



Massive Information &
Knowledge Engineering

Version 2014

Introduction to Computer & Programming

204113 Computer & Programming

Dr. Arnon Rungsawang
Dept. of computer engineering
Kasetsart University

<https://mike.cpe.ku.ac.th/204113>



01204113 Computer & Programming for CPE_KU

3

The age of computing

- Computers are everywhere.



From: <https://pixabay.com/en/network-iot-internet-of-things-782707/>



01204113 Computer & Programming for CPE_KU



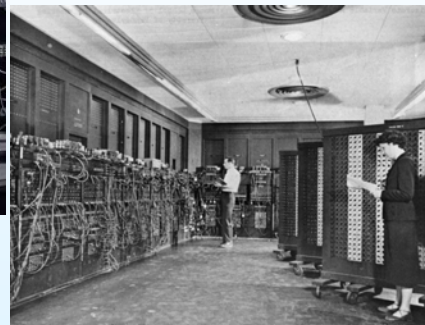
By Marc van der Chijs at <https://www.flickr.com/photos/chijs/21798665468>



By Philip Wilson (BY-ND) at <https://www.flickr.com/photos/internetsense/9900738813/>

2

What "computers" used to be..



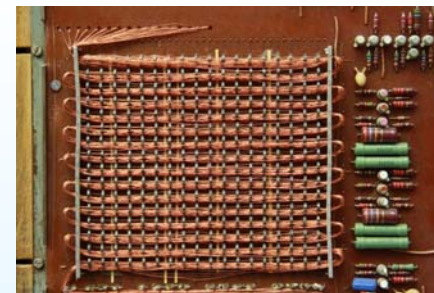
<https://penntoday.upenn.edu/news/worlds-first-general-purpose-computer-turns-75> (~ 1943)



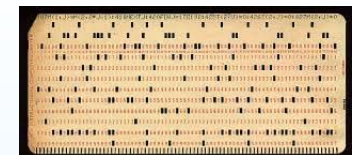
01204113 Computer & Programming for CPE_KU

3

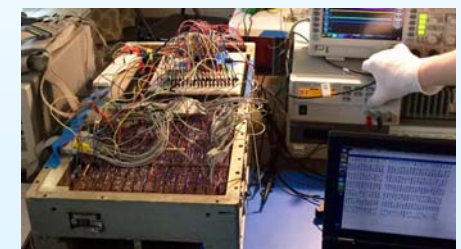
What "computers" used to be.. (2)



<https://www.galvanize.com/blog/todays-computers-vs-the-apollo-11-moon-landing-machine/> (~ 1969)



<https://alicklstory.com/2016/04/10/programming-the-guidance-systems-for-apollo/>



<https://hackaday.com/2018/11/12/an-apollo-guidance-computer-laid-bare/>



01204113 Computer & Programming for CPE_KU

4

Computer programming

- Programming – an act of developing computer programs.
- What is a computer program?



A computer program



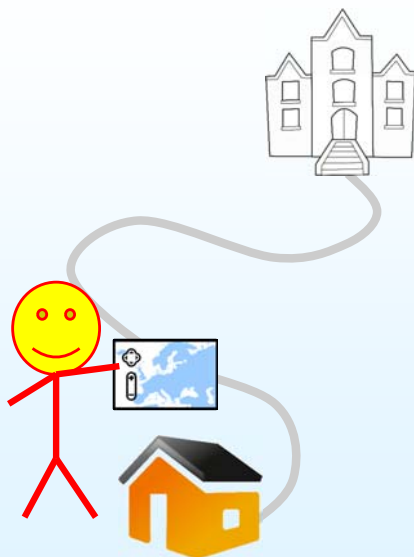
Public domain image from
[https://en.wikipedia.org/wiki/Margaret_Hamilton_\(scientist\)/#media/File:Margaret_Hamilton.gif](https://en.wikipedia.org/wiki/Margaret_Hamilton_(scientist)/#media/File:Margaret_Hamilton.gif)

- Margaret Hamilton with the computer program that took Apollo 11 to the moon.
- You can read the code at:
<https://github.com/chrislgarry/Apollo-11>

```
129 # Page 348T
130 CS  POSHOK # ASCENT (ON ON LUNAR SURFACE)
131 TS  -2387LON # ALWAYS 2 SETS FOR P-AXES RATE FORWARD
132 CAP  OCT14 # INITIALIZE INDEX AT 12.
133 TS  PHAC
134 CS  LEPPASS # CHECK IF PASS TOO HIGH. CATCH STAGNE.
135 AD  HEADCENT
136 EXTEND
137 B2NF  PASSOFF
138 CS  LEPPASS # CHECK IF PASS TOO LONG. THIS LIFETS THE
139 AD  LOASCENT # DECREMENTING BY PASSION.
140 EXTEND
141 B2NF  F(PASS)
```



From home to school



- To understand how computer works, let's try to make an analogy with how people solve some problem.
- **Problem:** It's the first day of school. You want to go to KU from your home. What do you have to do?
 - Assume that your home is close to KU, so you decide to walk to KU.



Walking from home to school

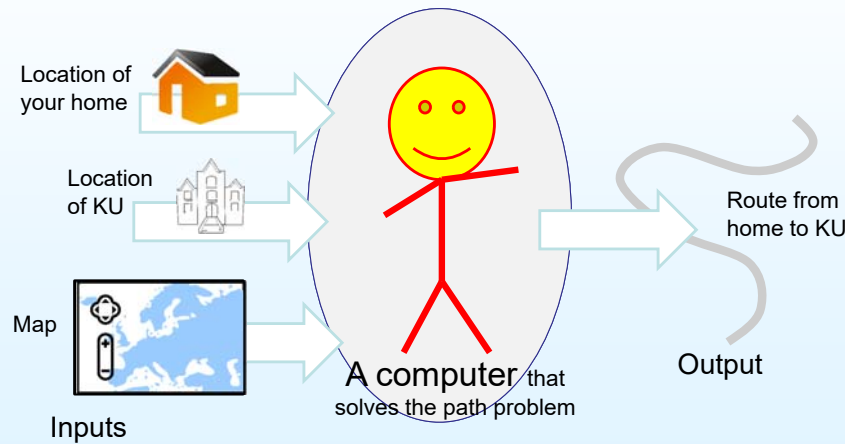


- If you know the way to KU, you can just walk. But if you don't you may want to look at **the map** and use it to plan your route to KU.
- Note that if you can **plan your walking route with a map**, you can solve this kind of problems not just for going from your home to KU, but from any place to any other place.



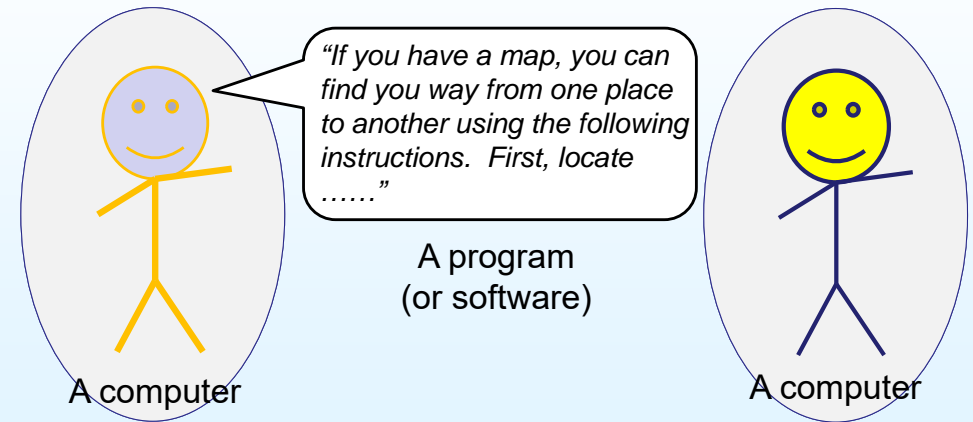
A computer, inputs, and outputs

- In a way, you are a computer.



A program

- Can you teach other people to solve the same problem?



Formal and Natural languages

Natural languages

- The languages people speak
- Have to figure out what the structure of the sentence is, this process is called **parsing**
- Full of Ambiguous
- Lots of Redundancy
- English, Spanish, Thai

Formal languages

- Designed by people for specific applications
- Strict rules about syntax
- Nearly unambiguous
- Less redundant
- Chemical structure : H_2O
Math Equation : $3+3=6$
Computer Language



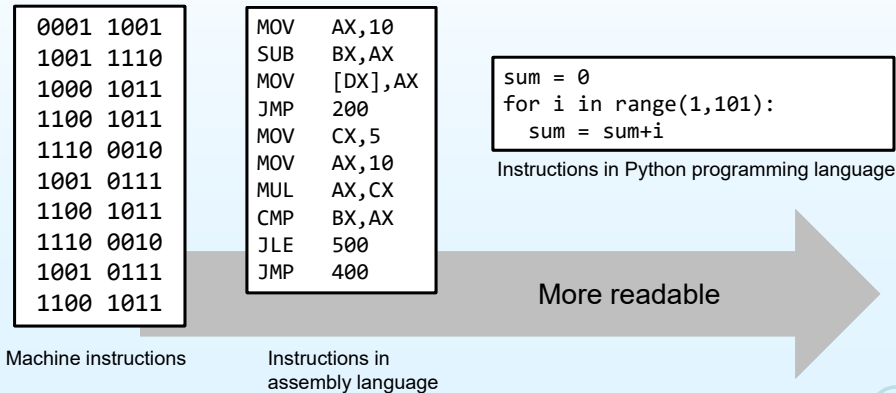
Why do we need a computer language?

- Problem Solving** : The ability to formulate problems, think creatively about solutions, and express a solution clearly and accurately.

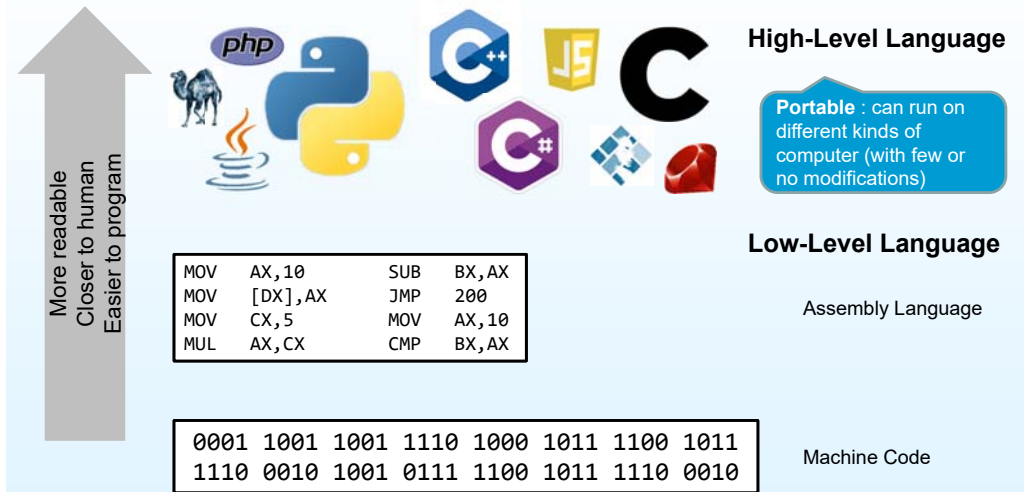


A computer program

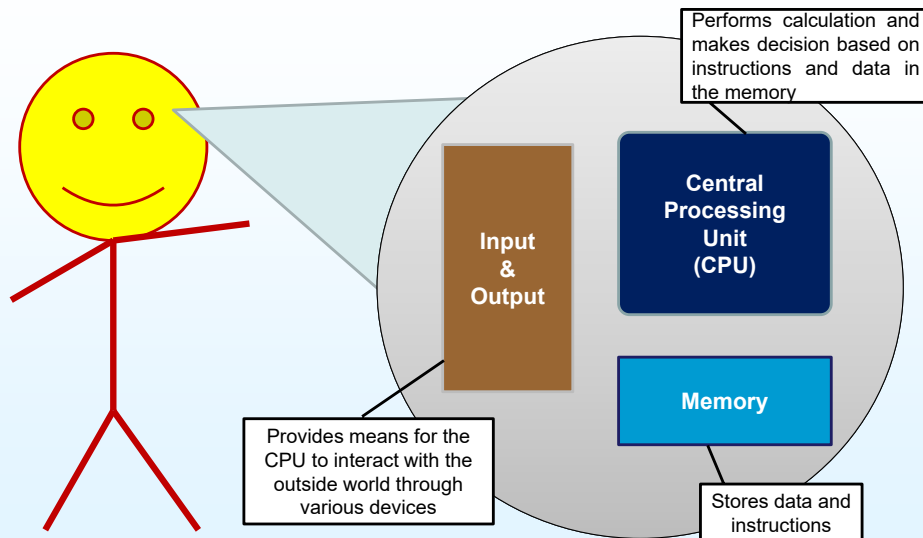
- A **computer program** is a sequence of **instructions** to be executed by computers.
- Examples of computer programs in various forms:



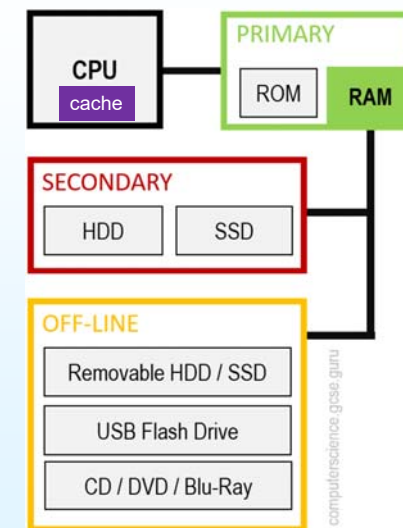
Programming Language



How computer works, abstractly



Memory



Inside the memory

- The smallest unit of information that can be processed by digital computers is a single binary digit (a **bit**), which can be either 0 or 1.
- We usually group them in groups of 8 bits, each called a **Byte**.
- A lot of bytes can be stored in a memory unit.

0 1
Two bits

0 1 0 1 1 1 0 0
One byte

- 1 kB = 1,000 bytes
- 1 MB = 1,000,000 bytes
- 1 GB = 1,000,000,000 bytes

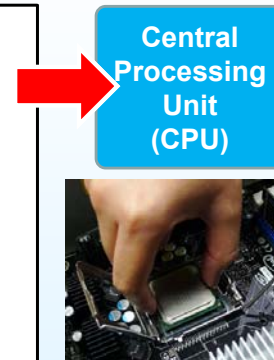
IEEE-1541 recommendation

- 1 kiB = 2^{10} bytes = 1,024 bytes
- 1 MiB = 2^{20} bytes = 1,024 kiB
- 1 GiB = 2^{30} bytes = 1,024 MiB



The instructions

```
0001 1001
1001 1110
1000 1011
1100 1011
1110 0010
1001 0111
1100 1011
1110 0010
1001 0111
1100 1011
```

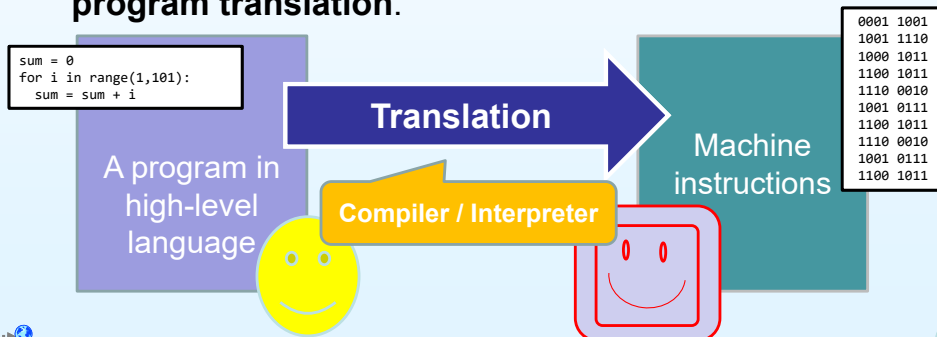


- The **memory**, not only keeps the data to be processed with the CPU, but it also keeps the instructions.
- These instructions are in the format that the **CPU** can easily understand, referred to as “**machine instructions**.”
- When writing a program, we rarely write in machine instructions.

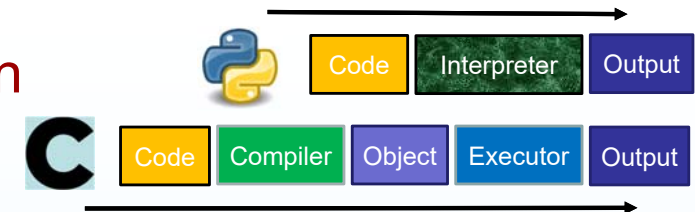


From programs to instructions

- Instead of working directly with machine instructions, people usually develop software with higher-level programming languages.
- But the program must be translated into a form that the computer can understand. This process is called **program translation**.



Translation



- Interpreter**
 - Translates program one statement at a time
 - Less time to analyze but overall execution time is slower
 - Continues translating the program until meet the first error
- Compiler**
 - Scan the entire program and translates it as whole into machine
 - Take large time to analyze the source code but execution time is faster
 - Generates the error message only after scanning the whole program



Syntax - Semantics

- **Syntax:** Structure of a program.
- **Semantics:** Meaning of a program.

I love programming	love I programming	Programming loves me
Correct Syntax and Semantics	Programming love I	Correct Syntax Wrong Semantics
	I programming love	Wrong Syntax Wrong Semantics



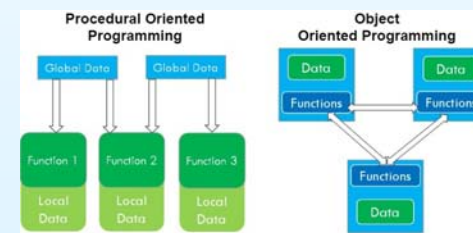
Programming Paradigm

Procedural oriented program

- Paradigm that uses a list of instructions to tell the computer **what to do step-by-step** (i.e., **top-down approach**).
- It relies on **procedures**, a.k.a. subroutines, containing a series of computational steps to be carried out.
- Languages: C, Fortran, Cobol, Pascal, ...

Object oriented program

- Approach to problem-solving where all computations are carried out using **objects**.
- Object is a component of a program that knows how to perform certain **actions** and how to interact with other **elements** of the program.
- Languages: C#, Java, ...



What is Python?

- Python is a programming language coming with simple syntax and a lot of tools.



History of Python

- Guido van Rossum creates Python when he was a Google employee.
- His first intention is to create a programming language that is
 - equipped with tools to help in doing stuffs
 - **as easy to read as plain English**



Python is good, Python is bad

Python is good

- Equipped with tools and libraries to help in doing stuffs.
- Plain and easy-to-understand syntax, as clear as plain English.
- Python expands its field very far: from **web framework** to **machine learning**.

Python is bad

- Lack fundamental concepts like datatypes and three-part for loop
- Dynamic typed language got both pros and cons
- Interpreter-based language means Python performance is much slower than compiler-based (like C)



Python coding mode

• Interactive mode

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

The chevron, >>>, is the prompt the interpreter uses to indicate that it is ready. If you type 1+1, the interpreter replies 2.

• Script mode

```
$ echo print("Hello World")>hello.py
$ python hello.py
Hello World
```

→ print("Hello World")
hello.py



Your first Python program

- Try writing the code below and running it.

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

Output

```
Hello World
```

- The command is understandable, and its task could be guessed, even for those without programming knowledge.



Printing

- **print** function is for printing texts to the screen.

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

Output

```
Hello World
```

- Every **print** function will make a new line after execution.

```
print("Hello CPE students")
print("Welcome to 111")
```

Output

```
Hello CPE students
Welcome to 111
```



Printing

- One print command may accept many strings

```
print("same", "line")
```

Output

```
same line
```

- Separated with comma, texts will be printed in the same line and separated with space.



Printing

- This is how you print texts to one single line with multiple print commands:

```
print("Kasetsart", end=" ")  
print("University")
```

Output

```
Kasetsart University
```

- We can use **end** to tell print function that what character will be used instead of new line (new line is a character too).



Escape sequence

- Some character can cause problems in programs. For example,

```
'I'm Engineer'
```

- Python thinks that single quote mark in **'I'm'** ends string, we can fix it by adding backslash (\) before quote.

```
'I\'m Engineer'
```

- Characters with backslash are called **escape sequence**.



Escape sequence

Escape sequence	Meaning
\\	Backslash (\)
\'	Single quote
\"	Double quote
\n	New line
\t	Tab space



Exercise

- Write a program to introduce yourself by using this pattern.
- Only one print function is allowed.

```
tab space My name is your name.  
tab space I'm your age years old.  
tab space Nice to meet you, CPE friends  
  
tab space \\ \\ \\ \\ "Lorem ipsum dolor sit amet" / \\ / \\ /
```



Variable

- Variable is used for storing value by using equal sign (=)
- The **value** on the right side of equal sign is stored in variable(**identifier**) on the left side.

x = 10

- Variable must be named by beginning with letter (A-Z and a-z) or underscore (_).
- Python has **Reserved Words** which can't be used for naming variable.



Reserved Words

and	del	from	not
as	elif	global	or
assert	else	if	pass
break	except	import	print
class	exec	in	raise
continue	finally	is	return
def	for	lambda	try
while	with	yield	



Data type

- In some programming language, you must tell it that what kind of value will be stored in variable (such as C language in the camp using **int** for integer, **char** for character, etc.)
- But Python can choose appropriate type of variable for the value which you provided.



Leave it to me.
I will make your
programming life
easier.



Data type

- Python has 5 standard data type

1. Numbers

- Integer** (e.g., 1, 15, 500)
- Floating point** (e.g., 3.14, 24.7)

2. String

a character or a set of characters represented in *single quote* or *double quote* mark (e.g., "CPE", 'Hello World')

```
myNumber = 10
myString = "Hello CPE students!"
```



Data type

- List** : set of items ,which can be different type, separated with comma (,) and enclosed within square brackets ([]) (e.g. [1, 'a', 1.5])
- Tuple** : similar to list but it's items cannot be edited. Its values are enclosed within parentheses (())
- Dictionary** : similar to list but using key to indicate value. Its values are enclosed within brackets ({}))

```
myList = [1, "I am list", 555]
myTuple = (2, "I am Tuple", 666)
myDict = {name: "John", age: 18}
```



type function

- If you are not sure what type a value has, you can use **type()** to inspect them

```
print(type('Hello World'))
print(type(17))
Print(type(3.2))
```

Output

```
<class 'str'>
<class 'int'>
<class 'float'>
```



Input

- Let's know how to get input for your program

```
a = input("Enter something: ")
```

- Running the code above, program will print "Enter something: " to your screen and wait you to enter input.
- The value which you just entered will be stored as **string** type in variable a.



Input

- If you want to change data type of input, you can do it by **casting**.

```
a = int(input("Enter something: "))
```

convert input to integer type

- Casting** is one of the way to convert data type by telling the type which you want such as
 - `int(...)` : convert to integer
 - `float(...)` : convert to floating point
 - `str(...)` : convert to string
 - etc.



Arithmetic expression

- Guess what the following Python program does.

```
a = int(input())
b = int(input())
result = a + b
print(f"{a} + {b} = {result}")
```

The output

```
11
27
11 + 27 = 38
```

The program reads two integers, and outputs their summation.



A short program

- The following fragment of the code
 - Is easier to understand because it states its intention clearly.

```
import math

radius = float(input("Enter circle radius: "))
area = math.pi*radius*radius
print("The area of the circle is", area)
```

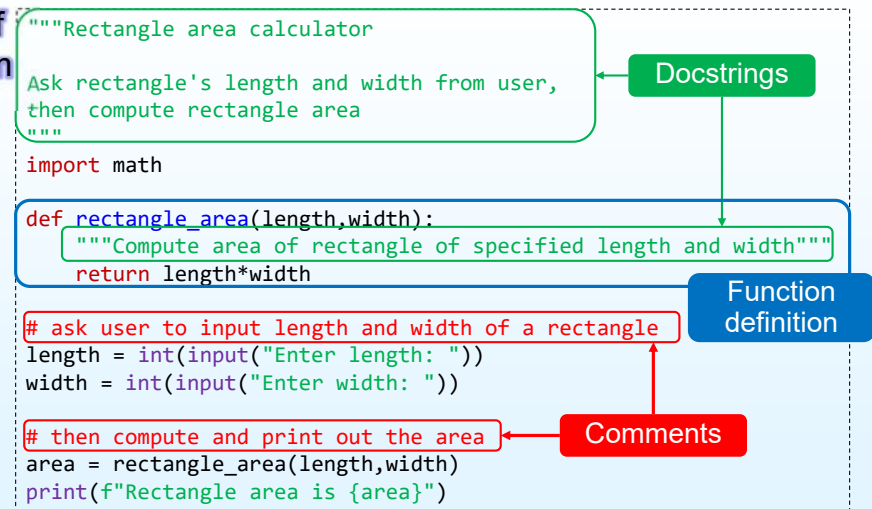
The output

```
Enter circle radius: 10
The area of the circle is 314.15926538979
```



A typical Python program

Flow of program



Comments & Docstrings

- They are in the code to provide insights into what code is doing, how code works, or give other notes.
 - They do not affect how program works.
- Docstrings are displayed when calling for help.

```
def rectangle_area(length,width):  
    """Compute area of rectangle of specified length and width"""  
    return length*width  
  
# ask user to input length and width of a rectangle  
length = int(input("Enter length: "))  
width = int(input("Enter width: "))
```

docstring

comment



A lot of keywords

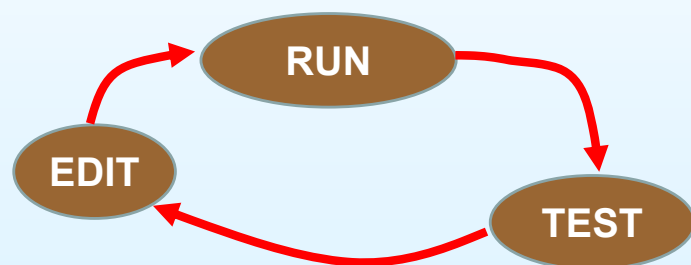
- The Python language uses many **keywords** to let you describe your ideas precisely.
- You will later learn the usage of some of the keywords, but not all.

```
import math  
  
def rectangle_area(length,width):  
    return length*width  
  
length = int(input("Enter length: "))  
width = int(input("Enter width: "))  
area = rectangle_area(length,width)  
print(f"Rectangle area is {area}")
```



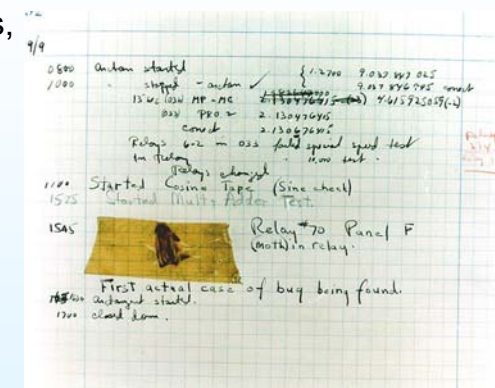
Edit-Run-Test Loop

- Because everything in life doesn't always work the first time you try it, your program may not always do exactly like what you want.
- It may be correct in some case, but it might fail in some other. Therefore, you need to **test** your program. If it is not correct, you have to **fix** it (**debug** it), and try to test it again.
- Your process for writing a Python program would look like this.



Debugging

- When programmers try to fix mistakes in their codes, they usually refer to the activity as "**debugging**" (killing the bugs).
- In the early days of computing, there was a **bug** in the machine. Therefore, problems with computers are often referred to as **bugs**.



<https://daily.jstor.org/the-bug-in-the-computer-bug-story/>



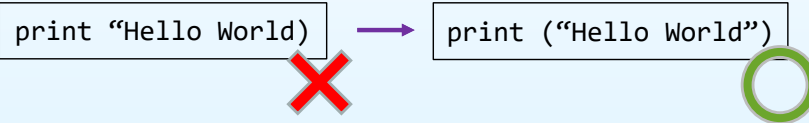
Bugs

- **Bugs** is programming errors. The process of tracking them down is call **debugging**.

Syntax Errors : cased from wrong structure of program

Ex.

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

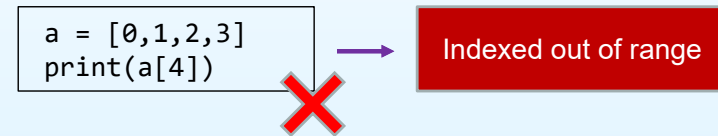


Bugs

Runtime errors : appear after the program has started running. This type also called **exceptions** because they usually indicate that something exceptional has happened.

Ex.

`a = [0,1,2,3]`
`print(a[4])` → Indexed out of range

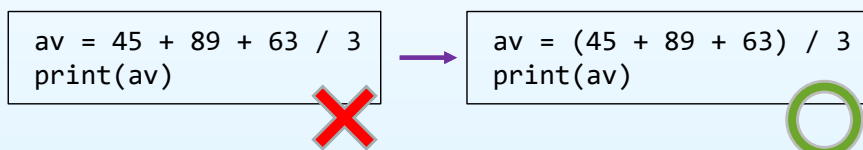


Bugs

Semantic errors : the program run successfully, and the computer will not generate any error messages, but it will not do the right thing.

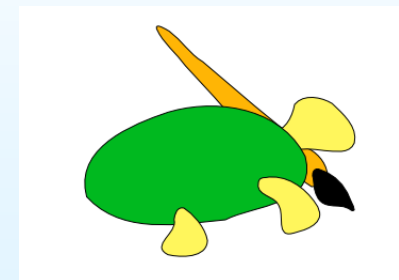
Ex.

`av = 45 + 89 + 63 / 3`
`print(av)` → `av = (45 + 89 + 63) / 3`
`print(av)`

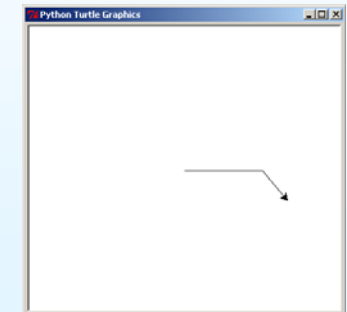


Python Turtle Graphics

- Python comes with Turtle graphics module that allows you to control movement of a robotic turtle on screen.
- The turtle carries a pen and draws over the path he moves on.



Graphics illustrated by Jittat Fakcharoenphol



Basic Turtle Movements

- `turtle.forward(d)` – tell Turtle to walk forward *d* steps
- `turtle.right(a)` – tell Turtle to turn right for *a* degrees
- `turtle.left(a)` – tell Turtle to turn left for *a* degrees



Tell Turtle to draw something

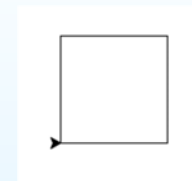
- The code is executed *sequentially* from top to bottom
- What is Turtle drawing?

Flow of
program



```
import turtle

turtle.forward(100)
turtle.left(90)
turtle.forward(100)
turtle.left(90)
turtle.forward(100)
turtle.left(90)
turtle.forward(100)
turtle.left(90)
```

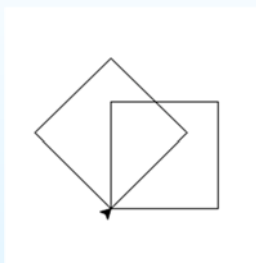


Guess the result

- What does this code do?

```
import turtle

turtle.forward(100)
turtle.left(90)
turtle.forward(100)
turtle.left(90)
turtle.forward(100)
turtle.left(90)
turtle.forward(100)
turtle.left(45)
turtle.forward(100)
turtle.left(90)
turtle.forward(100)
turtle.left(90)
turtle.forward(100)
turtle.left(90)
turtle.forward(100)
turtle.left(90)
```



Abstraction with *subroutines*

- A set of instructions can be defined into a subroutine so that they can be called for execution later.

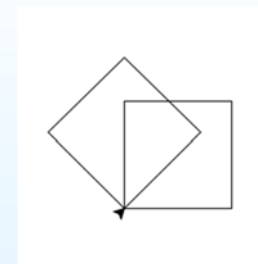
Define a subroutine
"draw_square"

```
import turtle

def draw_square():
    turtle.forward(100)
    turtle.left(90)
    turtle.forward(100)
    turtle.left(90)
    turtle.forward(100)
    turtle.left(90)
    turtle.forward(100)
    turtle.left(90)
```

Call the subroutine
"draw_square"

```
draw_square()
turtle.left(45)
draw_square()
```



Make it more general

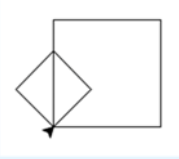
- The `draw_square` subroutine can be generalized to draw rectangles of any sizes.

```
import turtle

def draw_square(size):
    turtle.forward(size)
    turtle.left(90)
    turtle.forward(size)
    turtle.left(90)
    turtle.forward(size)
    turtle.left(90)
    turtle.forward(size)
    turtle.left(90)

draw_square(100)
turtle.left(45)
draw_square(50)
```

"draw_square" now takes a parameter



Draw a square of size 100

Draw a square of size 50



Get rid of *repetitive* code

- Most programming languages provide special constructs for repeated actions.

```
import turtle

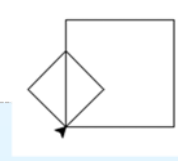
def draw_square(size):
    turtle.forward(size)
    turtle.left(90)
    turtle.forward(size)
    turtle.left(90)
    turtle.forward(size)
    turtle.left(90)
    turtle.forward(size)
    turtle.left(90)

draw_square(100)
turtle.left(45)
draw_square(50)
```

```
import turtle

def draw_square(size):
    for _ in range(4):
        turtle.forward(size)
        turtle.left(90)

draw_square(100)
turtle.left(45)
draw_square(50)
```



Execute actions *selectively*

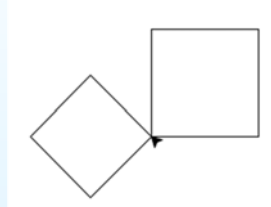
- Some actions need to be executed only when certain conditions are met.

```
import turtle

def draw_square(size):
    if size > 0:
        for _ in range(4):
            turtle.forward(size)
            turtle.left(90)

draw_square(100)
turtle.left(45)
draw_square(-50)
turtle.left(90)
draw_square(80)
```

This code is executed only when size is greater than zero.



This call gives no result



Conclusions

- Computer programming skills (or coding skills) are very important.
- A computer takes instructions in a machine-readable form. We write codes in a higher-level language which needs to be translated.
- Computer programs consist of instructions. Complex ideas are expressed in forms of
 - Sequential instructions
 - Subroutines
 - Repetitive execution
 - Selective (conditional) execution
- Finally, in this course you will learn to program using the Python programming language.

