CS61A: Python Basic

Lin Sheng

January 2022

Abstract

CS:61A course learning notes before Lab02. It contains some basic knowledge and important points about python.

1 Primitive expressions

Names evaluate to the value that they are bound to in the current environment. One way to bind a value is with an assignment statement.

2 Arithmetic expressions

- Floating point division (/): divides the first number number by the second, evaluating to a number with a decimal point even if the numbers divide evenly.
- Floor division (//): divides the first number by the second and then rounds down, evaluating to an integer.
- Modulo (%): evaluates to the positive remainder left over from division.

3 Assignment expressions

Take an example below.

```
>>> a = (100 + 50) // 2
>>> a
75
```

Note that a is bound to the value 75, not the expression (100 + 50 // 2). **Names are bound to values but not expressions.**

4 Lambda expressions

An expression that evaluates to a function. **Important: No "return" keyword!**

```
>>> x = 10
>>> square = x * x
>>> square = lambda x: x * x
```

The example shows that square = lambda x: x * x is a function with formal parameter x, that returns the value of "x * x". **It must be a single expression.**

The tiny difference between lambda expression and def statement is that only the def statement gives the function an intrinsic name.

5 Environment diagrams

- Every user-defined function has a parent frame (often global).
- The parent of a function is the frame in which it was defined.
- Every local frame has a parent frame (often global).
- The parent of a frame is the parent of the function called.

6 Self reference

A function can refer to its own name within its body. Let's see one example below.

```
def print_all(x):
    print(x)
    return print_all
print_all(1)(3)(5)
# Output:
1
3
5
```

7 Tree recursion

Tree-shaped processes arise whenever executing the body of a recursive function makes more than one call to that function.(e.g. Fibonacci Sequence, $fib(n) = fib(n-1) + fib(n-2)$)

8 Function currying

Currying: Transforming a multi-argument function into a single-argument, higher-order function.

9 Function decorators

```
@trace1
def triple(x):
    return 3 * x
```

is identical to

```
def triple(x):
    return 3 * x
triple = trace1(triple)
```

@trace is the function decorator, and the function ”triple” is the decorated function. What a function decorator do is rebinding the name of the function to a traced version of that function.

Here is the definition of the function ”trace”.

```
def trace1(fn):
    def traced(x):
        print('Calling', fn, 'on argument', x)
        return fn(x)
return traced
```

10 Range

For an expression range(m,n), it represents a consecutive sequence, which includes m but excludes n. List comprehension is a powerful tool using for expression.(e.g. [x for x in odds if 25 % x == 0], odds is a list)

11 Strings

Single-quoted and double-quoted strings are equivalent, but if there is an apostrophe in the string, it should be a double-quoted string. A triple-quoted string can span multiple lines.

Strings are sequences. Length and element selection are similar to all sequences. **But be careful that an element of a string is itself a string, but with only one element!**

However, the ”in” and ”not in” operators match substring.

```
>>> s = 'Hello'
>>> s.upper()
'HELLO'
>>> s.lower()
'hello'
>>> s.swapcase()
'hELLO'
>>> a = 'A'
>>> ord(a)
65
```

12 Dictionary

```
    numerals = {'I': 1, 'V': 5, 'X': 10}
    >>> numerals['X']
    10
    >>> numerals[10]
    ERROR(10 is not a key)
    >>> numerals.key()
    dict_keys(['X', 'V', 'I'])
    >>> numerals.values()
    dict_values([10, 5, 1])
    >>> numerals.items()
    dict_items([('X', 10), ('V', 5), ('I', 1)])
    >>> items = [('X', 10), ('V', 5), ('I', 1)]
    >>> dict(items)
    {'I': 1, 'V': 5, 'X': 10}
    >>> 'X' in numerals
    True
    >>> numerals.get('X', 0)
    10
    >>> numerals.get('X-ray', 0)
    0
    # Dictionary comprehension
    >>> {x:x*x for x in range(5)}
    {0: 0, 1: 1, 2: 4, 3: 9, 4: 16}
```

Dictionary is built by some pairs shaped like "key: value". Dictionaries don't have an order inherently. They establish a relationship between value and key, but different values have no relationship. So Python free to shuffle them around however it chooses.

Restrictions: You can't have the same key twice. If you do it will just throw out one of your elements. It's fine to have a sequence of multiple values for a given key. However, you are not allowed to use lists or dictionaries(or any mutable type) as keys.

13 Slicing

```
>>> odds = [3, 5, 7, 9]
>>> odds[1:3]
[5, 7]
>>> odds[1:]
[5, 7, 9]
>>> odds[:3]
[3, 5, 7]
>>> odds[:]
[3, 5, 7, 9]
```

Slicing always creates new values.

14 Sequence Aggregation

Some built-in functions are able to take iterable arguments and aggregate them into a value.

- **sum**(iterable[, start]) → value: "start" is an optional argument. Function "sum" adds every element in "iterable" to "start" or 0 and return the sum. **Notice**: It can't plus strings!

  ```
  >>> sum([2, 3, 4], 5)
  14
  >>> sum([[2, 3], [4]], [])
  [2, 3, 4]
  ```

- **max**(iterable[, key=func]) → value: return the item that gets the largest value under the key function.

  ```
  >>> max(range(10))
  9
  >>> max(range(10), key=lambda x: 7-(x-4)*(x-2))
  3
  ```

- **all**(iterable) → bool: return True if bool(x) is True for all values x in the iterable. **If the iterable is empty, return True.**

15 Mutation operators

```
>>> suits = ['coin', 'string', 'myriad']
>>> original_suits = suits
```

```
    >>> suits.pop()
    'myriad'
    >>> suits.remove('string')
    >>> suits
    ['coin']
    >>> suits.append('cup')
    >>> suits.extend(['sword', 'club'])
    >>> suits
    ['coin', 'cup', 'sword', 'club']
    >>> suits[2] = 'spade'
    >>> suits[0:2] = ['heart', 'diamond']
    >>> suits
    ['heart', 'diamond', 'sword', 'club']
    >>> original_suits
    ['heart', 'diamond', 'sword', 'club']
```

The same object can change in value throughout the course of computation. All names that refer to the same object are affected by a mutation. Only objects of **mutable types** can change: lists & dictionaries.

A function can change the value of any object in its scope.

16 Tuples

tuple() transfer an iterable sequence into a tuple.

```
>>> tuple([1, 2, 3, 4])
(1, 2, 3, 4)
>>> tuple({1:2, 3:4}) # return a tuple made up of keys
(1, 3)
>>> tuple((1, 2, 3, 4))
(1, 2, 3, 4)
```

Tuples are immutable sequences, but it may still change if it contains a mutable value as an element.

```
>>> s = ([1, 2], 3)
>>> s[0][0] = 4
>>> s
([4, 2], 3)
```

17 Identity operator

$\langle exp0 \rangle$ **is** $\langle exp1 \rangle$: return True if both $\langle exp0 \rangle$ and $\langle exp1 \rangle$ evaluate to the same object.

$\langle exp0 \rangle == \langle exp1 \rangle$: return True if both $\langle exp0 \rangle$ and $\langle exp1 \rangle$ evaluate to equal values.

A default argument value is part of a function value, not generated by a call, so mutable default arguments are dangerous.

```
>>> def f(s=[]):
...     s.append(5)
...     return len(s)
'''
Every time you call f, s is bound to the same value.
So s can be understood as a static variable.
'''
>>> f()
1
>>> f()
2
>>> f()
3
```

18 Nonlocal statement

nonlocal ⟨*name*⟩: future assignments to that name change its pre-existing binding in the **first non-local frame** of the current environment in which the name is bound.

From the Python 3 language reference:

- Names listed in a nonlocal statement must refer to pre-existing bindings in an enclosing scope.
- Names listed in a nonlocal statement must not collide with pre-existing bindings in the local scope.

Mutable values can be changed without a nonlocal statement.

19 Python Particulars

Python pre-computes which frame contains each name before executing the body of a function. Within the body of a function, all instances of a name must refer to the same frame.

20 Print

The Print function returns **None**. It also displays its arguments(separated by spaces) when it is called.

21 Object-oriented programming

21.1 Class statement

```
class <name>:
    <suite>
```

When a class is called:

1. A new instance of that class is created.(named self)
2. The `__init__` method of the class is called with the new boject as its first argument (named self), along with any additional arguments provided in the call expression.

```
class Account:
    def __init__(self, account_holder):
        self.balance = 0
        self.holder = account_holder
>>> a = Account('Jim')
# __init__ is called as a constructor
```

Identity operator 'is' and 'is not' test if two expressions evaluate to the same object.

21.2 Dot expression

```
<expression>.<name>
```

To evaluate a dot expression:

1. Evaluate the ⟨*expression*⟩ to the left of the dot, which yields the object of the dot expression.
2. ⟨*name*⟩ is matched against the instance attributes of that object; if an attribute with that name exists, its value is returned.
3. If not, ⟨*name*⟩ is looked up in the class, which yields a class attribute value.
4. That value is returned unless it is a function, in which case a bound method is returned instead.

21.3 Methods

```
class Account:
    def __init__(self, account_holder):
        self.balance = 0
        self.holder = account_holder
    def deposit(self, amount):
        self.balance = self.balance + amount
        return self.balance
    def withdraw(self, amount):
        if amount > self.balance:
            return 'Insufficient funds'
        self.balance = self.balance - amount
        return self.balance
```

Dot notation automatically supplies the first argument to a method.
Built-in functions:

- getattr(object, attribute_str): return the value of the attribute in the object. It's equivalent to object.attribute.
- hasattr(object, attribute_str): return True if the object has the attribute.

Python distinguishes between:

- Functions, which are created in classes or in the global environment.
- Bound methods, which couple together a function and the object on which that method will be invoked.

Object + Function = Bound Method

21.4 Attributes

Class attributes are shared across all instances of a class because they are attributes of the class, not the instance. So class attributes are not copied into every instance of the class, but kept in the class as shared attributes. Its changes will reflect in all of the instances. Instance attributes are private attributes, different from instance to instance.

```
class Account:
    interest = 0.02 # A class attribute
    def __init__(self, account_holder):
        self.balance = 0
        self.holder = account_holder
```

21.5 Assignments to attributes

Assignment statements with a dot expression on their left-hand side affect attributes for the object of that dot expression.

- If the object is an instance, then assignment sets an instance attribute. It will add or modify the attribute of that name.
- If the object is a class, then assignment sets a class attribute.

21.6 Inheritance

Inheritance is a method for relating classes together.

```
    class <name>(<base class>):
        <suite>
```

The new subclass ”shares” attributes with its base class, and the subclass may override certain inherited attributes.

The base class attributes aren’t copied into subclasses! To look up a name in a class:

1. If it names an attribute in the class, return the attribute value.
2. Otherwise, look up the name in the base class, if there is one.

Multiple Inheritance: A class may inherit from multiple base classes in Python.

The rule is that you look up at the subclass before you loop up at the base class.

But multiple inheritance is very complicated, so when you use it, be careful.

21.7 String representation

In Python, all objects produce two string representations:
str: legible to human
repr: legible to the Python interpreter

The repr function returns a Python expression (a string) that evaluates to an equal object. The result of calling repr on a value is what Python prints in an interactive session. eval(repr(object)) == object

```
>>> repr(min)
'<built-in function min>'
>>> 12e5
1200000.0
>>> print(repr(12e5))
1200000.0
```

The result of calling str on the value of an expression is what Python prints using the print function. print(object) is equivalent to print(str(object)).

```
>>> from fractions improt Fraction
>>> half = Fraction(1, 2)
>>> repr(half)
'Fraction(1, 2)'
>>> half
Fraction(1, 2)
>>> eval(repr(half))
```

```
Fraction(1, 2)
>>> str(half)
'1/2'
>>> print(half)
1/2
>>> eval(str(half))
0.5
```

21.8 Polymorphic functions

Polymorphic function: A function that applies to many different forms of data.
str and repr are both polymorphic; they apply to any object.

```
    def repr(x):
        return type(x).__repr__(x)
```

str is a class, but not a function. An instance attribute called `__str__` is ignored. If no `__str__` attribute is found, uses repr string.

```
class Bear:
    def __repr__(self):
        return 'Bear()'
oski = Bear()
print(oski)
print(str(oski))
print(repr(oski))
print(oski.__str__())
print(oski.__repr__())

# Output:
Bear()
Bear()
Bear()
Bear()
Bear()
```

You can see the difference if `__str__` is defined:

```
class Bear:
    def __repr__(self):
        return 'Bear()'
    def __str__(self):
        return 'a bear'
oski = Bear()
print(oski)
print(str(oski))
print(repr(oski))
print(oski.__str__())
print(oski.__repr__())

# Output:
a bear
a bear
Bear()
a bear
Bear()
```

Both str and repr use `__str__` and `__repr__` from the class function, but not the method in the object.

```
class Bear:
    def __init__(self):
        self.__repr__ = lambda: 'oski'
        self.__str__ = lambda: 'this bear'
    def __repr__(self):
        return 'Bear()'
    def __str__(self):
        return 'a bear'
oski = Bear()
print(oski)
print(str(oski))
print(repr(oski))
print(oski.__str__())
print(oski.__repr__())

# Output:
a bear
a bear
Bear()
this bear
oski
```

Thus we can implement the function str and repr.

```
class Bear:
    def __init__(self):
        self.__repr__ = lambda: 'oski'
        self.__str__ = lambda: 'this bear'
    def __repr__(self):
        return 'Bear()'
    def __str__(self):
        return 'a bear'
oski = Bear()
print(oski)
print(str(oski))
print(repr(oski))
print(oski.__str__())
print(oski.__repr__())

def repr(x):
    return type(x).__repr__(x)
def str(x):
    t = type(x)
    if hasattr(t, '__str__'):
        return t.__str__(x)
    else:
        return repr(x)

# Output:
a bear
a bear
Bear()
this bear
oski
```

21.9 Interfaces

Message passing: Objects interact by looking up attributes on each other. This attribute look-up rules allow different data types to respond to the same message.

A shared message(attribute name) that elicits similar behavior from different object classes is a powerful method of abstraction.

An interface is a set of shared messages, along with a specification of what they mean.

Example: Classes that implement `__repr__` and `__str__` methods that return Python-interpretable and human-readable strings implement an interface for producing string representations.

```
class Ratio:
    def __init__(self, n, d):
        self.numer = n
        self.denom = d
    def __repr__(self):
        return 'Ratio({0}, {1})'.format(self.numer, self.denom)
    def __str__(self):
        return '{0}/{1}'.format(self.numer, self.denom)

>>> half = Ratio(1, 2)
>>> print(half)
1/2
```

```
>>> half
Ratio(1, 2)
```

21.10 Special method names

Certain names are special because they have built-in behavior. These names always start and end with two underscores.

- `__init__`: Method invoked automatically when an object is constructed.
- `__repr__`: Method invoked to display an object as a Python expression.
- `__add__`: Method invoked to add one object to another.
- `__bool__`: Method invoked to convert an object to True or False.
- `__float__`: Method invoked to convert an object to a float (real number).

22 Measuring efficiency

22.1 Space

Active environments:

- Environments for any function calls currently being evaluated.
- Parent environments of functions named in active environments.

23 Linked lists

A linked list is either empty or a first value and the rest of the linked list.

Linked list is made of some link instances, which have "first" and "rest" attributes. "first" is the value of the first element in the linked list, and "rest" points to the rest as a linked list. The last link instance's "rest" points to Link.empty.

A linked list is a pair of values. The first element is an attribute value, the rest of the elements are stored in a linked list. A class attribute represents an empty linked list.

```
class Link:
    empty = () # Some zero-length sequence
    def __init__(self, first, rest=empty):
        assert rest is Link.empty or isinstance(rest, Link)
        self.first = first
        self.rest = rest
```

24 Property methods

In some cases, we want the value of instance attributes to be computed on demand.

The @property decorator on a method designates that it will be called whenever it is looked up on an instance.

A @<attribute>.setter decorator on a method designates that it will be called whenever that attribute is assigned. ¡attribute¿ must be an existing property method.

```
    class Link:
        ...
        @property
        def second(self):
            return self.rest.first
        @second.setter
        def second(self, value):
            self.rest.first = value

        >>> s
        Link(3, Link(4,Link(5)))
        >>> s.second
        4
        >>> s.second = 6
        >>> s.second
        6
        >>> s
        Link(3, Link(6,Link(5)))
```

25 Sets

One more built-in Python container type.

- Set literals are enclosed in braces
- Duplicate elements are removed on construction
- Sets have arbitrary order, just like dictionary entries

Set is implemented by linked list.

```
>>> s = {3, 2, 1, 4, 4}
>>> s
{1, 2, 3, 4}
>>> 3 in s
True
>>> len(s)
```

```
4
>>> s.union({1, 5}) # return a new set
{1, 2, 3, 4, 5}
>>> s.intersection({6, 5, 4, 3})
{3, 4}
>>> s
{1, 2, 3, 4}
>>> s.add(5)
>>> s
{1, 2, 3, 4, 5}
```

It's not allowed to have a set with a list inside of it.

26 Exceptions

A built-in mechanism in a programming language to declare and respond to exceptional conditions.

Mastering exceptions: Exceptions are objects! They have classes with constructors. They enable non-local continuations of control. (non-local jump)

26.1 Raising exceptions

26.1.1 Assert statements

Assert statements raise an exception of type AssertionError:

assert `<expression>`, `<string>`

Assertions are designed to be used liberally. They can be ignored to increase efficiency by running Python with the "-O" flag. "O" stands for optimized.

Whether assertions are enabled is governed by a bool `__debug__`.

26.1.2 Raise statements

Exceptions are raised with a raise statement.

raise `<expression>`

`<expression>` must evaluate to a subclass of BaseException or an instance of one.

Exceptions are constructed like any other object. E.g., TypeError('Bad argument!')

- TypeError: A function was passed the wrong number/type of argument
- NameError: A name wasn't found
- KeyError: A key wasn't found in a dictionary
- RuntimeError: Catch-all for troubles during interpretation

```
>>> raise TypeError('Bad argument')
TypeError: Bad argument
```

26.2 Handling exceptions

26.2.1 Try statements

Try statements handle exceptions.

```
try:
    <try suite>
except <exception class> as <name>:
    <except suite>
```

Exception rule:

1. The `<try suite>` is executed first.
2. If, during the course of executing the `<try suite>`, an exception is raised that is not handled otherwise, and if the class of the exception inherits from `<exception class>`, then
3. The `<except suite>` is executed, with `<name>` bound to the exception.

```
>>> try
        x = 1/0
    except ZeroDivisionError as e:
        print('handling a', type(e))
        x = 0
```

Multiple try statements: Control jumps to the except suite of the most recent try statement that handles that type of exception.

27 Dynamic scope

The way in which names are looked up in Scheme and Python is called lexical scope (or static scope).

Lexical scope: The parent of a frame is the environment in which a procedure was defined.

Dynamic scope: The parent of a frame is the environment in which a procedure was called.

28 Functional programming

All functions are pure functions; No re-assignment and no mutable data types; Name-value bindings are permanent.

Advantages of functional programming:

- The value of an expression is independent of the order in which sub-expressions are evaluated.

- Sub-expressions can safely be evaluated in parallel or on demand (lazily).
- Referential transparency: The value of an expression does not change when we substitute one of its sub-expressions with the value of that sub-expression.

29 Iterators

A container can provide an iterator that provides access to its elements in some order.

- **iter**(iterable): Return an iterator over the elements of an iterable value
- **next**(iterator): Return the next element in an iterator

```
>>> s = [3, 4, 5]
>>> t = iter(s)
>>> next(t)
3
>>> next(t)
4
```

Iterators are always ordered, even if the container that produced them is not.

```
>>> d = {'one': 1, 'two': 2, 'three': 3}
>>> k = iter(d)
>>> next(k)
'one'
>>> next(k)
'three'
>>> next(k)
'two'
>>> s = iter(d.values())
>>> next(s)
1
>>> next(s)
3
>>> next(s)
2
```

Keys and values are iterated over in an arbitrary order which is non-random, varies across Python implementations, and depends on the dictionary's history of insertions and deletions. If keys, values and items views are iterated over with no intervening modifications to the dictionary, the order of items will directly correspond.

```
>>> s = [1, 2, 3]
>>> next(s)
1
>>> list(s)
[2, 3]
>>> next(s)
StopIteration
```

29.1 Built-in functions for iteration

Many built-in Python sequence operations return iterators that compute results **lazily**.

All of the functions below return iterator:

- **map**(func, iterable): Iterate over func(x) for x in iterable
- **filter**(func, iterable): Iterate over x in iterable if func(x)
- **zip**(first_iter, second_iter): Iterate over co-indexed (x, y) pairs
- **reversed**(sequence): Iterate over x in a sequence in reverse order

To view the contents of an iterator, place the resulting elements into a container:

- **list**(iterable): Create a list containing all x in iterable
- **tuple**(iterable): Create a tuple containing all x in iterable
- **sorted**(iterable): Create a sorted list containing all x in iterable

```
>>> def double(x):
...     print('**', x, '=>', 2*x, '**')
...     return 2*x
>>> m = map(double, range(3, 7))
>>> f = lambda y: y >= 10
>>> t = filter(f, m)
>>> next(t)
** 3 => 6 **
** 4 => 8 **
** 5 => 10 **
10
>>> next(t)
** 6 => 12 **
12
>>> list(t)
[]
>>> d = {'a': 1, 'b': 2}
>>> d
{'b': 2, 'a': 1}
>>> items = iter(d.items())
>>> next(items)
('b', 2)
>>> next(items)
('a', 1)
>>> items = zip(d.keys(), d.values())
>>> next(items)
('b', 2)
>>> next(items)
('a', 1)
```

29.2 Generators

Generator is a special kind of iterator. A generator is returned from a generator function.

```
>>> def plus_minus(x):
...     yield x
...     yield -x

>>> t = plus_minus(3)
>>> next(t)
3
>>> next(t)
-3
```

A generator function is a function that yields values instead of returning them. A normal function returns once; a generator function can yield multiple times. A generator is an iterator created automatically by calling a generator function. When a generator function is called, it returns a generator that iterates over its yields.

Important: When you create a generator by calling a generator function, you haven't even begun executing the body of this function yet. It's not until next is called that the body begins to be executed and it keeps executing until a yield statement is reached. In a word, the computation will be executed lazily.

A **yield from** statement yields all values from an iterator or iterable (Python 3.3)

```
def a_then_b(a, b):
    yield from a
    yield from b

>>> list(a_then_b([3, 4], [5, 6]))
[3, 4, 5, 6]
```

Here is an example about substrings:

```
def prefixes(s):
    if s:
        yield from prefixes(s[:-1])
        yield s
def substrings(s):
    if s:
        yield from prefixes(s)
        yield from substrings(s[1:])
```

30 Python and SQL

A Python program can construct and execute SQL statements.

```
import sqlite3

db = sqlite3.Connection("n.db")
db.execute("CREATE TABLE nums AS SELECT 2 UNION SELECT 3;")
db.execute("INSERT INTO nums VALUES (?), (?), (?);", range(4,7))
print(db.execute("SELECT * FROM nums;").fetchall())

[(2,), (3,), (4,), (5,), (6,)]

db.commit() # save changes

db.executescript(cmd)
# It takes a string with many statements and execute them all.
```

Important: If you'd like to insert a string into a larger SQL statement, you should do it using the execute statement's built-in option of passing in values.

31 Database connections

Multiple programs can connect to the same database at the same time and they can modify and check the same one.