Charles Truscott Watters

Certificate in Python from MIT (Massachusetts Institute of Technology)

Certificate of Unit Completion in Python from TAFE

School Certificate (Byron Bay High School, Class of 20120

Here within is my entire computer understanding, having foregone an understanding in cybersecurity to align with mathematical understanding and to seek a less-stressful work progression when I am not on disability payments.

The project began after a unit at MIT which I was very thankful to pass with a unit of completion in Computer Science and Programming Using Python (6.0001)

1. The x86 Instruction Set, Segmentation and Paging and Microprocessor Technology, Basic Notes

2. The ARM RISC Instruction Set, Basic Notes

3. The C and C++ Programming Languages, Two Examples

4. The Python Programming Language, Language Memorised

5. Algorithmic Complexity, Definitions from MIT and Counting Steps of Dominant Operations and Dominant Algebraic Terms of Growth

6. Standard Library of Python Memorised and Reduced, from the Python 3 Text on the Standard Library

7. Approximation, Searching and Sorting Definitions, from the course at MIT

8. Object Oriented Program Definitions, Basic Definitions and Example of Inheritance

9. Program Definition, Requirements Analysis and System Theory, Three Definitions as a Guide

10. Algorithm Design and Data Structures, Collected Notes from Dropping out of a Masters Degree

11. Thought Experiment [Definitions, from a Philosophical Text]

x86 Instructions Memorised

NOP, PUSH, POP, CALL, RET, MOV, LEA, ADD, SUB, CMP, TEST, JMP, JCC (Jump if Condition Met), AND, OR, XOR, NOT, IMUL, DIV, MUL, IDIV, REP STOS, REP MOVS, LEAVE, SYSENTER

16 General Purpose Registers, EFLAGS

otool

symbols

CISC - RISC

endianness

two's complement, one's complement, binary - decimal - hex conversions (shr, shl, radices, e.t.c.)

8 general purpose registers and the instruction pointer, x86 has 32-bit registers, x86-64 has 64-bit registers

Register conventions:

EAX - Stores function return values

EBX - Base pointer to the data section

ECX - Counter for string and loop operations

EDX - I/O pointer

ESI - Source pointer for string operations

EDI - Destination pointer for string operations

ESP - Stack pointer

EBP - Stack frame base pointer

EIP - Pointer to the next instruction to execute (instruction pointer)

Caller-save registers - EAX, EDX, ECX

If the caller has anything in the registers that it cares about the caller is in charge of saving the value before a call to a subroutine, and restoring the value after the call returns

Put another way, the callee can (and is highly likely to) modify the values in caller-save registers

SAVING REGISTERS FROM STACK, RESTORING REGISTERS

Callee-save registers - EBP, EBX, ESI, EDI

If the callee needs to use more registers than are saved by the caller, the callee is responsible for making sure the values are stored / restored

Put another way, the callee must be a good citizen and not modify registers which the caller didn't save, unless the callee itself saves and restores the existing values

The call to a function does not alter these registers

It can use the registers but saving them and restoring them

8/16/32 bit register addressing

EAX

AX

AH AL

ECX

CX

CH CL

EDX

DX

DH DL

EBX

BX

BH BL

ESP

SP

EBP

BP

ESI

SI

EDI

DI

EIP

IP

EFLAGS

EFLAGS register holds many single-bit flags.

Such as:

- Zero flag (ZF), Set the result if some instruction is zero, cleared otherwise

- Sign flag (SF), Set equal to the most-significant bit of the result, which is the sign bit of a signed integer (0 indicates a positive value, 1 a negative value)

The one-byte NOP instruction is a mnemonic for the XCHG EAX, EAX instruction (from book p. 112)

The stack

The stack is a conceptual area of main memory (RAM) which is designated by the operating system when a program is started.

A stack is a LIFO (last-in, first-out) data structure in which memory is pushed onto the top of the top of the stack and popped of the top of the stack.

By convention the stack grows toward lower memory addresses. Adding something to the stack means the top of the stack is now at a lower memory address.

ESP points to the top of the stack, the lowest address which is being used

While data exists at addresses beyond the top of the stack, it is considered undefined

The stack keeps track of which functions were called before the current one, it holds local variables and

is frequently used to pass arguments to the next function to be called

A firm understanding of what is happening on the stack is essential to understanding a program's operation

** POP ** e.g. POP EAX, EBX, ECX, EDX, ESI, EDI, ESP, EBP

** PUSH ** PUSH word, dword, qword onto the stack

x86, always a DWORD, can be either an immediate (numeric constant), or the value in a register

The push instruction automatically decrements the stack pointer, ESP, by 4

CALLING CONVENTIONS

How code calls a subroutine is compiler-dependent and configurable.

cdecl, stdcall

x86 calling conventions

"C declaration" - most common calling convention

Function parameters are pushed onto the stack from right to left

Saves the old stack frame pointer and sets up a new stack frame

EAX or EAX:EDX returns the result for primitive data types

Caller is responsible for cleaning up the stack

Calling conventions, Microsoft C++ code e.g. Win32 API

Function parameters pushed onto the stack from right to left

Saves the old stack frame pointer and sets up a new stack frame

EAX:EDX or EAX returns the result for primitive data types

Callee is repsonsible for cleaning up any parameters it takes

** CALL **

CALL's job is to transfer control to a different function, in a way that control can be later resumed where

it left off. First it pushes the address of the next function onto the stack (for use by RET when the procedure

is done). Then it changes EIP to the address given in the instruction. Destination addresses can be specified

in multiple ways:

- Absolute address

- Relative address (relative to the end of the instruction)

** RET **

Two forms

- Pop the top of the stack into EIP

  (in this form the instruction is just written as RET)

  typically used by cdecl functions

- Pop the top of the stack into EIP and add a constant number of bytes to ESP

  e.g. RET 0x8, RET 0x20

  typically used by stdcall functions

p. 133

** MOV **

- register to register
- memory to register, register to memory
- immediate to register, immediate to memory

no memory to memory

memory addresses are given in r/m32 form (talked about later)

Stack frame operation

stack bottom

| Local Variables | ------- | main() |

| undef |

| undef |

| Local Variables | ------- | main() |

| Caller-save | | undef |

| registers | | undef |

| Arguments to |

| pass to callee |

| Local Variables | ------- | main() |

| Caller-save | | foo()'s|

| frame |

| registers | | undef |

| Arguments to |

| pass to callee |

| caller's RET |

| Frame pointer |

EBP always starts at a stack frame

| Local Variables | ------- | main() |

| Caller-save | | foo()'s|

| frame |

| registers | | undef |

| Arguments to |

| pass to callee |

| caller's RET |

| Frame pointer |

| Callee-save |

| r/m32 |

| Local variables |

| Local Variables | ------- | main() |

| Caller-save | | foo()'s|

| registers | | frame |

| Arguments to | | undef |

| pass to callee |

Compile with GCC on a UNIX/LINUX system

Compile with Visual Studio C++ cl.exe

"r/m32" addressing forms

** LEA ** Load Effective Address

** ADD and SUB **

NOP

PUSH / POP

CALL / RET

MOV

LEA

ADD / SUB

(RET? EIP POP, e.g. RETN, RETN 4, RETN 8)

CONTROL FLOW

Two forms of control flow

- Conditional (go somewhere if a condition is met, e.g. 'if', 'switch', 'for(i = 0; i < n ; ++i)'

- Unconditional (go somewhere no matter what, e.g. 'function(int, double, float, char, char[], char **)', 'goto', exceptions, interrupts)

We have already seen procedure calls manifest themselves as push / call / ret

Let's see how goto manifests itself

** JMP **

- Change EIP to the given address

- Main forms of the address

** SHORT RELATIVE

- JMP 0x00402481 doesn't have the number 0x00402481 in it, it is simply just JMP 0xEE bytes forward

** NEAR RELATIVE

- 4 byte displacement from current EIP

** ABSOLUTE

- Hardcoded address in function

- Absolute indirect (address calculated with r/m32)

JMP -2 (infinite loop for short relative JMP)

p. 129

** JCC **

- Jump if condition is met

There are more than 4 pages of conditional jump types, luckily most are synonyms for each other

- JNE == JNZ (Jump if not equal, Jump if not zero, both check the EFLAGS register ZF, zero flag) e.g. ZF == 0

p. 137

NOTABLE CONDITIONAL JUMP INSTRUCTIONS (refer to manual, online references)

JZ / JE: if ZF == 1

JNZ / JNE: if ZF == 0

JLE / JNG: if ZF == 1 or SF != OF

JGE / JNL: if SF == OF

JBE: if CF == 1 or ZF == 1

JB: if CF == 1

FLAG SETTING

- Before you can do a conditional jump, you need something to set the condition flags for you

- Typically done with CMP, TEST, or whatever instructions are already inline and happen to have flag setting side effects

** CMP ** Compare two operands

" The comparison is performed by subtracting the second operand from the first operand and setting the

status flags in the same manner as the SUB instruction "

What's the difference from just doing SUB? Difference is that with SUB the result has to be stored

somewhere. With CMP the result is computed, the flags are set, but the result is discarded

Thus this only sets flags and doesn't mess up any of your registers

- Modifies CF, OF, SF, ZF, AF, and PF (implies that SUB modifies all those too)

p. 138

** TEST ** Logical compare

" Computes the bit-wise logical AND of the first operand (source 1 operand) and the second operand

(source 2 operand) and sets the SF, ZF, and PF status flags according to the result. "

- Like CMP, sets flags, and throws away the result

p. 232

Refresher - Boolean (bitwise) logic

AND & OR | XOR ^ NOT ~

** AND ** Logical AND

Destination operand can be r/m32 or register

Source operand can be r/m32 or register or immediate (No source `and` destination as r/m32s)

and al, bl and al, 0x42

p. 231

** OR ** Logical Inclusive OR

- Destination can be r/m32 or register

- Source operand can be r/m32 or register or immediate (" ")

or al, bl or al, 0x42

p. 231

** XOR ** Logical Exclusive OR

- Destination can be r/m32 or register

- Source operand can be r/m32 or register or immediate (" ")

xor al, al xor al, 0x42

XOR is commonly used to zero a register, by XORing it with itself, because it's faster than MOV

p. 231

** NOT ** One's Complement Negation

- Single source/destination operand can be r/m32

not al not [al + bl]

****

| NOP PUSH / POP CALL / RET MOV / LEA ADD / SUB JMP / JCC
|

|
|

| CMP / TEST AND/OR/XOR/NOT
|

|

|

|_____
_____|

NOP

PUSH / POP

CALL / RET

MOV / LEA

ADD / SUB

NPPCR

MLASJJ

CT

AOXN

5624

** SHL ** Shift Logical Left

** SHR ** Shift Logical Right

** IMUL ** Signed Multiply

** DIV ** Unsigned Divide

** REP STOS ** Repeat Store String

p. 35 lecture notes

** REP MOVS ** Repeat Move Data String to String

** LEAVE ** High Level Procedure Exit

NOP

PUSH / POP

CALL / RET

MOV / LEA

ADD / SUB

JMP / JCC

CMP / TEST

AND / OR / XOR / NOT

SHL / SHR

IMUL / DIV

REP STOS / REP MOV

LEAVE

big bro <3 <3 <3

POP QUIZ

EAX, EBX, ECX, EDX, ESI, EDI, ESP, EBP, EIP

1) EAX, stores return values

2) EBX, base pointer to data section

3) ECX, counter for string and loop operations

4) EDX, ? I/O pointer

5) ESI, source pointer for string operations

6) EDI, destination pointer for string operations

7) ESP, stack pointer

8) EBP, stack frame base pointer

9) EIP, instruction pointer

1) EAX, stores return values

2) EBX, base pointer to data section

3) ECX, counter for string and loop operations

4) EDX, I/O pointer

5) ESI, source pointer for string operations

6) EDI, destination pointer for string operations

7) ESP, stack pointer

8) EBP, stack frame base pointer

9) EIP, instruction pointer

Part 1 - Segmentation

Part 2 - Paging

Part 3 - Interrupts

Part 4 - Debugging, I/O, Misc fun on a bun

IA32

IA-32 System-Level Registers and Data Structures

** CPUID ** CPU (feature) identification

- Different processors support different features

- CPUID is how we know if the chip we're running on supports newer features, such as hardware virtualisation, 64-bit mode, hyperthreading, thermal monitors, etc.

- CPUID doesn't have operands. Rather it "takes input" as value preloaded into EAX (and possibly ECX). After it finishes the output is stored to EAX, EBX, ECX, and EDX.

0F A2	CPUID	Returns processor identification and feature information to the EAX, EBX ECX, and EDX registers, as determined by input in EAX (in some cases, ECX as well).

** PUSHFD ** Push EFLAGS onto Stack

9C	PUSHF	Push lower 16 bits of EFLAGS
9C	PUSHFD	Push EFLAGS
9C	PUSHFD	Push RFLAGS

- If you need to read the entire EFLAGS register, make sure you use PUSHFD, not just PUSHF
  (Visual Studio 2008 forces the 16 bit form if you don't have the D)

** POPFD ** Pop Stack into EFLAGS

9D	POPF	Pop top of stack into lower 16 bits of EFLAGS
9D	POPFD	Pop top of stack into EFLAGS
REX.W + 9D	POPFQ	Pop top of stack and zero-extend into RFLAGS

- There are some flags which will not be transferred from the stack to EFLAGS unless you're in ring 0

- If you need to set the entire EFLAGS register, make sure you use POPFD, not just POPF
  "" ""

p. 17 intmx86 lecture slide one - Some Example CPUID Inputs and Outputs

PROCESSOR MODES

In the beginning, there was real mode, and it sucked.

- Real-address mode

"This mode implements the programming environment of the Intel 8086 processor with extensions
(such as the ability to switch to protected or system management mode). The processor
is placed in real-address mode following a power-up or reset.

- DOS runs in real mode

- No virtual memory, no privilege rings, 16 bit mode

- Protected Mode - This mode is the native state of the processor. Among the capabilities of protected
mode is the ability to directly execute 'Real-address mode' 8086 software in a
protected, multi-tasking environment. This feature is called virtual-8086 mode
although it is not actually a processor mode. Virtual-8086 mode is actually a

protected mode attribute that can be enabled for any task

- Virtual-8086 is just for backwards compatability, and I point it out only to say that Intel says it's not really its own mode

- Protected mode adds support for virtual memory and privilege rings

- Modern OSes operate in protected mode

- System Management Mode - This mode provides an operating system or executive with a transparent mechanism for implementing platform-specific functions such as power management and system security. The processor enters SMM when the external SMM interrupt pin (SMI#) is activated or an SMI is received from the advanced programmable interrupt controller (APIC).

- SMM has become a popular target for advanced rootkit discussions recently because access to SMM memory is locked so that neither ring 0 nor VMX hypervisors can access it. Thus is VMX is more privileged than ring 0 ("ring -1"), SMM is more privileged than VMX ("ring -2") because a hypervisor can't even read SMM memory.

- Reserving discussion of VMX and SMM for Advanced x86 class

Vol. 1, Sect 3.1

p. 23 lecture notes 1

Figure 1-6 Operating Modes of the AMD64 Architecture

e.g. 64 bit mode, compatability mode (long mode)

Protected Mode, Virutal 8086 Mode

Real Mode, System Management Mode

PRIVILEGE RINGS

MULTICS was the first OS with support for privilege rings

x86's rings are also enforced by hardware

You often hear that normal programs execute in ring 3, (userspace/usermode), and the privileged code

runs in ring 0 (kernelspace / kernelmode)

The lower the ring number, the more privileged the code is

In order to find the rings, we need to understand a capability called segmentation

Vol 3a, Sect 5.5 (Level 0 - OS kernel, Level 1 - OS Services, Level 2 - OS Services, Level 3 - Applications)

Paravirtualized Xen (Requires a modified guest OS)

Figure 5: Execution mode

level3

level2

level1

level0

Xen

Guest OS

Application

(Newest Xen instead uses HW VMX to be more privileged than the OS kernel)

Segmentation

- "Segmentation provides a mechanism for dividing the processor's addressable memory space
(called the linear address space) into smaller protected address spaces called segments.")

Figure 3-4 - Multi-segment model

Vol 3a, Sect 3.1 & 3.2.3

Segment registers, segment descriptors, linear address space (or physical memory)

CS	Access	stack
	Limit	code
SS	Base address	data
		data
DS		data
ES		
FS		
GS		

Segment addressing

- "To locate a byte in a particular segment, a logical address (also called a far pointer) must be provided. A logical address consists of a segment selector and an

offset."

- "The physical address space is defined as the range of addresses that the processor can generate on its address bus."

- Normally the physical address space is based on how much RAM you have installed, up to a maximum of 2^32 (4GB). But there is a mechanism (physical address extensions - PAE) which we will talk about later which allows systems to access a space up to 2^36 (64GB).

- Basically a hack for more than 4GB of RAM but who aren't using a 64 bit OS.

- Linear address space is a flat 32 bit space

- If paging (talked about later) is disabled, linear address space is mapped 1:1 to physical address space

Vol. 3a, Sect 3.1

Segmentation restated

- Segmentation is not option

- Segmentation translates logical addresses to linear addresses automatically in hardware by

using table lookups

- Logical address (also called a far pointer) = 16 bit selector + 32 bit offset

- If paging (talked about later) is disabled, linear addresses map directly to physical addresses

Figure 3-5. Logical to Linear Address Translation

Logical Address, Segment Selector

Descriptor Table, Segment Descriptor

Base Address

Linear Address, Offset (effective address)

Figure 3-1. Segmentation and Paging

Vol. 3a, Sect 3.1

Segment Selectors

- A segment selector is a 16-bit value held in a segment register

It is used to select an index for a segment descriptor from one of two tables

- GDT (Global Descriptor Table), for use system-wide

- LDT (Local Descriptor Table), intended to be a table per-process and switched when the

kernel switches between process contexts

- Note that the table index is actually 13 bits not 16, so the tables can each hold 2^13 = 8192

descriptors

Figure 3-6. Segment Selector

Table Indicator (0 = GDT, 1 = LDT), Requested Privilege Level

The Six Segment Registers

- CS, code segment

- SS, stack segment

- DS, data segment

- ES/FS/GS, Extra (usually data) segment registers

- The "hidden part" is like a cache so that segment descriptor information doesn't have to be looked

up each time.

Figure 3-7. Segment Registers

Visible part, hidden part, segment selector, base address, limit, access information

CS

SS

DS

ES

FS

GS

Implicit use of segment registers

- When you're addressing the stack, you're implicitly using a logical address that is using the SS (stack segment)

register as the segment selector, (i.e. ESP == SS:ESP)

- Even if a disassembler doesn't show it, the use of segment registers is built into some of your favourite

F3 A5 REP MOVS m32, m32 Move (E)CX doublewords from DS:[(E)SI] to ES:[(E)DI]

Explicit use of segment registers

- You can write assembly which explicitly specifies which segment register it wants to use. Just prefix the

memory address with a segment register and colon

- e.g. "mov eax, [ebx]", as opposed to "mov eax, fs:[ebx]"

- The assembly just puts a prefix on the instruction to say "when this instruction is asking for memory, it's

actually asking for memory in this segment."

- In this way you're actually specifying a full logical address / far pointer

Inferences

- Windows maintains different CS, SS, & FS segment selectors for userspace processes vs kernel ones

- The RPL field seems to correlate with the ring for kernel or userspace

- Windows doesn't change DS or ES when moving between userspace and kernelspace (they were the exact same values)

- Windows doesn't use GS

One more time

One of the segment registers (SS/CS/DS/ES/FS/GS), address used in some assembly instruction, GDT or LDT

Figure 3-5. Logical Address to Linear Address Translation

Logical address, seg. selector

Descriptor table, seg. descriptor

Base address

Offset (effective address)

Linear address

+

GDT & LDT

Segment Selector, Global Descriptor Table (TI = 0), Local Descriptor Table (TI = 1)

All entries in these tables are "Segment Descriptor" structures

GDTR Register, LDTR Register

Limit, Base Address; Limit, Base Address, Seg. Sel.

Special registers point to the base of the tables and specify their size

Fig. 3-1. Global and Local Descriptor Tables

Global Descriptor Table Register (GDTR)

Vol 3a. Figure 2-5

- The upper 32 bits ("base address") of the register specify the linear address where the GDT is stored

- The lower 16 bits ("table limit") specify the size of the table in bytes

- Special instructions used to load a value into the register or store the value out to memory

    - LGDT (load 6 bytes from memory into GDTR)
    - SGDT (store 6 bytes of GDTR to memory)

Local Descriptor Table Register (LDTR)

Vol 3a. Figure 2-5

- Like the segment registers, the LDT has a visible part, the segment selector, and a hidden part, the cached segment info. which specifies the size of the LDT

    - The selector's "Table Indicator" (T) bit must be set to 0 to specify that it's selecting from the GDT, not from itself ;)

- Special instructions used to load a value into the register or store the value out to memory

- LLDT (load 16 bit segment selector into LDTR)

- SLDT (store 16 bit segment selector of LDTR to memory)

Segment Descriptors

2. Summarised ARM

ADD

ADC

SUB

SBC

RSB

RSC

MUL

MLA

MLS

SDIV

UDIV

NOP

ASR - arithmetic shift right

LSL - logical shift left

LSR - logical shift right

ROR - rotate right

RRX - rotate right with extend

MOV

MOVT

MOVS PC

MVN Bitwise NOT of value into dest

REV

REV16

REVSH

AND

BIC

EOR

ORR

ORN

CMP

CMN

TEQ

TST

BIC

LDR

STR

B

BL

BLE

BGT

BEQ

BNE

BLX

BlX

BX

CBZ

CBNZ

TBB

TBH

Acorn RISC (Reduced Instruction Set Computing) Advanced RISC Machine

RISC architecture work done at UCal Berkley and Stanford

Cortex-A9 ARMv7

ARM Features

Similar to RISC architecture (not purely RISC)

Variable-cycle instructions (LD / STR multiple)

Inline barrel shifter

16 bit (thumb) and 32 bit instruction sets combined called thumb 2

Conditional execution (reduces number of branches)

Auto increment decrement addressing modes

Modified Harvard architecture since ARMv5 ARM9

Extensions

TrustZone, VFP, NEON, SIMD, DSP and Multimedia Processing

REGISTERS

37 registers

- 30 general purpose

- 1 pc program counter

1 CPSR - current program status register

- SPSR saved program status register

(the saved CPSR for each of the five exception modes)

several exception modes

10:02

r0

r9

r10 (SL)

r11 (FP)

r12 (IP)

r13 (SP)

r14 (LR)

r15 (PC)

CPSR

(SP) stack pointer top element of the stack address

(LR) link register saves the pc when entering a subroutine

(PC) address of the next instruction (ARM +8 thumb +4)

(IP) not instruction pointer but intra procedural call scratch register

CPSR results of most recent operations including flags, interrupts, enable / disable and modes

fetch -> decode -> execute

18:51

push <reg list> decrements the SP (4 bytes) and stores the values in <reg list> at that location

pop <reg list> increments the SP (4 bytes) and stores the values at SP in <reg list>

ADD

ADC

SUB

SBC

RSB

RSC

MUL

MLA

MLS

multiply operations only store least significant 32 bits

SDIV

UDIV

NOP

Hardware optimisation inline with the arithmetic logic unit allows for a multiplier (power of 2) within same instruction cycle

Allows for shifting a register value by either an unsigned integer or value at the bottom byte of a nother

register

ASR - arithmetic shift right

LSL - logical shift left

LSR - logical shift right

ROR - rotate right

RRX - rotate right with extend

MOV

MOVT

MOVS PC

MVN Bitwise NOT of value into dest

REV

REV16

REVSH

AND

BIC

EOR

ORR

ORN

CMP

CMN

TEQ

TST

BIC

LDR

STR

B

BL

BLE

BGT

BEQ

BNE

BLX

BlX

BX

CBZ

CBNZ

TBB

TBH

ELF (executable linkable format)

3. The C / C++ Programming Language

```
#include <stdio.h>
#include <math.h>
#include <assert.h>

int main(void) {
        int x = 0;
        printf("Enter any number, an integer number: ");
```

```
	scanf("%d", &x);
	printf("\n%d squared is %d\n", x, pow(x, 2));
	return 0;
}



#include <string>
#include <iostream>
#include <vector>

int main(int argc, char ** argv) {
	int fahr = 0;
	int step = 5;
	int upper_bound = 160;
	for (int i = 0; int i <= upper_bound; i += step) {
		fahr = ((5/9) * i) - 32;
		std::cout << "\t" << i << " celsius is " << fahr << " fahrenheit.\n";
		std::cout << "\tCharles Truscott Watters" << std::endl;
	}




}
```

Declarative vs. Imperative Knowledge

Statements of Fact vs. How-To Methods

Operators

+	-	*	/	//	%	**	( )	+=	-=	*=	/=
	//=	%=	**=	!=	==	<	<=	>	>=	&	|
	^	~	<<	>>	&=	|=	^=	~=	<<=	>>=	.
	*	**	[ ]	[:]	[::]	[i][j][k] =		;	__	:	,

int, float, string, set, tuple, list, dictionary, boolean, class, object, lambda, function, function invocation, function return, combination of types, expression

True, False, Lambda, yield, from x import y as z, import x as z

break, continue, is, is not, in, as, else

assert, global, nonlocal, pass, del

try:

except Error:

finally:

else:

error as e

abstraction, decomposition

input x -> get y, breaking up a task into smaller sub-tasks

```
def function(args):
    body
    return

def main():
    body
if __name__ == "__main__": main()

for x in range(start, stop + 1, step):
for x in iterable: (e.g. string, set, tuple, list, dictionary)
for x in a:
    for y in b:
```

for x in a:

    for y in x:

while (boolean):

    while (boolean):

def recursive_function(a, b):

    base case 1:

    base case x:

    body

    recursive_function(a - 1, b - 1)

    return

iteration, recursion, base cases, memoisation, tabularisation, recursive call design

if (boolean)

    if (boolean)

        if (boolean)

        elif (boolean)

elif (boolean)

else

elif ...

elif ..

else

elif ...

elif ..

else

match (object):

case x:

body

case y:

body

case z:

global and local scope, branching and control flow, conditionals