CS 61A Notes

By: Ian Dong

Python Notes

- print(<func>) ⇒ Function
- order of operations
  - **
  - positive/negation
  - *, /, %
  - +, -
  - comparisons, memberships
  - not
  - and
  - or
  - lambda

higher order ↓ lower order

- New List
  1. lst + lst
  2. lst * n or n * lst
  3. lst[i:j]
  4. list(lst)

- only .rest can destructively change the passed in LinkedList since it references that LinkedList

- Mutations
  1. lst += lst
  2. .append() ⇒ adds to the end
  3. .extend(l) ⇒ adds all elements of a list to another
  4. .pop(i) ⇒ returns the item at index i
  5. .remove(i) ⇒ removes 1st instance of i

- slicing out of bounds does not error. returns empty list
- Tree branches must be Trees
- Shallow vs Deepcopy
  - only iterating thru the list will it be a deep copy
- max([]) and min([]) errors out but sum([]) returns 0.
- always print repr of an object when eval in an interactive session.
- for Linked lists the left side will update its box's contents while the right side evals to what the pointer is pointing at
- You can pass a child instance into a parent function
- super() doesn't require self but ParentClass does when calling on them
- @property converts a function into a boolean

- **Iterable**
  - list
  - iterators
  - tuple
  - dict
  - tuple
  - strings
- **Function that Returns Iterators**
  - reversed(iterable)
  - zip(iterable 1, iterable 2)
  - map(func, iterable)
  - filter(func, iterable)
  - __iter__(self)

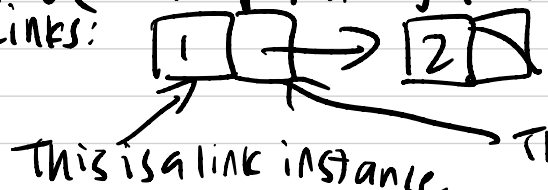
```
for x in self.lst
    yield x
```

OR `return iter(self.lst)`

- **Functions That Return Iterables**
  - list(iterable)
  - tuple(iterable)
  - sorted(iterable)

Returns all items in iterable

- generator is a type of iterator which is a type of iterable
- iter(iterable) creates a new iterator but iter(iterator) returns itself
- explicit calls to __str__ and __repr__ on returned strings will have quotes.
- implicit calls to __str__ and __repr__ which is print() and obj eval on returned strings will have no quotes since we are printing the result of calling __str__
- For Links:



- str(self) → __str__(self) → __repr__(self) → <object>
- repr(self) → __repr__(self) → <object>
- print(obj) ⇒ print(str(obj))
- obj ⇒ print(repr(obj))
- __iter__ returns an iterator object
- f-strings = f"{a}" where the value of a is eval. Embeds variables into a str
- look up procedure for Class.expressions = subclass instance variable → subclass class variable → subclass bound method → parent class variable → parent class bound method etc

Check str in all parent classes first

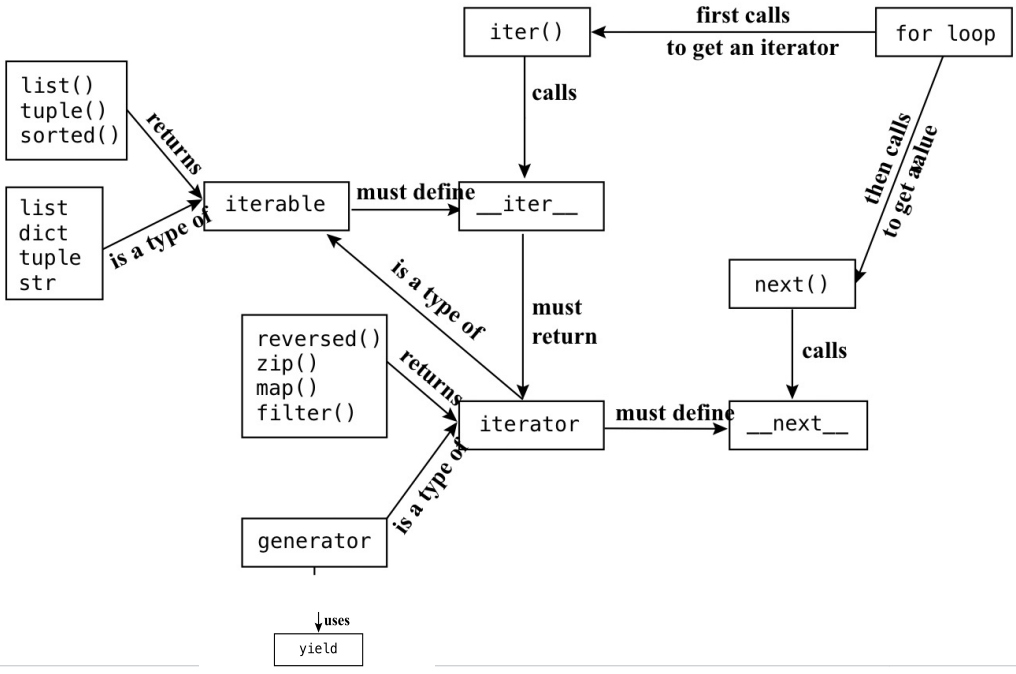
- calculate efficiency of the entire evaluation then discard lower terms and coefficients

Mutability in Lists

Function	Create or Mutate	Action/Return Value
lst.**append**(element)	mutate	attaches element to end of the list and returns None
lst.**extend**(iterable)	mutate	attaches each element in iterable to end of the list and returns None
lst.**pop**()	mutate	removes last element from the list and returns it
lst.**pop**(index)	mutate	removes element at index and returns it
lst.**remove**(element)	mutate	removes element from the list and returns None
lst.**insert**(index, element)	mutate	inserts element at index and pushes rest of elements down and returns None
lst += lst2	mutates	attaches lst2 to the end of lst and returns None same as lst.extend(lst2)
lst[start:end:step size]	create	creates a new list that start to stop (exclusive) with step size and returns it
lst = lst2 + [1, 2]	create	creates a new list with elements from lst2 and [1, 2] and returns it
list(iterable)	create	creates new list with elements of iterable and returns it

(credits: Mihira Patel)

- list slice assignment removes that portion of the list and replaces it with a new list and moves everything after the slice past the new list.
- For efficiency drop coefficients and lower terms
- raise StopIteration in custom next method
- If there is no return statement in a function the return value is a generator instance
- == is the same thing as __eq__()



Scheme Notes

- Defining functions and variables will return the name
- Creating lambda function will return the entire lambda function
- Only falsy value is #f
- and: return the first false value. if none of them are false return the last one
- or: return the first True value if none of them are true return #f
- = is equal check which only works for numbers
- eq? is identity check which works for num, bool, symbol
- equal? is value check which works for value of object
- List
  - '(1 2 3)
  - (cons 1 (cons 2 (cons 3 nil)))
  - (list 1 2 3)
- (map procedure lst)
- (filter pred lst)
- returning symbols will eval them so there are no quotes
- cons must take in two operands
- eval lists in the interactive session creates tuples with the elements inside
- cdr returns everything except
- quotient is floordiv and / is truediv
- General Rules
  1. name: look up in environment
  2. literal: numbers, booleans which eval to themselves
  3. call expression:
     1. eval operator
     2. eval operands
     3. apply operator to operands
  4. Define variable: eval last operand
  5. define function: eval entire line
  6. eval entire bodies and call expressions as a whole once
- don't need to eval special forms
- Special Forms
  - begin
  - if
  - quote
  - define
  - cond
  - and
  - let
  - or
  - lambda
- let: only eval last
- begin: eval everything except entire line
- cond: eval each if until there is a true statement
- let is similar to Python's multiple assignment
- mu procedures' parents are the frames in which they were called
- lambda procedures' parents are the frames in which they were defined
- malformed lists include a . between the car and cdr

- A tail call is the expression in a tail context and does not need more computation
- A tail context is the last body in a lambda function, the 2nd and 3rd bodies in a tail context if expression, all non predicate bodies in cond the last body in and, or, begin, let, and
- In a tail recursive function, every recursive call must be a tail call
- Subexpressions within a tail context are not in a tail context since the whole body of code is in a tail context

Quasiquotation

There are two ways to quote an expression:

- Quote: `'(a b)` => `(a b)`
- Quasiquote: `` `(a b) `` => `(a b)`

They are different because parts of a quasiquoted expression can be **unquoted** with `,`

```
(define b 4)
```

- Quote: `'(a ,(+ b 1))` => `(a (unquote (+ b 1))`
- Quasiquote: `` `(a ,(+ b 1)) `` => `(a 5)`

- map applies the function to every element in the list

- **Lexical (static) scope**: The parent of a frame is the frame in which a procedure was *defined*
- **Dynamic scope**: The parent of a frame is the frame in which a procedure was *called*

- the last symbol in any series of statements gets returned.

Regex

The re module

The re module provides many helpful functions.

Function	Description
`re.search(pattern, string)`	returns a match object representing the first occurrence of pattern within string
`re.fullmatch(pattern, string)`	returns a match object, requiring that pattern matches the entirety of string
`re.match(pattern, string)`	returns a match object, requiring that string starts with a substring that matches pattern
`re.findall(pattern, string)`	returns a list of strings representing all matches of pattern within string, from left to right
`re.sub(pattern, repl, string)`	substitutes all matches of pattern within string with repl

will return all str in ()
?: will not capture that str

- regex returns the whole pattern if there are no capturing groups
- use r" " in python for regex
- fullmatch, match, findall always prioritizes capturing groups before the full expression

Match groups

- group(0) grabs the result of the re function

If there are parentheses in a patterns, each of the parenthesized groups will become groups in the match object.

```
x = "There were 12 pence in a shilling and 20 shillings in a pound."
mat = re.search(r'(\d+).*(\d+)', x)
```

```
mat.group(0)  # '12 pence in a shilling and 20'
mat.group(1)  # 12
mat.group(2)  # 20
mat.groups()  # (12, 20)
```

{x, y} x ≤ # of times ≤ y
{x, } x ≤ # of times < ∞
{, y} # of times ≤ y
{x} x = # of times = x

Regex Notes

- regex is case/space/length sensitive
- use \ for special characters for both Regex and BNF
- The following are special characters in regular expressions: `\ ( ) [ ] { } + * ? | $ ^ .` - when it is inside a set since it represents a range

Character classes

Pattern	Description	Example	Matches:
`[]`	Denotes a *character class*. Matches characters in a set (including ranges of characters like `0-9`). Use `[^]` to match characters outside a set.	`[top]`	t
`.`	Matches *any character* other than the newline character.	`1.`	1?
`\d`	Matches any *digit character*. Equivalent to `[0-9]`. `\D` is the *complement* and refers to all non-digit characters.	`\d\d`	12
`\w`	Matches any *word character*. Equivalent to `[A-Za-z0-9_]`. `\W` is the complement.	`\d\w`	4Z
`\s`	Matches any *whitespace character*: spaces, tabs, or line breaks. `\S` is the complement.	`\d\s\w`	9 a

^ represents the complement inside a character class

Combining patterns

Patterns `P1` and `P2` can be combined in various ways.

Combination	Description	Example	Matches:
`P1P2`	A match for P_1 followed immediately by one for P_2.	`ab[.,]`	ab,
`P1|P2`	Matches anything that either P_1 or P_2 does.	`\d+|Inf`	Inf
`(P1)`	Matches whatever P_1 does. Parentheses group, just as in arithmetic expressions.	`(<3)+`	<3<3<3

Quantifiers

These indicate how many of a character/character class to match.

- can use () for longer patterns/capturing groups

Pattern	Description	Example	Matches:
`*`	Matches *0 or more* of the previous pattern.	`a*`	aaa
`+`	Matches *1 or more* of the previous pattern.	`lo+l`	lool
`?`	Matches *0 or 1* of the previous pattern.	`lo?l`	lol
`{}`	Used like `{Min, Max}`. Matches a *quantity* between Min and Max of the previous pattern.	`a{2,4}`	aaaa

- no spaces between min, max
- ?: is a non capturing group

Anchors

These don't match an actual character, they indicate the position where the surrounding pattern should be found.

Pattern	Description	Example	Matches:
`^`	Matches the *beginning* of a string.	`^aw+`	aww
`$`	Matches the *end* of a string.	`\w+y$`	stay
`\b`	Matches a *word boundary*, the beginning or end of a word	`\w+e\b`	bridge

- use ^ and $ if you only want the full string to match
- use \b and \b if you only want a full word within a string to match

Ambiguous quantifiers

Likewise, there is ambiguity with `*`, `+`, and `?`.

```
mat = re.match(r'(x*)(.*)', 'xxx')
mat.groups()  # ('xxx', '')

mat = re.match(r'(x+)(.*)', 'xxx')
mat.groups()  # ('xxx', '')

mat = re.match(r'(x?)(.*)', 'xxx')
mat.groups()  # ('x', 'xx')

mat = re.match(r'(.*)/(.+)', '12/10/2020')
mat.groups()  # ('12/10', '2020')
```

Python chooses to match **greedily**, matching the pattern left-to-right and, when given a choice, matching as much as possible while still allowing the rest of the pattern to match.

Lazy operators

Sometimes, you don't want to match as much as possible.

The lazy operators `*?`, `+?`, and `??` match only as much as necessary for the whole pattern to match.

```
mat = re.match(r'(.*)(\d*)', 'I have 5 dollars')
mat.groups() # ('I have 5 dollars', '')

mat = re.match(r'(.*?)(\d+)', 'I have 5 dollars')
mat.groups() # ('I have ', '5')

mat = re.match(r'(.*?)(\d*)', 'I have 5 dollars')
mat.groups() # ('', '')
```

The ambiguities introduced by `*`, `+`, `?`, and | don't matter if all you care about is whether there is a match!

BNF Notes

Basic BNF

A BNF grammar consists of a set of grammar rules. We will specifically use the rule syntax supported by the Lark Python package.

The basic form of a grammar rule:

```
symbol0: symbol1 symbol2 ... symboln
```

Symbols represent sets of strings and come in 2 flavors:

- **Non-terminal symbols**: Can expand into either non-terminal symbols (themselves) or terminals.
- **Terminal symbols**: Strings (inside double quotes) or regular expressions (inside forward slashes).

To give multiple alternative rules for a non-terminal, use `|`:

BNF example

• Parsing tree includes non-terminal symbols

A simple grammar with three rules:

```
?start: numbers
numbers: INTEGER | numbers "," INTEGER
INTEGER: /-?\d+/
```

" = ,

For the Lark library,

- Grammars need to start with a `start` symbol.
- Non-terminal symbol names are written in lowercase.
- Terminal symbols are written in UPPERCASE.

What strings are described by that grammar?

Defining terminals

Terminals are the base cases of the grammar (like the tokens from the Scheme project).

In Lark grammars, they can be written as:

- Quoted strings which simply match themselves (e.g. `"*"` or `"define"`)
- Regular expressions surrounded by `/` on both sides (e.g. `/\d+/`)
- Symbols written in uppercase which are defined by lexical rules (e.g. `NUMBER: /\d+(\.\d+)/`

It's common to want to always ignore some terminals before matching. You can do that in Lark by adding an `%ignore` directive at the end of the grammar.

```
%ignore /\s+/ // Ignores all whitespace
```

Repetition

EBNF is an extension to BNF that supports some shorthand notations for specifying how many of a particular symbol to match.

EBNF	Meaning	BNF equiv
`item*`	Zero or more items	`items: | items item`
`item+`	One or more items	`items: item | items item`
`item?`	Optional item	`optitem: | item`

All of our grammars for Lark can use EBNF shorthands.

Grouping

Parentheses can be used for grouping.

```
name : /\w+/
number: /\d+/
list: ( name | number )+
```

Square brackets indicate an optional group.

```
numbered_list : ( name [ ":" number ] )+
```

Exercise: Describe a comma-separated list of zero or more names (no comma at the end).

```
comma_separated_list : [ name ("," name)* ]
```

Importing common terminals

Lark also provides pre-defined terminals for common types of data to match.

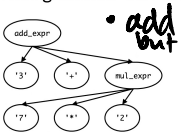
```
%import common.NUMBER
%import common.SIGNED_NUMBER
%import common.DIGIT
%import common.HEXDIGIT
```

Ambiguity resolution

One way to resolve this ambiguity:

```
?start: expr
?expr: add_expr
?add_expr: mul_expr | add_expr ADDOP mul_expr
?mul_expr: NUMBER | mul_expr MULOP NUMBER
ADDOP: "+" | "-"
MULOP: "*" | "/"
```

That grammar can only produce this parse tree:

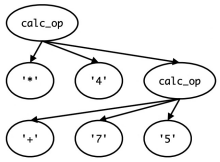


• add can be the parent of mul but not the other way around

Calculator tree breakdown

```
?start: calc_expr
?calc_expr: NUMBER | calc_op
calc_op: "(" OPERATOR calc_expr* ")"
OPERATOR: "+" | "-" | "*" | "/"
```

• don't need ? in front ?-nodes



- Leaves are either terminals or non-terminals with no children.
- Lark removes parts of the rules that are quoted-string terminals (like `"("`) but does show named terminals (like `NUMBER`) or terminals defined by regular expressions.
- Lark removes any nodes whose rules start with `?` and have only one child, replacing them with that child (like `calc_expr`).

• might keep nodes w/ ? if it needs it for branching purposes
• remove terminal/?/single branch nodes

Because the tree is simplified, we call it an **abstract syntax tree**.

Example: Sentences

```
?start: sentence
sentence: noun_phrase verb
noun: NOUN
noun_phrase: article noun
article : | ARTICLE // The first option matches ""
verb: VERB
NOUN: "horse" | "dog" | "hamster"
ARTICLE: "a" | "the"
VERB: "stands" | "walks" | "jumps"
%ignore /\s+/
```

What strings can this grammar parse?

```
the horse jumps
a dog walks
hamster stands
```

SQL Notes

SELECT statement

The `SELECT` statement queries a database and returns zero or more rows of data.

Return all the rows:

```
SELECT * FROM songs;
```

Return a subset of columns from all rows:

```
SELECT artist, title FROM songs;
```

Rename columns in the returned data:

```
SELECT artist AS singer, title AS song_title FROM songs;
```

Manipulate column values in the returned data:

```
SELECT title, views * 1000000 FROM songs;
```

ORDER BY clause

The `SELECT` statement can have multiple optional clauses.

The `ORDER BY` clause specifies the order of rows returned:

Return the rows sorted by a column (highest to lowest):

```
SELECT title, views FROM songs ORDER BY views DESC;
```

Ditto, but ascending (lowest to highest):

```
SELECT title, views FROM songs ORDER BY views ASC;
```

Return the rows sorted by multiple columns:

```
SELECT title, release_year, views  FROM songs
    ORDER BY release_year DESC, views DESC; LIMIT #
```

WHERE clause

The `WHERE` clause can be used to filter rows, and must appear before the `ORDER BY` clause.

```
SELECT [columns] FROM [table] WHERE [condition] ORDER BY [order]
```

Returning rows that match a particular value:

```
SELECT title FROM songs WHERE artist = "Lil Nas X";
```

Using other comparison operators:

```
SELECT artist, title, release_year FROM songs
    WHERE release_year > 2020;
```

Using logical operators:

```
SELECT artist, title, release_year FROM songs
    WHERE release_year > 2020 AND views > 5;
```

Creating tables with SELECT

The `CREATE TABLE` statement can be used to create a table in various ways.

Creating a table with the results of a `SELECT`:

```
CREATE TABLE top_music_videos AS
    SELECT title, views FROM songs ORDER BY views DESC;
```

That limits the new table to a subset of existing data, however.

Creating tables with UNION

It's possible to use a `SELECT` to create a row of entirely new data, and save that into a table.

```
CREATE TABLE musical_movies AS
    SELECT "Mamma Mia" as title, 2008 as release_year;
```

We can use `UNION` to merge the results of multiple `SELECT` statements:

```
CREATE TABLE musical_movies AS
    SELECT "Mamma Mia" as title     , 2008 as release_year UNION
    SELECT "Olaf's Frozen Adventure", 2017  UNION
    SELECT "Across the Universe"    , 2007  UNION
    SELECT "Moana"                  , 2016  UNION
    SELECT "Moulin Rouge"           , 2001;
```

2-step table creation

The most common approach is to first use `CREATE` to declare the column names and types:

```
CREATE TABLE musical_movies (title TEXT, release_year INTEGER);
```

Then use `INSERT` to insert each row of data:

```
INSERT INTO musical_movies VALUES ("Mamma Mia", 2008);
INSERT INTO musical_movies VALUES ("Olaf's Frozen Adventure", 2017
INSERT INTO musical_movies VALUES ("Across the Universe", 2007);
INSERT INTO musical_movies VALUES ("Moana", 2016);
INSERT INTO musical_movies VALUES ("Moulin Rouge", 2001);
```

Joining related tables

A join on two tables A and B yields all combinations of a row from table A and a row from table B.

Select the parents of curly-furred dogs:

```
SELECT parent FROM parents, dogs
    WHERE child = name AND fur = "curly";
```

Joining a table with itself

Two tables may share a column name (especially when they're the same table!). Dot expressions and aliases disambiguate column values.

Select all pairs of siblings:

```
SELECT a.child AS first, b.child AS second
    FROM parents AS a, parents AS b
    WHERE a.parent = b.parent AND a.child < b.child;
```

String expressions

The `||` operator does string concatenation:

```
SELECT (views || "M") as total_views FROM songs;
```

There are basic functions for string manipulation as well:

```
SELECT SUBSTR(release_year, 3, 2) AS two_digit_year
   FROM songs ORDER BY two_digit_year;
```

Numerical expressions

Multiple parts of a `SELECT` statement can include an **expr**ession.

`SELECT [result-column] FROM [table] WHERE [expr];`

`result-column` can expand to either `expr` AS `column-alias` or `*`.

Expressions can contain function calls and arithmetic operators.

- Combine values: `+`, `-`, `*`, `/`, `%`, `and`, `or`
- Transform values: `ABS()`, `ROUND()`, `NOT`, `-`
- Compare values: `<`, `<=`, `>`, `>=`, `<>`, `!=`, `=`

Aggregate functions

So far, all SQL expressions have referred to the values in a single row at a time.

```
SELECT [columns] FROM [table] WHERE [expression];
```

An aggregate function in the `[columns]` clause computes a value from a group of rows.

Starting from this table of solar system objects, find the biggest:

```
SELECT MAX(mean_radius) FROM solar_system_objects;
```

Mixing aggregate functions & single values

An aggregate function also selects some row in the table to supply the values of columns that are not aggregated. In the case of `MAX` or `MIN`, this row is that of the `MAX` or `MIN` value.

```
SELECT body, MAX(mean_radius) FROM solar_system_objects;
```

```
SELECT body, MAX(surface_gravity) FROM solar_system_objects;
```

```
SELECT body, MIN(mean_radius) FROM solar_system_objects;
```

Otherwise, an individual value will just pick from an arbitrary row:

```
SELECT SUM(mass) FROM solar_system_objects;
```

Grouping rows

Rows in a table can be grouped using `GROUP BY`, and aggregation is performed on each group.

```
SELECT [columns] FROM [table] GROUP BY [expression];
```

The number of groups is the number of unique values of an expression.

Based on this animals table, find the max weight per each number of legs:

```
SELECT legs, max(weight) FROM animals GROUP BY legs;
```

Filtering groups

A `HAVING` clause filters the set of groups that are aggregated

```
SELECT [columns] FROM [table] GROUP BY [expression]
HAVING [expression];
```

Find the weight/leg ratios that are shared by more than one kind of animal:

```
SELECT weight/legs, count(*) FROM animals
    GROUP BY weight/legs HAVING COUNT(*) > 1;
```

• for each usually means to use GROUP BY