Introduction to Computer Programming

COMS 127

Hebi Li

Outline

Team

- COMS 127: Introduction to Computer Programming
- Instructor: Prof. Forrest Bao

Team

- COMS 127: Introduction to Computer Programming
- Instructor: Prof. Forrest Bao

- Hebi Li (me): random person, 1st week lecture

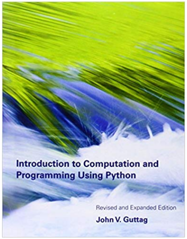
Team

- COMS 127: Introduction to Computer Programming
- Instructor: Prof. Forrest Bao

- Hebi Li (me): random person, 1st week lecture
- TAs: TBD

Syllabus

- Meating times: Monday, Wednesday, Friday 2:10-3:00 PM
- Location: Food Science 2432
- Textbook: Guttag, John V. *Introduction to computation and programming using Python*. Mit Press, 2013.



Plan for this week

Monday: Introduction to programming

- Intro to Programming
- Programming history & trends
- Introduction to python

Wednesday: Setup python enviroment

Bring your laptop!

- Editors
- IDEs
- Online interpreters: Jupyter notebooks

Friday: expressions and variables

TOC

Have you ever writing any programs?

If so, which is your favorite, and why?

Have you ever writing any programs?

If so, which is your favorite, and why?

"Main stream"

- ☐ C/C++
- ☐ Java
- ☐ Javascript
- ☐ python

Have you ever writing any programs?

If so, which is your favorite, and why?

"Main stream"

- ☐ C/C++
- ☐ Java
- ☐ Javascript
- ☐ python

Other options

- ☐ scripting languages: perl, ruby, shell script
- ☐ Lisp dialect: common lisp, clojure, scheme
- ☐ Domain specific: R
- ☐ shining & new languages: rust, go, julia

What operating system do you use

- ☐ Linux
- ☐ Windows
- ☐ Mac OS

What editor do you use?

- ☐ notepad
- ☐ Sublime text
- ☐ Atom
- ☐ Eclipse
- ☐ VS Code
- ☐ Emacs
- ☐ Vim
- ☐ Online editors (e.g. jupyter notebook)

TOC

What is a program?

wikipedia definition

A computer program is a collection of instructions that performs a specific task when executed by a computer.

...

A computer program is usually written by a computer programmer in a programming language.

...

From the program in its human-readable form of source code, a compiler or assembler can derive machine code. ... Alternatively, a computer program may be executed with the aid of an interpreter.

Hello World!

Hello world:

```
print ("Hello World")
```

Hello World!

Hello world:

```python
print ("Hello World")
```

`print` is a library call. You can print anything out in python:

```python
print(1)
print(1+2)
1+2
```

Major components of computer programs

Basic ingredients

- variables
- expressions
- statements

Major components of computer programs

Basic ingredients

- variables
- expressions
- statements

Control flow

- conditional: if .. else ..
- loop: for-loop

Major components of computer programs

Basic ingredients

- variables
- expressions
- statements

Control flow

- conditional: if .. else ..
- loop: for-loop

data structures

- list
- hash tables (dictionaries)

Major components of computer programs

Basic ingredients

- variables
- expressions
- statements

Abstraction

- Functions
- Classes

Control flow

- conditional: if .. else ..
- loop: for-loop

data structures

- list
- hash tables (dictionaries)

Major components of computer programs

Basic ingredients

- variables
- expressions
- statements

Control flow

- conditional: if .. else ..
- loop: for-loop

data structures

- list
- hash tables (dictionaries)

Abstraction

- Functions
- Classes

project organization

- modules & files

Major components of computer programs

Basic ingredients

- variables
- expressions
- statements

Control flow

- conditional: if .. else ..
- loop: for-loop

data structures

- list
- hash tables (dictionaries)

Abstraction

- Functions
- Classes

project organization

- modules & files

libraries

- stardard libraries
- 3rd-party libraries

Basic ingredients

- variables
- expressions
- statements

```
pi = 3.14              # variable: pi
r = 10
pi * r * r             # expression, evaluates to 314
print(pi * r * r)
area = pi * r * r      # assignment statement
print(area)            # print statement
```

Control flow

- conditional: if .. else ..
- loop: for-loop

Control flow

- conditional: if .. else ..
- loop: for-loop

conditional

```
a = -3
if a > 0:
  print('a is positive')
elif a < 0:
  print('a is negative')
else:
  print('a = 0')
```

Control flow

- conditional: if .. else ..
- loop: for-loop

conditional

```
a = -3
if a > 0:
  print('a is positive')
elif a < 0:
  print('a is negative')
else:
  print('a = 0')
```

loop

```
for i in range(2, 10):
  print(i)

for i in range(2, 10):
  if is_prime(i):
    print(i)
```

data structures

- list
- hash tables (dictionaries)

data structures

- list
- hash tables (dictionaries)

list
```
l = [10, 8, 9, 1, 5, 6]
print(l[0])   # => 10
print(l[1])   # => 8
print(l[2:4]) # => [9,1]
print(len(l)) # => 6
print(sorted(l))
# => [1, 5, 6, 8, 9, 10]
```

data structures

- list
- hash tables (dictionaries)

list
```
l = [10, 8, 9, 1, 5, 6]
print(l[0])   # => 10
print(l[1])   # => 8
print(l[2:4]) # => [9,1]
print(len(l)) # => 6
print(sorted(l))
# => [1, 5, 6, 8, 9, 10]
```

dict
```
d = dict()
d["hello"] = "world"
d[2] = "two"
d["five"] = 5
print(d["hello"])
# => "world"
```

Functions

statements

```
pi = 3.14
r = 10
area = pi * r * r
print(area)

r2 = 11
area2 = pi * r2 * r2
r3 = 12
area3 = pi * r3 * r3
```

Functions

statements

```
pi = 3.14
r = 10
area = pi * r * r
print(area)

r2 = 11
area2 = pi * r2 * r2
r3 = 12
area3 = pi * r3 * r3
```

- For code reuse and abstraction.

Functions

statements
```
pi = 3.14
r = 10
area = pi * r * r
print(area)

r2 = 11
area2 = pi * r2 * r2
r3 = 12
area3 = pi * r3 * r3
```

- For code reuse and abstraction.

function
```
def circle_area(r):
    pi = 3.14
    return pi * r * r

for r in range(10):
  print(circle_area(r))
```

Classes: object-oriented programming

Class consists of

- fields: class local variables
- methods: functions to operate on the fields

Classes: object-oriented programming

Class consists of

- fields: class local variables
- methods: functions to operate on the fields

```
class Shape():
    def area(self):
        return 0
```

Classes: object-oriented programming

Class consists of

- fields: class local variables
- methods: functions to operate on the fields

```
class Shape():
    def area(self):
        return 0
```

```
class Circle():
    def __init__(self, r):
        self.r = r
    def area(self):
        return (3.14 * self.r
                * self.r)
```

Classes: object-oriented programming

Class consists of

- fields: class local variables
- methods: functions to operate on the fields

```
class Shape():
    def area(self):
        return 0
```

```
class Circle():
    def __init__(self, r):
        self.r = r
    def area(self):
        return (3.14 * self.r
                * self.r)
```

```
class Rectangle():
    def __init__(self, w, h):
        self.w = w
        self.h = h
    def area(self):
        return self.w * self.h
```

project organization

- modules & files

lib.py
```python
def area(r):
    pi = 3.14
    return pi * r * r
def perimeter(r):
    pi = 3.14
    return 2 * r * pi
```

project organization

- modules & files

lib.py

```python
def area(r):
    pi = 3.14
    return pi * r * r
def perimeter(r):
    pi = 3.14
    return 2 * r * pi
```

use.py

```python
from lib import area
from lib import perimeter
r = 10
print('Area: ', area(r))
print('Perimeter: ',
      perimeter(r))
```

Python standard library [1]

- numerical: `math`, `random`
- File IO: `os`, `io`
- dataset: `pickle`, `sqlite3`
- GUI: `tkinter`
- multi-thread: `threading`, `subprocess`
- network: `socket`, `asyncio`
- web: `html`, `xml`, `http`

[1] https://docs.python.org/3/library/

Recap: program components

Basic ingredients

- variables
- expressions
- statements

Control flow

- conditional: if .. else ..
- loop: for-loop

data structures

- list
- hash tables (dictionaries)

Abstraction

- Functions
- Classes

project organization

- modules & files

libraries

- stardard libraries
- 3rd-party libraries

TOC

Early days (50s-60s)

- 1957: FORTRAN 

- 1958: Lisp-1.5: code is data
- 1958 ALGOL
  - ALGOL 58
  - ALGOL 60
  - ALGOL 68
- 1959 COBOL
- 1964 BASIC
- 1966 APL

Early days (50s-60s)

- 1957: FORTRAN 

- 1958: Lisp-1.5: code is data
- 1958 ALGOL
  - ALGOL 58
  - ALGOL 60
  - ALGOL 68
- 1959 COBOL
- 1964 BASIC
- 1966 APL

Lisp is worth learning for the profound enlightenment experience you will have when you finally get it; that experience will make you a better programmer for the rest of your days, even if you never actually use Lisp itself a lot.

– Eric Raymond, in *How to Become a Hacker*

1970s: Establishing fundamental paradigms

- 1970: Pascal, in ALGOL family
- 1972: C
    - 1973: Unix V4 rewritten in C
- 1972: Smalltalk
- 1972: Prolog
- 1973: ML, "Lisp with types"
- 1975 Scheme: cleanly designed Lisp
    - proper lexical scoping
    - hygenic macros
- 1978 SQL

1980s

- 1980: C++
  - add object-oriented to C
- 1980: Ada
- 1984: Common Lisp
- 1984: Matlab
- 1986: Erlang
- 1986: Labview
  - graphical programming
- 1986: Perl
- 1988: Tcl

1980s

- 1980: C++
  - add object-oriented to C
- 1980: Ada
- 1984: Common Lisp
- 1984: Matlab
- 1986: Erlang
- 1986: Labview
  - graphical programming
- 1986: Perl
- 1988: Tcl

Philip Greenspun:

> *Any sufficiently complicated C or Fortran program contains an ad hoc informally-specified bug-ridden slow implementation of half of Common Lisp.*

1990s: the Internet age, developer productivity

- 1990: Haskell
    - pure-functional language
    - Monad
- 1991: Python
    - scripting made easier
- 1991: Visual Basic
- 1993: Lua
- 1993: R
- 1995: Ruby
- 1995: Java
    - most popular for 2 decades
- 1995: PHP
- 1995: Javascript

1990s: the Internet age, developer productivity

- 1990: Haskell
  - pure-functional language
  - Monad
- 1991: Python
  - scripting made easier
- 1991: Visual Basic
- 1993: Lua
- 1993: R
- 1995: Ruby
- 1995: Java
  - most popular for 2 decades
- 1995: PHP
- 1995: Javascript

Atwood's Law: any application that can be written in JavaScript, will eventually be written in JavaScript.

2000-2010

- 2001: C#
- 2001: D
- 2002: scratch Demo: `https://scratch.mit.edu`
- 2003: scala
- 2005: F#
- 2007: Clojure
- 2009: Go

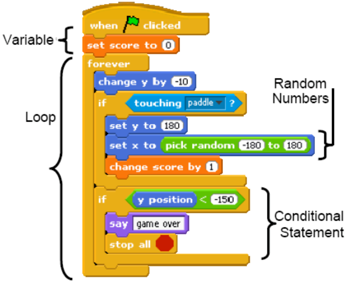


Figure: a Scratch program

2010+

- 2010: Rust: memory safety
- 2011: Dart: cross platform (Apple, Android, Web)
- 2011: Kotlin: can we fix Java?
- 2011: Elixir
- 2012: Julia: machine learning, the new R
- 2012: Typescript: Javascript with static typing
- 2014: Swift

2010+

- 2010: Rust: memory safety
- 2011: Dart: cross platform (Apple, Android, Web)
- 2011: Kotlin: can we fix Java?
- 2011: Elixir
- 2012: Julia: machine learning, the new R
- 2012: Typescript: Javascript with static typing
- 2014: Swift

What will be the new python in 5 years?

Lisp family

- 1958: lisp 1.5
  - 1975: scheme
    - 1995: R5RS
    - 2005: R6RS
    - 2010: R7RS
  - 1984: common lisp, 1994 ANSI CL with CLOS
  - 1985: Emacs Lisp
  - 1993: GNU guile
  - 1995: racket
  - 2007: Clojure

TOC

TIOBE Index [2]

Aug 2019	Aug 2018	Programming Language	Ratings
1	1	Java	16.028%
2	2	C	15.154%
3	4	Python	10.020%
4	3	C++	6.057%
5	6	C#	3.842%
6	5	Visual Basic .NET	3.695%
7	8	JavaScript	2.258%
8	7	PHP	2.075%
9	14	Objective-C	1.690%
10	9	SQL	1.625%
24		Dart	0.715%
28		Rust	0.450%
29		Scratch	0.448%
34		Lisp	0.362%
39		Julia	0.279%
49		Haskell	0.174%

[2] Data from search engine. `https://www.tiobe.com/tiobe-index/`.

TIOBE Index history

PL	2019	2014	2009	2004	1999	1994	1989
Java	1	2	1	1	14	-	-
C	2	1	2	2	1	1	1
Python	3	7	5	7	24	21	-
C++	4	4	3	3	2	2	2
VB .NET	5	9	-	-	-	-	-
C#	6	5	6	6	19	-	-
JavaScript	7	8	8	8	16	-	-
PHP	8	6	4	5	-	-	-
SQL	9	-	-	89	-	-	-
Objective-C	10	3	31	38	-	-	-
Perl	16	11	7	4	3	10	22
Lisp	32	13	19	13	12	5	3
Pascal	220	16	14	88	6	3	20

github language stats [3]

Ranking	PL	Percentage	change
1	JavaScript	19.922%	(-2.425%)
2	Python	17.803%	(+1.519%)
3	Java	10.482%	(+0.591%)
4	Go	7.916%	(+0.295%)
5	C++	7.253%	(+0.167%)
6	Ruby	6.296%	(-0.249%)
7	PHP	5.515%	(-0.285%)
8	TypeScript	5.415%	(+0.641%)
9	C#	4.001%	(+0.659%)
10	C	3.190%	(+0.248%)
14	Rust	0.714%	(-0.062%)
15	Kotlin	0.656%	(+0.113%)
20	Dart	0.376%	(+0.055%)
24	Clojure	0.271%	(-0.091%)
28	Haskell	0.190%	(-0.071%)
37	Julia	0.068%	
48	Common Lisp	0.024%	

[3] https://madnight.github.io/githut/

TOC

Python

Table: Python language feature

Features	Option 1	Option 2
Compilation model	interpreted	compiled
Typing checking	strong	weak
Variable typing	static	dynamic
Scoping	lexical	dynamic
Paradigm	procedual	functional
Pure-functional	pure	impure
Memory model	manual	garbage collected
Object-Oriented	Yes	No

Compiler vs. Interpreter

Compiler

source code compiled to object code then runs

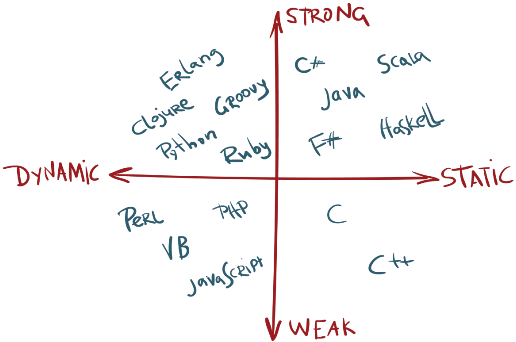
- check errors during compilation
- whole-program optimization, thus run fast

Interpreter

source code, directly run

- Interactive
- Incremental development, thus develop fast

strong/weak & static/dynamic Typing



- Strong/Weak typing: Whehter type safety is enforced on values.
- Static/Dynamic typing: Whether variables are typed.

strong typing & weak typing

Whehter type safety is enforced on values.

Strong typing: python

```
1 + "hello"
```

strong typing & weak typing

Whehter type safety is enforced on values.

Strong typing: python

```
1 + "hello"
```

TypeError: unsupported operand type(s) for +: 'int' and 'str'

strong typing & weak typing

Whehter type safety is enforced on values.

Strong typing: python

```
1 + "hello"
```

TypeError: unsupported operand type(s) for +: 'int' and 'str'

Weak typing: Javascript

```
1 + "hello"
```

strong typing & weak typing

Whehter type safety is enforced on values.

Strong typing: python

```
1 + "hello"
```

TypeError: unsupported operand type(s) for +: 'int' and 'str'

Weak typing: Javascript

```
1 + "hello"
```

"1hello"

static & dynamic typing

Whether variables are typed.

static typing: C

```
int var;
var = 8;
var = "hello";  // compile error
```

static & dynamic typing

Whether variables are typed.

static typing: C

```c
int var;
var = 8;
var = "hello";   // compile error
```

dynamic typing: python

```python
var = 8
var = "hello"
```

memory safety & management

- manual memory management
- automatic memory management: gabbage collection
- ownership (Rust)

memory safety & management

- manual memory management
- automatic memory management: gabbage collection
- ownership (Rust)

Manual memory management: C

```c
int *var;
var = (int*) malloc(sizeof(int));
*var = 5;
free(var);
```

memory safety & management

- manual memory management
- automatic memory management: gabbage collection
- ownership (Rust)

Manual memory management: C

```c
int *var;
var = (int*) malloc(sizeof(int));
*var = 5;
free(var);
```

Gabbage collected: python

```python
l = [1,2,3]
l.append(4)
print(l)      # => [1,2,3,4]
```

functional & procedual programming

functional:

- function is a first-class citizen
  - *first-class* means a function can appear anywhere value can occur
  - can be assigned to variable, pass to a function, or returned by a function
- function in *mathmatical sense*: given input, compute output. There is no side effect.

```python
var = 8
def bad_square(a):
    global var
    var = a
    return a * a
```

```python
print(var) # => 8
print(bad_square(3))
# => 9
print(var) # => 3
```

TOC

Prime number

A prime number (or a prime) is a natural number greater than 1 that cannot be formed by multiplying two smaller natural numbers.

- composite number: 6 is not prime, because 6 = 2 * 3
- list of prime numbers: 2, 3, 5, 7, 11, 13, ...

```python
def is_prime(n):
  for i in range(2, round(n/2)+1):
    if n % i is 0:
      return False
  return True

for i in range(2, 10):
  if is_prime(i):
    print(i)
```

Fibonacci number

$$
\begin{aligned}
F_0 &= 1 \\
F_1 &= 1 \\
F_n &= F_{n-1} + F_{n-2} \qquad \text{for n} > 1
\end{aligned}
$$

Example: 1, 1, 2, 3, 5, 8, 13

Fibonacci number

$$F_0 = 1$$
$$F_1 = 1$$
$$F_n = F_{n-1} + F_{n-2} \qquad \text{for n} > 1$$

Example: 1, 1, 2, 3, 5, 8, 13

```
def fib(n):
  if n is 0: return 1
  if n is 1: return 2
  return fib(n-1) + fib(n-2)

for i in range(10):
  print(fib(i))
```

Trainning a neural network classifier

Online runtime: `https://colab.research.google.com`
Import libraries:

```
import keras
from keras.layers import Input, Dense, Flatten
from keras.models import Model
from keras.datasets import mnist
import random
import matplotlib.pyplot as plt
import numpy as np
```

Trainning a neural network classifier

Setup model:

```
inputs = Input(shape=(28,28,))
x = Flatten()(inputs)
x = Dense(64, activation='relu')(x)
x = Dense(64, activation='relu')(x)
predictions = Dense(10, activation='softmax')(x)
model = Model(inputs, predictions)
model.compile(optimizer='rmsprop',
              loss='categorical_crossentropy',
              metrics=['accuracy'])
```

Trainning a neural network classifier

Load MNIST data and train the model:

```
(x_train, y_train), (x_test, y_test) = mnist.load_data()
y_train = keras.utils.to_categorical(y_train, 10)
y_test = keras.utils.to_categorical(y_test, 10)

model.fit(x_train, y_train, epochs=10)
model.evaluate(x_test, y_test)
```

Trainning a neural network classifier

Visualization:

```
index = random.randint(0, 1000)
plt.imshow(x_train[index])
plt.savefig("image.jpg")
pred = np.argmax(
    model.predict(
        np.array([x_train[index]]))[0])
print("prediction: ", pred)
```