COMP 2211 Exploring Artificial Intelligence
Python Fundamentals (A Crash Course/Review)
Huiru Xiao

Materials Provided by Dr. Desmond Tsoi
Department of Computer Science & Engineering
The Hong Kong University of Science and Technology, Hong Kong SAR, China

A Crash Course on Python Programming Language

The lecture notes serve as a crash course or a review of fundamental Python programming. You should have already learned the majority of these concepts in COMP 1021 Introduction to Computer Science. Please read and familiarize yourself with all the contents before attending the lecture on 'advanced' Python programming for AI. :)

Python

- Python is a high-level, dynamically typed multi-paradigm programming language.
- Python code is often said to be almost like pseudo-code, since it allows you to express very powerful ideas in very few lines of code while being very readable.
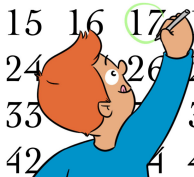
A Python Program that Prime Factorizes a Number

```python
# File: prime_factorize.py

def prime_check(n):
  for i in range(2,n):
    if n%i == 0: return False
  return True

n = int(input("Enter the Number: "))
print(str(n)+" = ",end='')
```

```python
for i in range (2,n+1):
  c = 0
  if prime_check(i) == True:
    while True:
      if(n%i == 0):
        n /= i
        c+= 1
      else: break
    if c == 1:
      print(str(i)+" x ",end='')
    elif c !=0:
      print(str(i)+"^"+str(c)+" x ",end='')

print(" \b\b\b ")
```

15 16 17 18 19 20 21
24 26 27 28 29 30
33 36 37 38 39
42 4 45 46 47 48

Python Versions

- As of January 1, 2020, Python has officially dropped support for Python 2.
- For this course, all code will use Python 3.11.13.
- The latest version is Python 3.15.4
- You can check your Python version at the command line after activating your environment by running

```
import sys
print(sys.version)
```

# Variables

- Variables are names given to computer memory locations in order to store data in a program.
- A variable is created the moment we first assign a value to it.
- Variables do not need to be declared with any particular type, and can even change type after they have been set.

**Syntax**
```
<variable name> = <value>
```

```
x = 4         # x is a variable of type int
x = 3.14      # x is now of type float
# More about data types soon
```
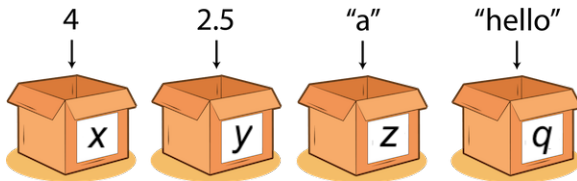
Data Types

- Like most languages, Python has a number of basic types including integers, floats, booleans, strings, and containers including lists, dictionaries, sets, tuples.
- The basic types behave in ways that like the other programming languages.

Name	Type	Description
Integers	int	Whole numbers, such as: 3, 300, 200
Floating points	float	Numbers with a decimal point: 2.3, 4.6, 100.0
Booleans	bool	Logical value indicating True or False
Strings	str	Ordered sequence of characters: "Hello", 'Desmond', "2000"
Lists	list	Ordered sequence of objects: [10, "Hello", 200.3]
Dictionaries	dict	Ordered* Key:Value pairs: {"mykey" : "value", "name" : "Desmond"}
Sets	set	Unordered collection of unique objects: {"a", "b"}
Tuples	tup	Ordered immutable sequence of objects: ("a", "b")

*Starting from Python 3.7, dictionaries are ordered

type() method returns class type of the argument(object) passed as parameter. type() function is mostly used for debugging purposes.

Syntax
```
type(<object>)
```

Integers and Floats

- Python supports integers and floating-point numbers.
- It also implements all arithmetic operators, i.e.,
  +, -, *, /, //, %, **, (), +=, -=, *=, /=, %=

```python
x = 3
print(type(x))              # Print "<class 'int'>"
print(x)                    # Print "3"
print(x + 1)                # Addition; print "4"
print(x - 1)                # Subtraction; print "2"
print(x * 2)                # Multiplication; print "6"
print(x / 2)                # Division, print "1.5"
print(x // 2)               # Integer division, print "1"
print(x % 2)                # Modulus; print "1"
print(x ** 2)               # Exponentiation; print "9"
x += 1;  print(x)           # Print "4"
x *= 2;  print(x)           # Print "8"
y = 2.5; print(type(y))     # Print "<class 'float'>"
print(y, y + 1, y * 2, y ** 2) # Print "2.5 3.5 5.0 6.25"
```

Booleans

- In Python, booleans represent one of two values: True or False.
- Python implements all the relational operators (comparison operators), i.e., $==, !=, >, >=, <, <=$, which returns True or False.
- Python also implements all of the usual operators for Boolean logic, but uses English (and/or) rather than symbols (&&, ||, etc.).

```python
print(1 + 1 != 3)  # Print "True"
print(4 + 6 == 10) # Print "True"
x = 3; y = 5
print(x + y > 7)   # Print "True"
print(x * y <= 7)  # Print "False"

t = True; f = False
print(type(t)) # Print "<class 'bool'>"
print(t and f) # Logical AND; print "False"
print(t or f)  # Logical OR; print "True"
print(not t)   # Logical NOT; print "False"
print(t != f)  # Logical XOR; print "True"
```

Strings

- A string is a sequence of characters.
- Strings in Python are surrounded by either single quotation marks, or double quotation marks.
- Python has great support for strings.

```python
hello = 'hello'             # String literals can use single quotes
world = "world"             # or double quotes; it does not matter
print(hello)                # Print "hello"
print(len(hello))           # String length; print "5"
hw = hello + ' ' + world    # String concatenation
print(hw)                   # Print "hello world"
hw12 = '%s %s %d' % (hello, world, 12)  # sprintf style string formatting
print(hw12)                 # Print "hello world 12"
print(type(hw12))           # Print "<class 'str'>"
print(hello * 3)            # Print "hellohellohello",
                            # i.e., "hello" is duplicated by 3 times
```

Strings

- We can add a prefix in front of strings.

Prefix	Meaning	Example
u or U	Unicode string (default in Python 3)	u"Desmond"
b or B	Byte (ASCII) string Bytes are machine-readable (can be directly stored on the disk, need decoding to become human-readable) and string is human-readable (need encoding before they can be stored on disk)	b"Desmond"
r or R	Raw string, i.e., a character following a backslash is included in the string without change. This is useful when we want to have a string that contains backslash and do not want it to be treated as an escape character.	r"\nDesmond\n"
f or F	Formatted string, i.e., contains expressions inside braces	a = 1 b = 2 print(f'a+b={a+b}')

Examples

```python
unicode_string = u'COMP2211'
print(unicode_string)           # Print "COMP2211"

byte_string = b'COMP2211'
print(type(byte_string))        # Print "<class 'bytes'>"
decode_string = byte_string.decode('utf-8')
print(type(decode_string))      # Print "<class 'str'>"
encode_string = decode_string.encode('utf-8')
print(type(encode_string))      # Print "<class 'bytes'>"

raw_string = r'Hi\nHello'
print(raw_string)               # Print "Hi\nHello"

a = 1
b = 2
print(f'a + b = {a+b}')         # Print "a + b = 3"
```

String Objects

- Strings are actually objects in Python.
- String objects have a bunch of useful methods; for example:

```python
s = "hello"
print(s.capitalize())          # Capitalize a string; prints "Hello"
print(s.upper())               # Convert a string to uppercase; prints "HELLO"
print(s.rjust(7))              # Right-justify a string, padding with spaces;
                               # Print "  hello"
print(s.center(7))             # Center a string, padding with spaces;
                               # Print " hello "
print(s.replace('l', '(ell)')) # Replace all instances of one substring
                               # with another; print "he(ell)(ell)o"
print('  world '.strip())      # Strip leading and trailing whitespace;
                               # print "world"
```

Full List of all String Methods

https://docs.python.org/3.10/library/stdtypes.html#string-methods

Type Conversion

- We can convert the data type of an object to required data type using the predefined functions like int(), float(), bool(), str(), etc to perform explicit type conversion.

Syntax

<required–datatype>(<expression>)

- Parameters:
  - required-datatype: int, float, bool, str, etc.
  - expression: The expression to be type-converted

```python
print(int('3') + 5)   # Print "8"
print('3' + str(5))   # Print "35"
print(float(3) + 3)   # Print "6.0"
print(int(3.14) + 3)  # Print "6"
```

Output Data to the Standard Output Device

- print() function is used to output data to the standard output device (screen).

Syntax

```
print(*<objects>, <sep>=' ', <end>='\n', <file>=sys.stdout, <flush>=False)
```

- Parameters:
    - objects: The value(s) to be printed
    - sep: The separator is used between the values. It defaults into a space character.
    - end: After all values are printed, end is printed. It defaults into a new line.
    - file: The object where the values are printed and its default value is sys.stdout (screen).
    - flush: To ensure that we get output as soon as print() is called, we need to set flush to True.

Output formatting can be done by using str.format() method. {} are used as placeholders, and we can specify the order in which they are printed by using numbers. Also, we can even use keyword arguments to format the string.

Output Data to Standard Output Device

```python
print('COMP2211 is the best COMP course :D') # Print a string

age = 18
print('I am', age, 'years old')          # Print 'I am 18 years old'

print(2, 2, 1, 1)                         # Print 2 2 1 1
print(2, 2, 1, 1, sep='@')                # Print 2@2@1@1
print(2, 2, 1, 1, sep='~', end='*')       # Print 2~2~1~1*
print()                                   # Print '\n', i.e., move to the next line

x = 'A+'; y = 'A'
# Print 'Desmond will get A+ and John will get A'
print('Desmond will get {} and John will get {}'.format(x,y))
# Print 'Desmond will get A+ and John will get A'
print('Desmond will get {0} and John will get {1}'.format(x,y))
# Print 'Desmond will get A and John will get A+'
print('Desmond will get {1} and John will get {0}'.format(x,y))
# Print 'Desmond will get A+ and John will get A'
print('Desmond will get {a} and John will get {b}'.format(a = 'A+', b = 'A'))
```

Input Data

- We can take the input from the user using input() function.
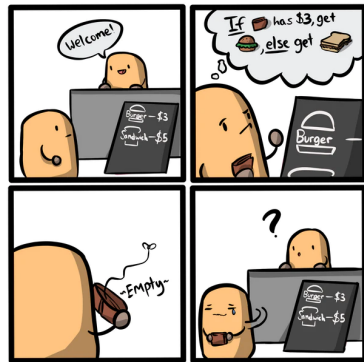
Syntax

```
input(<prompt>)
```

- Parameter:
  - prompt: The string we wish to display on the screen. It is optional.

  Note: The entered data is a string.

```python
age = input('Enter your age: ')
print(age)              # Assume the input is 18. It prints "18"
print(type(age))        # Print "<class 'str'>"
print(type(int(age)))   # Print "<class 'int'>"
print(type(float(age))) # Print "<class 'float'>"
age = int(age) + 1      # Convert age to int and increase it by 1
print('Now you are', age, 'years old') # Print "Now you are 19 years old"
```

Branching Statements

- Branching statements are used to execute the specific blocks of code on the basis of some condition.
- Python provides 4 types of branching statements:
  1. "if" statements, where a specific block of code is executed if a condition is met.
  2. "if-else" statements, where a specific block of code is executed if a condition is met and another block of code is executed if the condition is not met.
  3. "if-elif-else" statements, where multiple conditions are checked in sequence, and the first true condition's corresponding block of code is executed.
  4. "Nested if-else" statements, where there is an outer if statement, and inside it another if-else statement is present.

Branching Statements

Syntax: if
```
if <boolean expr>:
    <statement>
    ...
```

Syntax: if-else
```
if <boolean expr>:
    <statement>
    ...
else:
    <statement>
    ...
```

Syntax: if-elif-else
```
if <boolean expr>:
    <statement>
    ...
elif <boolean expr>:
    <statement>
    ...
else:
    <statement>
    ...
```

Syntax: Nested if-else
```
if <boolean expr>:
    <statement>
    ...
    if <boolean expr>:
        <statement>
        ...
    elif <boolean expr>:
        <statement>
        ...
    else:
        <statement>
        ...
else:
    <statement>
    ...
```

- Parameters:
  - boolean expr: An expression that returns True/False
  - statement: A programming statement

Branching Statements - if/if-else Statements

- if statement

```python
num = int(input('Enter a number: '))

if num % 2 == 0:
    print("The number is even.")
```

Output:

Enter a number: 10
The number is even.

- if-else statement

```python
num = int(input('Enter a number: '))

if num % 2 == 0:
    print("The number is even.")
else:
    print("The number if odd.")
```

Output (Two Sessions):

Enter a number: 3
The number if odd.

Enter a number: 8
The number is even.

# Branching Statements - if-else using logical operator

```python
score1 = int(input('Enter first score: '))
score2 = int(input('Enter second score: '))
score3 = int(input('Enter third score: '))

if score1 == score2 and score2 == score3:
    print("All three scores are the same")
else:
    print("Not all three scores are the same")
```

Output (Two Sessions):

```
Enter first score: 8
Enter second score: 8
Enter third score: 7
Not all three scores are the same

Enter first score: 8
Enter second score: 8
Enter third score: 8
All three scores are the same
```
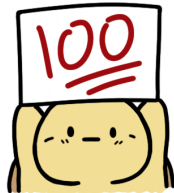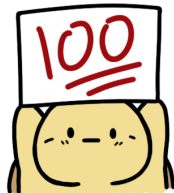
Branching Statements - if-elif-else

```python
num1 = int(input('Enter first number: '))
num2 = int(input('Enter second number: '))
num3 = int(input('Enter third number: '))

if num > num2 and num1 > num3:
    largest = num1
elif num2 > num1 and num2 > num3:
    largest = num2
else:
    largest = num3
print("The largest number is", largest)
```

Output:

```
Enter first number: 8
Enter second number: 6
Enter third number: 9
The largest number is 9
```

Branching Statements - Nested if-else statement

```python
num = int(input('Enter a number: '))

if num != 0:
   if num > 0:
      print("Number is positive")
   else:
      print("Number is negative")
else:
   print("Number is zero")
```
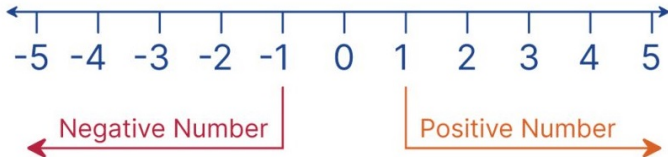
Output (3 Sessions):

```
Enter a number: 20
Number is positive

Enter a number: 0
Number is zero

Enter a number: -5
Number is negative
```

Iterative/Looping Statements

- Iterative/looping statements are used to execute a block of statements or code repeatedly for several times.
- Python provides 3 types of iterative/looping statements:
    1. while loop, where it executes a block of statements repeatedly until a given condition is false. When the condition becomes false, the line immediately after the loop in the program is executed.
    2. for loop, where it is used for iterating over a sequence, i.e., a list, a tuple, a set, a dictionary, or a string. There is "for-in" loop in Python which is similar to for-each loop in other languages.
    3. Nested loops, where a loop inside a loop.
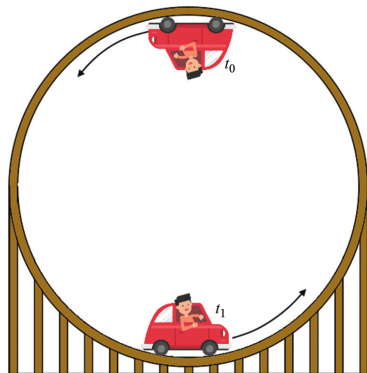
Iterative/Looping Statements

Syntax: while
```
while <boolean expr>:
   <statement>
   ...
```

Syntax: for loop
```
for <element> in <sequence>:
   <statement>
   ...
```

- Parameters:
    - boolean expr: An expression that returns True/False
    - statement: A programming statement
    - element: A variable that will store each element in the sequence
    - sequence: An iterable sequence, e.g., list, dictionary, tuple, set, string

Iterative/Looping Statements

<table>
<tr><td>

Syntax: Nested while loops

```python
while <boolean expr>:
    <statement>

    ...
    while <boolean expr>:
        <statement>

        ...
    <statement>

    ...
```

</td><td>

Syntax: Nested for loops

```python
for <element> in <sequence>:
    <statement>

    ...
    for <element> in <sequence>:
        <statement>

        ...
    <statement>

    ...
```

</td></tr>
</table>

- Parameters:
  - boolean expr: An expression that returns True/False
  - statement: A programming statement
  - element: A variable that will store each element in the sequence
  - sequence: An iterable sequence, e.g., list, dictionary, tuple, set, string

continue and break Keywords

- In Python, break and continue are loop control statements executed inside a loop.
- The continue statement causes the loop to skip its current execution at some point and move on to the next iteration.
- The break statement terminates the loop immediately and transfers execution to the new statement after the loop.
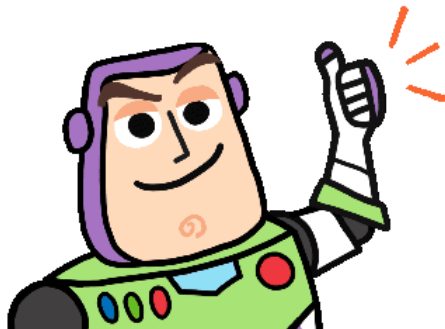
While Loop

```python
n = int(input('Enter a positive value n: '))
count = 0
while(count < n):
  count = count + 1
  print("COMP 2211 is a great COMP course")
```

Output:
```
Enter a positive value n: 5
COMP 2211 is a great COMP course
COMP 2211 is a great COMP course
COMP 2211 is a great COMP course
COMP 2211 is a great COMP course
COMP 2211 is a great COMP course
```

While Loop with continue Statement

- while loop with continue statement

```python
i = 0
while i <= 10:
    if i == 4:
        i += 1
        continue
    print("Number:", i)
    i += 1
```

Output:

```
Number: 0
Number: 1
Number: 2
Number: 3
Number: 5
Number: 6
Number: 7
Number: 8
Number: 9
Number: 10
```

While Loop with break Statement

- while loop with break statement

```
i = 0
while i <= 10:
    if i == 4:
        i += 1
        break
    print("Number:", i)
    i += 1
```

Output:

Number: 0
Number: 1
Number: 2
Number: 3

for Loop

```python
course = "COMP 2211"
for character in course:
    print(character)

languages = ["Python", "Java", "C++", "Rust"]
for lang in languages:
    print(lang)
```

Output:

```
C
O
M
P

2
2
1
1
Python
Java
C++
Rust
```

for Loop with continue Statement

```python
course = "COMP 2211"
for character in course:
    if character == " ":
        continue
    print(character)

languages = ["Python", "Java", "C++", "Rust"]
for lang in languages:
    if lang == "Java":
        continue
    print(lang)
```

Output:

```
C
O
M
P
2
2
1
1
Python
C++
Rust
```

for Loop with break Statement

```python
course = "COMP 2211"
for character in course:
    if character == " ":
        break
    print(character)

languages = ["Python", "Java", "C++", "Rust"]
for lang in languages:
    if lang == "Java":
        break
    print(lang)
```

Output:

```
C
O
M
P
Python
```

Nested for Loops

```python
print("Prime numbers in the range 2 to 100 are:")
# range returns a sequence of numbers starting from 2, increments by 1 each time,
# and stops before 101, i.e., 2, 3, 4, ..., 100
for number in range(2,101):
    isPrime = True
    # Check if number is a prime number or not
    for value in range(2,number):
        # Check if any number in the range [2, number-1] divides value completely
        if number % value == 0:
            isPrime = False # The number is not prime
            break           # Quit the loop
    if isPrime:             # If isPrime is True, print number
        print(number,end=" ")
```

Output:

```
Prime numbers in the range 2 to 100 are:
2 3 5 7 11 13 17 19 23 29 31 37 41 43 47 53 59 61 67 71 73 79 83 89 97
```

Nested while Loops

```python
# Print a 10x10 multiplication table

i = 1
while i <= 10:
  j = 1
  while j <= 10:
    print(("{:3d}".format(i * j)), end='')
    j += 1
  i += 1
  print()
```

Output:

```
  1  2  3  4  5  6  7  8  9 10
  2  4  6  8 10 12 14 16 18 20
  3  6  9 12 15 18 21 24 27 30
  4  8 12 16 20 24 28 32 36 40
  5 10 15 20 25 30 35 40 45 50
  6 12 18 24 30 36 42 48 54 60
  7 14 21 28 35 42 49 56 63 70
  8 16 24 32 40 48 56 64 72 80
  9 18 27 36 45 54 63 72 81 90
 10 20 30 40 50 60 70 80 90100
```

Containers

- Python includes several built-in container types
    - Lists
    - Dictionaries
    - Sets
    - Tuples

Container	Ordered?	Duplicate?	Indexing/ Slicing?	Changeable/ Mutable?	Constructor	Example
List	Yes	Yes	Yes	Yes	[ ] or list()	[5.7, 4, 'yes', 5.7]
Dictionary	Yes	No	Yes	Yes	{ } or dict()	{'Jun':75, 'Jul':89}
Set	No	No	No	Yes	{ } or set()	{5.7, 4, 'yes'}
Tuple	Yes	Yes	Yes	No	( ) or tuple()	(5.7, 4, 'yes', 5.7)

This table summarizes most of the properties of different containers. As soon as we've covered all the details, return to this page to read the summary.

Lists

- A list is the Python equivalent of an array, but is resizeable and can contain elements of different types.

Syntax

```
<list-name> = [<value1>, <value2>, <value3>, ...]
```

- Parameters:
    - list-name: A variable name of a list
    - value1, value2, value3, ···: List values

    Note: List literals are written within square brackets [ ].

- The list items can be accessed using index operator ([ ] – not to be confused with an empty list). The expression inside the brackets specifies the index.
    - The first element has index 0, and the last element has index (# of elements in the list - 1).
    - Negative index values will locate items from the right instead of from the left.

Lists

- The append() method adds an item to the end of the list.

Syntax

```
<list-name>.append(<item>)
```

- Parameters:
  - list-name: The list that we want to append the element to
  - item: An item (number, string, list, etc.) to be added at the end of the list

- The += operator adds a list of elements to the list

Syntax

```
<list-name> += <new-list-name>
```

- Parameters:
  - list-name: The list that we want to append the new list to
  - new-list-name: The new list of elements to add

Lists

- The pop() method removes and returns the last value from the list or the given index value.

Syntax

```
<list-name>.pop(<index>)
```

- Parameters:
    - list-name: The variable name of list that we want to remove an element
    - index: The value at index is popped out and removed. If the index is not given, then the last element is popped out and removed

- The remove() method removes the specified value from the list

Syntax

```
<list-name>.remove(<element>)
```

- Parameters:
    - list-name: The variable name of list that we want to remove an element.
    - element: The element that we want to remove. If the element does not exist, it throws an exception.

Lists

```python
xs = [3, 1, 2]      # Create a list
print(xs, xs[2])    # Print "[3, 1, 2] 2"
print(xs[-1])       # Negative indices count from the end of the list; print "2"
xs[2] = 'foo'       # Lists can contain elements of different types
print(xs)           # Print "[3, 1, 'foo']"
xs.append('bar')    # Add a new element to the end of the list
print(xs)           # Print "[3, 1, 'foo', 'bar']"
x = xs.pop()        # Remove and return the last element of the list
print(x, xs)        # Print "bar [3, 1, 'foo']"
ys = ['a', 3, [4.5, 'c', True]] # Create another list
print(ys)           # Print ['a', 3, [4.5, 'c', True]]
print(ys[2][2])     # Print "True"
ys += [4, 5, 6]     # Add 3 elements, 4, 5, 6 to ys
print(ys)           # Print ['a', 3, [4.5, 'c', True], 4, 5, 6]
ys.remove(5)        # Remove 5 from the list
print(ys)           # Print ['a', 3, [4.5, 'c', True], 4, 6]
```

More Details About Lists

https://docs.python.org/3.10/tutorial/datastructures.html#more-on-lists

Slicing

- In addition to accessing list elements one at a time, Python provides concise syntax to access sublists; this is known as slicing.

Syntax

```
<new-list-name> = <list-name>[<start>:<end(exclusive)>:<jump>]
```

- Parameters:
  - list-name: The name of list to be sliced
  - new-list-name: The name of the extracted sub-list
  - start: The index of the start element
  - end: The index of the element after the last element
  - jump: The index step jump
- It returns the portion of the list from index "start" to index "end" (exclusive), at a step size "jump".

Note: If parameters are omitted, the default value of start, end, and jump are 0, (#elements in the list), 1, respectively.

Slicing

```python
nums = list(range(5))      # range is a built-in function that creates a
                           # list of integers, [0, 1, 2, 3, 4]
print(nums)                # Print "[0, 1, 2, 3, 4]"
print(nums[2:4])           # Get a slice from index 2 to 4 (exclusive)
                           # print "[2, 3]"
print(nums[2:])            # Get a slice from index 2 to the end
                           # print "[2, 3, 4]"
print(nums[:2])            # Get a slice from the start to index 2 (exclusive)
                           # print "[0, 1]"
print(nums[:])             # Get a slice of the whole list
                           # print "[0, 1, 2, 3, 4]"
print(nums[:-1])           # Slice indices can be negative
                           # print "[0, 1, 2, 3]"
print(nums[0:4:2])         # Print "[0, 2]"
nums[2:4] = [8, 9]         # Assign a new sublist to a slice
print(nums)                # Print "[0, 1, 8, 9, 4]"
```

Loops

- You can loop over the elements of a list like this:

```python
animals = ['cat', 'dog', 'monkey']
for animal in animals:
    print(animal)  # Prints "cat", "dog", "monkey", each on its own line.

# Alternative
# for index in range(3):
#     print(animals[index])
# Prints "cat", "dog", "monkey", each on its own line.

i = 0
while i < len(animals):
    print(animals[i])
    i = i + 1        # Prints "cat", "dog", "monkey", each on its own line.
```

Dictionaries

- A dictionary stores (key, value) pairs, similar to a Map in Java or an object in Javascript.

Syntax

```
<dict-name> = { <key1>:<value1>, <key2>:<value2>, <key3>:<value3>, ... }
```

- Parameters:
  - dict-name: The name of a dictionary
  - key1, key2, key3, $\cdots$: The keys
  - value1, value2, value3, $\cdots$: The values

  Note: Dictionary literals are written within curly brackets { }.

- Items of a dictionary can be accessed by referring to its key name, inside square brackets.

Dictionaries

- Adding an item to the dictionary is done by using a new index key and assigning a value

Syntax

```
<dict-name>[<new-key>] = <new-value>
```
- Parameters:
    - dict-name: The name of a dictionary
    - new-key: The key name of the item you want to add
    - new-value: The corresponding value of the item you want to add

- The get() method returns the value of the item with the specified key.

Syntax

```
<dict-name>.get(<keyname>, <value>)
```
- Parameters:
    - dict-name: The name of a dictionary
    - keyname: The key name of the item you want to return the value from
    - value: Optional. A value to return if the specified key does not exist. Default value None

Dictionaries

- The pop() method removes the specified item from the dictionary. The value of the removed item is the return value of the pop() method.

Syntax

```
<dict-name>.pop(<keyname>, <default-value>)
```

- Parameters:
  - dict-name: The name of a dictionary
  - keyname: The key name of the item you want to remove
  - default-value: A value to return if the specified key do not exist. If this parameter is not specified, and the item with the specified key is not found, an error is raised

Dictionaries

- ```python
  d = {'cat': 'cute', 'dog': 'furry'}  # Create a new dictionary with some data
  print(d['cat'])        # Get an entry from a dictionary; prints "cute"
  print('cat' in d)      # Check if a dictionary has a given key; print "True"
  d['fish'] = 'wet'      # Set an entry in a dictionary
  print(d['fish'])       # Print "wet"
  # print(d['monkey'])   # KeyError: 'monkey' not a key of d
  print(d.get('monkey', 'N/A'))  # Get an element with a default; print "N/A"
  print(d.get('fish', 'N/A'))    # Get an element with a default; print "wet"
  del d['fish']                  # Remove an element from a dictionary
  print(d.get('fish', 'N/A'))    # "fish" is no longer a key; print "N/A"
  item = d.pop('cat')    # Remove the item with key 'cat' from the dictionary
  print(item)            # Print "cute"
  print(d)               # Print {'dog': 'furry'}
  ```

More Details About Dictionaries

https://docs.python.org/3.10/library/stdtypes.html#dict

Dictionaries

- It is easy to iterate over the keys in a dictionary:

```python
d = {'person': 2, 'cat': 4, 'spider': 8}
for animal in d:
    legs = d[animal]
    print('A %s has %d legs' % (animal, legs))
# Print "A person has 2 legs", "A cat has 4 legs", "A spider has 8 legs"
```

- If you want to access keys and their corresponding values, use the `items` method:

```python
d = {'person': 2, 'cat': 4, 'spider': 8}
for animal, legs in d.items():
    print('A %s has %d legs' % (animal, legs))
# Print "A person has 2 legs", "A cat has 4 legs", "A spider has 8 legs"
```

Sets

- A set is an unordered collection of distinct elements (i.e., no duplicates).

Syntax

```
<set-name> = { <value1>, <value2>, <value3>, ...}
```

- Parameters:
    - set-name: The name of a set
    - value1, value2, value3, $\cdots$: Set values

  Note: Set literals are written within curly brackets { }.

- A set is unchangeable (i.e., cannot change the items after the set has been created), and unindexed. But we can add new items and remove items.

Sets

- The add() method adds a given element to a set if the element is not present in the set.

Syntax

```
<set-name>.add(<element>)
```
- Parameters:
  - set-name: The name of a set that we want to add an element to
  - element: The element that we want to add

- The remove() method removes the specified element from the set.

Syntax

```
<set-name>.remove(<element>)
```
- Parameters:
  - set-name: The name of a set that we want to remove an element from
  - element: The element that we want to remove

Sets

- As a simple example, consider the following:

```python
animals = {'cat', 'dog'}
print('cat' in animals)    # Check if an element is in a set; prints "True"
print('fish' in animals)   # Print "False"
animals.add('fish')        # Add an element to a set
print('fish' in animals)   # Print "True"
print(len(animals))        # Number of elements in a set; prints "3"
animals.add('cat')         # Adding an element that is already in the set
                           # does nothing
print(len(animals))        # Print "3"
animals.remove('cat')      # Remove an element from a set
print(len(animals))        # Print "2"
```

More Details About Sets

https://docs.python.org/3.10/library/stdtypes.html#set

Sets

- Loops: Iterating over a set has the same syntax as iterating over a list; however since sets are unordered, you cannot make assumptions about the order in which you visit the elements of the set:

```python
animals = {'cat', 'dog', 'fish'}
for idx, animal in enumerate(animals):
    print('#%d: %s' % (idx + 1, animal))
# Prints
# #1: cat
# #2: dog
# #3: fish
```

Tuples

- A tuple in an (immutable/unchangeable) ordered list of values.

Syntax

```
<tuple-name> = (<value1>, <value2>, <value3>, ...)
```

- Parameters:
  - tuple-name: The variable name of a tuple that we want to add element(s) to
  - value1, value2, value3, ⋯: The list of values that we want to add to the tuple

  Note: Tuple literals are placed inside parentheses (), separated by commas.

- The tuple items can be accessed using index operator ([ ] – not to be confused with an empty list). The expression inside the brackets specifies the index.
  - The first element has index 0, and the last element has index (#elements in the tuple - 1).
  - Negative index values will locate items from the right instead of from the left.

Tuples

- A tuple is in many ways similar to a list; one of the most important differences is that tuples can be used as keys in dictionaries and as elements of sets, while lists cannot.
- Here is a trivial example:

```python
d = {(x, x + 1): x for x in range(10)}  # Create a dictionary with tuple keys
t = (5, 6)              # Create a tuple
print(type(t))          # Print "<class 'tuple'>"
print(d[t])             # Print "5
print(d[(1, 2)])        # Print "1
```

More Details About Tuples

https://docs.python.org/3.10/tutorial/datastructures.html#tuples-and-sequences

Functions

- Python functions are defined using def keyword.
- The return keyword is to exit a function and return a value.
- Python function body cannot be empty. But for some reason, you need to define a function with empty body, the pass statement has to be put in it to avoid getting an error.

Syntax

```
def <function-name>(<arguments>):
    <statement>
    ...
    return <ret-value>
```

- Parameters:
  - function-name: The name of the function
  - argument: Value(s) passed to the function
  - statement: Any programming statement
  - ret-value: The returning value to the function caller

```
# The following shows a
# function with empty
# function body

def a_function_with_empty_body:
    pass
```

Example

```python
# Define a function sign with argument x
def sign(x):
    if x > 0:
        return 'positive'
    elif x < 0:
        return 'negative'
    else:
        return 'zero'

for x in [-1, 0, 1]:
    print(sign(x))  # Call the function sign
```

Output:

```
negative
zero
positive
```

Functions

- Normally, when we create a variable inside a function, that variable is local, and can only be used inside that function.
- To create a global variable inside a function, we can use the global keyword.

```python
def assign_word():
  global word
  word = "cool"

assign_word()

print("Desmond is really " + word)
```

Output:

```
Desmond is really cool
```

Functions

- Function can be defined with arguments that have default values.

```python
def hello(name, loud=False):
    if loud:
        print('HELLO, %s!' % name.upper())
    else:
        print('Hello, %s' % name)

hello('Bob')              # Print "Hello, Bob"
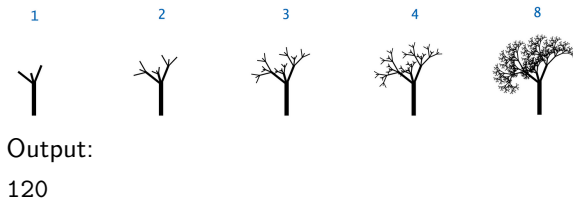hello('Fred', loud=True)  # Print "HELLO, FRED!"
```

More Details About Functions

https://docs.python.org/3.10/tutorial/controlflow.html#defining-functions

Recursion

- A recursion function is one that calls itself.
- Every recursive function definition includes two parts:
    1. Base case(s) (non-recursive)
       One or more simple cases that can be done right away.
    2. Recursive case(s)
       One or more cases that require solving "simpler" version(s) of the original problem.
         - "Simpler" means "smaller" or "shorter" or "closer to the base case".

```python
def factorial(n):
    if n == 0: # base case
        return 1
    else:      # recursive case
        return n * factorial(n-1)

print(factorial(5))
```

Output:

120

That's all!

Any questions?