Handout 2: Python Expressions

Interpreter

1. Read-eval-print
   a. You type the *expression* "`2 + 2`"
   b. Python **evaluates** it to determine result: 4
   c. Python **prints** the result
2. Editing what you type
   a. Arrow keys
   b. c-A, c-E, c-U, c-K
   c. c-C, c-D
   d. c-P, c-N
3. Underscore
   a. A special variable, contains the **value** of the previous expression

```
>>> 40 + 2
42
>>> _ + 10
52
```

4. Printing is disabled during **batch** execution

```
$ cat foo.py
x = 2 + 2
y = 3 * x - 1
$ python foo.py
$ python
>>> import foo
>>>
```

5. Explicit printing: *returns no value*

```
>>> 42
42
>>> print(2 + 2)
4
>>> _ + 10
52
```

Variables

6. Variables

a. A name with a value:

```
>>> text1
<Text: Moby Dick by Herman Melville 1851>
```

b. **Assignment operator** `=`. Automatically creates variable.

```
>>> x = 5 + 7
>>> x
12
```

7. Don't try to get the value if you have never set it.

```
>>> z * 2
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  NameError: name 'z' is not defined
```

Mathematical and logical expressions

8. Arithmetic operators: `+ - * /`

a. Example:

```
>>> 3 + 4 * 2
11
```

b. Note that "`*`" **takes precedence** over "`+`"

c. Good practice to be explicit: use parentheses

9. Everything revolves around *evaluating* expressions.

a. What are the precise steps in evaluating line 3?

```
>>> lst = [2, 4, 8, 16]
>>> x = 2
>>> x = x * lst[x] + 42
```

b. Draw a tree

10. What happens? `x = 10 + print(2)`

11. Comparison expressions. Assume `x = 10` and `y = 2`.

a. Example:

```
>>> x == 10
True
>>> x == y
False
>>> x == y + 8
True
```

b. **Boolean** values: `True`, `False`

c. Complex comparisons

```
>>> (not x == 10) or (y == y and 2 + 2 == 4)
True
```

12. An assignment is a *command,* not an expression.

```
>>> y < 20 or x = 10
  File "<stdin>", line 1
    y < 20 or x = 10
              ^
SyntaxError: invalid syntax
```

13. Other comparison operators: `< > <= >= !=`

Strings

14. Strings

a. A string is a sequence of characters, "`+`" is concatenation

```
>>> 40 + 2
42
>>> "40" + "2"
'402'
```

b. Three options: `"hi"`, `'hi'`, `'''hi'''`.

c. How can you get a string whose only character is single quote?

d. What happens? `print('2 + 2', 2 + 2)`

15. Length. What is the value? `len("The End")`

16. Accessing the characters:

a. Numbered from 0. The *first character* is *character number zero.*

```
>>> s = 'abcde'
>>> s[0]
'a'
>>> s[1]
'b'
```

b. Convenience for getting the last element(s):

```
>>> s[-1]
'e'
>>> s[-2]
'd'
```

c. The space character

```
>>> 'The End'[3]
' '
```

17. A character is a string of length one!

```
>>> c = 'hi'[0]
>>> c
'h'
>>> len(c)
1
>>> c[0]
'h'
```

18. Slices

a. Extracting a substring

```
>>> s[1:3]
'bc'
```

b. Why doesn't it include `s[3]`?

```
>>> s[0:2]
'ab'
>>> s[2:len(s)]
'cde'
>>> len(s[1:3]) == 3 - 1
True
```

c. Convenient shorthands: `s[:2]`, `s[2:]`.

19. Converting between strings and lists

```
>>> 'dog walker'.split()
['dog', 'walker']
>>> ' '.join(['dog', 'walker'])
'dog walker'
>>> ''.join(['dog', 'walker'])
'dogwalker'
```

20. *Exercises.*

a. Set `s = 'pteranadon'`. Using the variable `s` but no literals, write an expression whose value is `'ted'`.

b. Assume `s = 'hi'` and `t = 'bye'`. Evaluate: `s[0] + t[len(s)-1:]`.

c. Assume `i = 2`. Evaluate: `'eagle'[i-1:i+1]`.

d. The following was intended to determine whether the length of s is 2 or not. Where is the error?

```
>>> len(s) = 2
```

Lists

21. Lists

a. Example: `[42, 'hi', -12]`

b. Ordered sequence of elements. `['a', 'b'] != ['b', 'a']`

c. Concatenation:

```
>>> [10,20] + [5,2]
[10, 20, 5, 2]
```

22. Accessing elements is just like accessing characters in strings:

```
>>> x = [10, 20, 30, 40, 50]
>>> x[1]
20
>>> x[-2]
40
```

23. Lists can contain any mix of objects

a. Example: `y = ['hi', 'there']`

b. What is `y[0][1]`? `y[1][0]`?

24. A slice of a list is a *sublist,* not an element

```
>>> x[1:2]
[20]
>>> x[2:]
[30, 40, 50]
```

25. Elements can be changed and added

```
>>> y = ['a', 'b']
>>> y[0] = 'A'
>>> y
['A', 'b']
>>> y.append('c')
>>> y
['A', 'b', 'c']
```

26. *Exercises.*

a. There are two kinds of square brackets in the following. Explain.

```
>>> mylist = [3, s[1], 'hi']
```

b. Evaluate.

```
>>> 'abcd'[:3]
>>> x[1:1]
>>> x[:]
>>> 'y' + 'water'[len('hi') + 1] * 2
```

Sets

27. Sets

a. Order doesn't matter, no duplicates

```
>>> {10, 42} == {42, 10}
True
>>> A = {'hi', 'there', 'hi'}
>>> A
{'there', 'hi'}
```

b. Elements print in random order

c. Number of elements: `len(A)`

d. Cannot access members by index, but can test for presence

```
>>> 'there' in A
True
>>> 'h' in A
False
```

28. The operator "`in`"

a. It also works with strings and lists:

```
>>> 'b' in ['a', 'b', 'c']
True
>>> 'b' in 'abc'
True
>>> 'is' in 'invisible'
True
```

b. The behavior with strings is different than with sets/lists—how so?

c. Explain exactly how this is evaluated, and what result you get:

```
>>> ('the' + 'are'[1:]) in ['hi', 'there']
```

29. Set operations. Set `B = {'hi', 'bye'}`.

a. Intersection

```
>>> A & B
{'hi'}
```

b. Union

```
>>> A | B
{'there', 'hi', 'bye'}
```

c. Difference:

```
>>> A - B
{'there'}
>>> B - A
{'bye'}
```

d. **Caution:** empty set is `set()`, *not* `{}`.