = 'hello bob'
>>> niv = int(nsv)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module
```

String Operators

- Some operators apply to strings
  - `+` concatenation
  - `*` multiple concatenation
  - `in`, `not in` contains/not contains

```
>>> print('abc' + '123')
abc123
>>> print('Hi' * 5)
HiHiHiHiHi
>>> "US" in "AmongUS"
True
>>> "us" not in "AmongUS"
True
```

String Operators

- Substring: `string[index:end:step]`
- `index`, `end`
  - `>=0`: start from beginning of string
  - `<`: start from end of string
  - Can be omitted

```
+---+---+---+---+---+---+
| P | y | t | h | o | n |
+---+---+---+---+---+---+
| 0 | 1 | 2 | 3 | 4 | 5 |
+---+---+---+---+---+---+
|-6 |-5 |-4 |-3 |-2 |-1 |
+---+---+---+---+---+---+
```

- `step`: How many letters to skip

String Operators

- `string[index:end:step]`

```
>>> s = "Advanced Programming with Python"
```

```
>>> s[9]
'P'
>>> s[9:20]
'Programming'
>>> s[9:20:2]
'Pormig'
```

```
>>> s[:20]
'Advanced Programming'
>>> s[9:]
'Programming with Python'
>>> s[-6:-4]
'Py'
>>> s[-6:]
'Python'
```

String Formats

- Similar to C's `printf()`
- Previously, in pre-3.6 Python

```
print("Greeting, {}. You are {}".format(name, age))
```

- From Python 3.6 onward: f-string, or formatted string literals

```
print(f"Greeting, {name}. You are {age}")
```

Comments

- What? # starts a line comment
- Why?
  - Description of code block
  - Document some extra info
  - Turn off a line of code

Comments

```
>>> s = "USTH"
>>> # print("nobody cares")
>>> print(s)
USTH
```

Conditions

Indentation Rules

- *Increase* indent after an if statement or for statement (after :)
  - Equivalent to C, Java's `{`
- *Maintain* indent to indicate the scope of the block
  - Which lines are affected by the if/for
- *Reduce* indent to *back* to the level of the if statement or for statement to indicate the end of the block
  - Equivalent to C, Java's `}`
- Blank lines are ignored - they do not affect indentation
- Comments on a line by themselves are ignored w.r.t. indentation

Indentation Rules

- Python cares a *lot* about how far line is indented
- Don’t mix tabs and spaces
  - “indentation errors” even if everything *looks* fine
- Use one only
  - Most text editors can turn tabs into spaces - make sure to enable this feature

if - else

```
x = 5
if x < 10:
    print('Smaller than 10')
else:
    print('Bigger than 10')
print('End')
```

Nested if - else

```
x = 5
if x < 10:
    print('Smaller than 10')
    if x > 5:
        print('  Still bigger than 5')
else:
    print('Bigger than 10')
print('End')
```

if - else - if - else

```
x = 21
if x < 10:
    print('Smaller than 10')
elif x < 20:
    print('Smaller than 20')
else:
    print('Bigger than 20')
print('End')
```

Functions

What & Why

- Group of related statements performing a specific task
- Break programs into small chunks
- Better code organization
- Code reusable

How

- Definition
  - Function Name
  - Parentheses
  - Arguments

```
def function_name(arguments):
    """docstring"""
    statement1
    statement2
    ...
```

- Call

```
function_name("a value")
```

Examples

```
def greet(name):
    """
    This function greets to
    the person passed in as
    a parameter
    """
    print("Hello, " + name + ". Good morning!")

greet("Emmanuel Macron")
```

Examples

- `len(arg)`: number of elements in `arg`
- `print(args)`: write `args` to `stdout`
- `input(prompt)`: `print(prompt)`, wait and read user input from `stdin`, return the entered string

Collections

What

- Multiple objects are grouped together
- Main types
  - Sets
  - Sequences
  - Maps
  - Streams

Set

- Unordered collection of items
- No duplication
- Operators
  - `s1.isdisjoint(s2)`: no common element
  - `s1 <= s2`, `s1.issubset(s2)`: $s1 \subseteq s2$
  - `s1 >= s2`, `s1.issuperset(s2)`: $s1 \supseteq s2$
  - `s3 = s1 | s2`, `s3 = s1.union(s2)`: $s3 = s1 \cup s2$
  - `s3 = s1 & s2`, `s3 = s1.intersection(s2)`: $s3 = s1 \cap s2$
  - `s3 = s1 - s2`, `s3 = s1.difference(s2)`: $s3 = s1 \setminus s2$

Sequences

- Ordered collection of items
- Can have duplications
- Positioned access
- Slicing similar to strings
  - `seq[start:end:step]`
- Implementations
  - `list`
  - `tuple`
  - `range`
- Others:
  - `str`

Lists

- Mutable sequence
  - Values can be changed later
- Flexible, widely used
- Comma separated declaration

```
>>> names = [ "ICT", "ict" ]
```

Lists

```
>>> names = [ "ICT", "ict" ]
```

Lists

```
>>> names = [ "ICT", "ict" ]
```

- `+` append elements at the end, same or `.extend()`

```
>>> names += ["Ict"]
>>> names
['ICT', 'ict', 'Ict']
```

Lists

```
>>> names = [ "ICT", "ict" ]
```

- `+` append elements at the end, same or `.extend()`

```
>>> names += ["Ict"]
>>> names
['ICT', 'ict', 'Ict']
```

- `=` replaces single value

```
>>> names[1] = "I See Tea"
>>> names
['ICT', 'I See Tea', 'Ict']
```

Lists

```
>>> names
['ICT', 'I See Tea', 'Ict']
```

Lists

```
>>> names
['ICT', 'I See Tea', 'Ict']
```

- = replaces bunch of values

```
>>> names[1:3] = [ "Icy Tea", "I See Tea" ]
>>> names
['ICT', 'Icy Tea', 'I See Tea']
```

Lists

```
>>> names
['ICT', 'I See Tea', 'Ict']
```

- `=` replaces bunch of values

```
>>> names[1:3] = [ "Icy Tea", "I See Tea" ]
>>> names
['ICT', 'Icy Tea', 'I See Tea']
```

- `+=` append elements at middle, same as `.insert()`

```
>>> names[1:1] += [ "Ice City" ]
>>> names
['ICT', 'Ice City', 'Icy Tea', 'I See Tea']
```

Lists

```
>>> names
['ICT', 'Ice City', 'Icy Tea', 'I See Tea']
```

Lists

```
>>> names
['ICT', 'Ice City', 'Icy Tea', 'I See Tea']
```

- `sort()` elements

```
>>> names.sort()
>>> names
['I See Tea', 'ICT', 'Ice City', 'Icy Tea']
```

Lists

```
>>> names
['ICT', 'Ice City', 'Icy Tea', 'I See Tea']
```

- `sort()` elements

```
>>> names.sort()
>>> names
['I See Tea', 'ICT', 'Ice City', 'Icy Tea']
```

- `del` delete elements

```
>>> del names[1]
>>> names
['I See Tea', 'Ice City', 'Icy Tea']
```

Lists

```
>>> names
['I See Tea', 'Ice City', 'Icy Tea']
```

Lists

```
>>> names
['I See Tea', 'Ice City', 'Icy Tea']
```

- .remove() occurrences

```
>>> names.remove("Icy Tea")
>>> names
['Ice City', 'I See Tea']
```

Range

- Generates a series of integers
- Very popular, widely used
- `range(end)` $ = [0..end-1]$.
- `range(start, end)` $ = [start..end-1]$.
- `range(start, end, step)` $ = {x | x = start + k * step, x < end}$

Range

```
>>> nums = range(10,15)
>>> print(nums)
range(10, 15)
>>> [x for x in nums]
[10, 11, 12, 13, 14]
```

Tuples

- Immutable sequence
- Contain any type of element.
- A very common use of tuples is a simple representation of pairs
  - Positition (x, y)
  - Size (w, h)
  - $\ldots$

Tuples

- Comma generated expression

```
>>> p = 10, 20
>>> p
(10, 20)
>>> p = (20, 40)
>>> p
(20, 40)
>>> type(p)
<class 'tuple'>
>>> p[1]=1
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'tuple' object does not support item assignm
```

Maps

- Key/value pairs
  - Key must be unique
  - Similar to JSON objects
- Unordered, mutable
- Implemented by `dict`

Maps

- Initialization

```
info = {"name": "USTH", "age": 10, \
        "depts": [ "ict", "ged"] }
```

- Key operations
  - `in`, `not in`: check key presence

    ```
    >>> "name" in info
    True
    ```

  - `max`, `min` of key

    ```
    >>> max(info)
    'name'
    ```

Maps: Operations

- Value operations
  - `d[k]`: get value by key
  - `d[k] = v`: set value to key
  - `del d[k]` remove key from dict

```
>>> info["name"]
'USTH'
>>> info["age"] = 11
>>> info["age"]
11
>>> del info["depts"]
>>> info
>>> info
{'name': 'USTH', 'age': 11
```

Maps: Methods

- `d.get(k[, default])`: same as `d[k]`, fallback to `default` if key not found
- `d.pop(k[, default])`: `del d[k]` and return previously deleted `d[k]`, fallback to `default` if key not found
- `d1.update(d2)`: for each key in `d2`, sets `d1[key]` to `d2[key]`, replacing the existing value if there was one
- `d.keys()`: returns list of keys
- `d.values()`: returns list of values
- `d.items()`: returns list of `(key,value)` tuples.

Maps: Methods

```
>>> info = {"name": "USTH", "age": 10, \
            "depts": [ "ict", "ged"] }
>>> info.get("name")
'USTH'
>>> info.get("address", "Earth")
'Earth'
>>> info.pop("depts")
['ict', 'ged']
>>> info.keys()
dict_keys(['name', 'age'])
>>> info.values()
dict_values(['USTH', 10])
>>> info.items()
dict_items([('name', 'USTH'), ('age', 10)])
```

Loops

What

- Loops (repeated steps) have iteration variables
- Iteration variable changes each time through a loop
- Often these iteration variables go through a sequence of numbers.

What

```python
n = 5
while n > 0 :
    print(n)
    n = n - 1
print('Blastoff!')
```

```
5
4
3
2
1
Blastoff!
```

break

- The break statement ends the current loop
- Jumps to the statement immediately following the loop

```
while True:
    line = input('> ')
    if line == 'done':
        break
    print(line)
print('Done!')
```

continue

- The continue statement ends the current iteration
- Jumps to the top of the loop and starts the next iteration

```
while True:
    line = input('> ')
    if line[0] == '#' :
        continue
    if line == 'done' :
        break
    print(line)
print('Done!')
```

```
> hello there
hello there
> # don't print this
> print this!
print this!
> done
Done!
```

range()

- range()
  - built-in function
  - returns sequence of numbers in a range
- Very useful in “for” loops
- 1, 2, or 3 arguments

```
x = range(5)
print(x)
[0, 1, 2, 3, 4]

x = range(3, 7)
print(x)
[3, 4, 5, 6]

x = range(10, 1, -2)
print(x)
[10, 8, 6, 4, 2]
```

range()

- for statement
  - Iterates over the members of a sequence in order
  - Executes the block each time

```
for i in <collection>
    <loop body>
```

- Examples

```
n = 5
while n > 0:
    print(n)
    n = n - 1
print('Blastoff!')
```

```
for n in range(5, 0, -1):
    print(n)
print('Blastoff!')
```

Practice!

Practical Work 0: git/github

- Fork the course's git repository to your github account
  - `https://github.com/SonTG/pp2022.git`
- Clone your forked repository to your home directory
  - `git@github.com:<YourAccount>/pp2022.git`
- Edit «README.md», write your name as instructed.
- Make a new commit with a message "First student commit"
- Push your new commit to your forked github repository

Practical work 1: student mark management

- Make a new Python program
  - Name it «1.student.mark.py»
  - Use tuples, dicts, lists, *NO* objects/classes
  - Build a student mark management system

Practical work 1: student mark management

- Functions
  - Input functions:
    - Input number of students in a class
    - Input student information: id, name, DoB
    - Input number of courses
    - Input course information: id, name
    - Select a course, input marks for student in this course
  - Listing functions:
    - List courses
    - List students
    - Show student marks for a given course
- Push your work to corresponding forked Github repository