COMP(2041|9044) 26T1 — Python Introduction

https://www.cse.unsw.edu.au/~cs2041/26T1/

Python

- Developed by Guido van Rossum in 1989.
- Is a useful tool to know because it is:
  - one of the most widely-used languages
  - widely available on Unix-like and other operating systems
  - Python scripts occur many places in existing systems
  - libraries available for many, many purposes
  - prototyping code can be very fast
  - useful as interactive calculator

Zen of Python

```
>>> import this
Beautiful is better than ugly.
Explicit is better than implicit.
Simple is better than complex.
Complex is better than complicated.
Flat is better than nested.
Sparse is better than dense.
Readability counts.
Special cases aren't special enough to break the rules.
Although practicality beats purity.
Errors should never pass silently.
Unless explicitly silenced.
In the face of ambiguity, refuse the temptation to guess.
There should be one -- and preferably only one -- obvious way to do it.
Although that way may not be obvious at first unless you're Dutch.
Now is better than never.
Although never is often better than *right* now.
If the implementation is hard to explain, it's a bad idea.
If the implementation is easy to explain, it may be a good idea.
Namespaces are one honking great idea -- let's do more of those!
```

Python

So what is the end product like?

- a language which makes it easy to build useful systems
- a language which makes it easy to prototype and iterate
- a language with high level libraries and functions
- interpreted: slow/high power consumption
- type checking added as afterthought
  - see https://dropbox.tech/application/our-journey-to-type-checking-4-million-lines-of-python

Summary: it's easy to write concise, powerful (but slow), readable programs in Python

Compilers versus Interpreters

compiler translates program to machine code which when executed implements program

```
$ clang hello.c -o hello
$ ./hello
Hello Andrew!
```

interpreter reads program and executes its statements directly

```
$ bash hello.sh
Hello Andrew!
```

- reality more complicated
  - compilation typically complex multi-step process
  - compilers may bundle mini-interpreter into machine code
  - interpreters often do some run-time compilation to the bytecode of a virtual machine

Compiled Languages versus Interpreted Languages

- in principle, all languages can be compiled or interpreted
  - usually only one or the other commonly used
- languages usually compiled to machine code: C, C++, Rust, Go, Swift
- languages usually interpreted: Python, Shell, JavaScript, R, Perl, Ruby, C#, Java, PHP
  - interpreters often translate program to intermediate form
  - intermediate form often thought as instruction of virtual (imaginary) machine
  - often called run-time-compilation or just-in-time-compilation
  - can also be to machine code
- languages where both compiled & interpreted implementations common: Haskell, OCaml, Basic, Pascal, LISP

Resources

Python official documentation superb:

- tutorial https://docs.python.org/3/tutorial/
- library https://docs.python.org/3/library/
  - especially types https://docs.python.org/3/library/stdtypes.html

So many other online resources

Books:

- Fluent Python - Luciano Ramalho
- Python Cookbook - David Beazley

Which Python

- Python 2.0 was released in 2000
- Python 3.0 was released in 2008
- Since 2020 only Python 3 is supported
- New minor version release every 17 months (3.X) (PEP-602)
- New patch version release every 2 months for 24 months (3.X.Y) (PEP-602/PEP-745)
- Current version is 3.14
- CSE servers currently run Python 3.13
  - all code you write for COMP(2041|9044) needs to work with Python 3.13
  - shouldn't be an issue - no major new features in Python 3.14/3.15
- huge amount of software libraries available
  - 300+ standard modules
  - PyPI has 767,000 packages which can be easily installed

Running Python

Python programs can be invoked in several ways:

- giving the filename of the Python program as a command line argument:

```
$ python3 code.py
```

- giving the Python program itself as a command line argument:

```
$ python3 -c 'print("Hello, world")'
```

- using the `#!` notation and making the program file executable:

```
$ head -n1 code.py
#! /usr/bin/env python3
$ chmod 755 code.py
$ ./code.py
```

Running Python

- although converting code Python 2 -> 3 is straightforward, still many legacy Python 2 applications
- Many systems have both Python 2 and Python 3 installed.
  - run `python3` you get Python 3
  - run `python2` you might get Python 2, or nothing
  - run `python` you might get Python 2, or Python 3, or nothing
- CSE servers no long provide Python 2

Variables

Python has a strong, dynamic (gradual), duck type system.

Python provides many built-in types:

- Numeric Types — int, float
  - float is 64 bit IEEE754 (like C double)
  - int is arbitrary
- Text Sequence Type — str
- Sequence Types — list, tuple, range
- Mapping Types — dict
- More: boolean, None, functions, class
- More: complex, iterator, bytes, bytearray, memoryview, set, frozenset, context manager, type, code, ...

Unlike C, you cannot have uninitialised variables

Variables can optionally be given a `type hint` for static type checking.

Comparison Operators

Python Comparison Operators are the same as C:

Operator	Name	Description
==	Equal	are both operands the same value
!=	Not equal	are both operands not the same value
>	Greater than	is the left operand bigger than the right operand
<	Less than	is the left operand smaller than the right operand
>=	Greater than or equal to	is the left operand bigger than or equal to the right operand
<=	Less than or equal to	is the left operand smaller than or equal to the right operand

Bitwise Operators

Python Bitwise Operators are the same as C:

Operator	Name	Description
&	and	Sets each bit to 1 if both bits are 1
|	or	Sets each bit to 1 if one of two bits is 1
^	xor	Sets each bit to 1 if only one of two bits is 1
~	not	Inverts all the bits
«	left shift (zero fill)	Shift left by pushing zeros in from the right, and let the leftmost bits fall off
»	right shift (sign extended)	Shift right by pushing copies of the leftmost bit in from the left, and let the rightmost bits fall off

Arithmetic Operators

Python Arithmetic Operators are *almost* the same as C:

Operator	Name
+	Addition
-	Subtraction
*	Multiplication
//	Division (returns int)
/	Division (returns float)
%	Modulus
**	Exponentiation

Assignment Operators

Operator	Name	Example	Equivalent
:=	assignment	x := 5	x := 5
&=	and	x &= 5	x = x & 5
|=	or	x |= 5	x = x | 5
^=	xor	x ^= 5	x = x ^ 5
«=	left shift (zero fill)	x «= 5	x = x « 5
»=	right shift (sign extended)	x »= 5	x = x » 5
+=	Addition	x += 5	x = x + 5
-=	Subtraction	x -= 5	x = x - 5
*=	Multiplication	x *= 5	x = x * 5
//=	Division (returns int)	x //= 5	x = x // 5
/=	Division (returns float)	x /= 5	x = x / 5
%=	Modulus	x %= 5	x = x % 5
**=	Exponentiation	x **= 5	x = x ** 5

Logical Operators

Unlike C, Python uses words for its logical operators:

C	Python	Description	Example
x || y	x *or* y	Returns True if both statements are true	`a > 5 and a < 10`
x && y	x *and* y	Returns True if one of the statements is true	`a < 0 or a > 100`
!x	*not* x	Reverse the result, returns False if the result is true	`not (a > 100 and a < 1000)`

New Operators

Python also has new operators:

Type	Python	Description	Example
Identity	x *is* y	Returns True if both variables are the same object	`a = []; b = a; a is b`
Identity	x *is not* y	Returns True if both variables are not the same object	`a = []; b = []; a is not b`
Membership	x *in* y	Returns True if a sequence with the specified value is present in the object	`5 in [4, 5, 6]`
Membership	x *not in* y	Returns True if a sequence with the specified value is not present in the object	`7 not in [4, 5, 6]`

Missing Operators

Python has two notable absences from the list of C operators:

Operator	C	Python
Increment	`x++`	`x += 1`
Decrement	`x--`	`x -= 1`

Operators

Examples:

```
x = '123'
# `x` assigned string "123"
y = "123 "
# `y` assigned string "123 "
z = 123
# `z` assigned integer 123
a = z + 1
# addition (124)
b = x + y
# concatenation ("123123 ")
c = y + z
# invalid (cannot concatenate int to str)
d = x == y
# compare `x` and `y` (False)
```


Python - what is True

- **`False`** is false
- **`None`** is false
- numeric zero is false
  - e.g. `0` `0.0`
- empty sequences, mappings, collections are false
  - so empty, strings, lists, tuples, dicts are false false
  - e.g. `""` `[]` `()` `{}` `set()`
- everything else is true
- beware all these values true: `"0"` `[0]` `(None,)` `[[]]`
  - but `(None)` is false - because its not a tuple just a bracketed value

Control Structures

Python **doesn't require** the use of semicolon `;` between or to end statements. Use of a semicolon `;` is **not recommended** in normal code, but can be useful when supply Python on command-line. All of there are valid:

```
x = 1
print("Hello")
```

```
x = 1;
print("Hello")
```

```
x = 1;
print("Hello");
```

```
x = 1; print("Hello")
```

```
x = 1; print("Hello");
```

Control Structures

All statements within control structures must be after a colon `:` Python uses indentation to show code blocks (C uses *{* and *}*).
Using brackets around the condition is optional.

```
if (x > 9999):
  print("x is big")
```

```
if (x > 9999): print("x is big")
```

```
if (x > 9999):
  print("x is big")
  print(f"the value of x is {x}")
```

```
if x > 9999: print("x is big")
```

```
if x > 9999:
  print("x is big")
  print(f"the value of x is {x}")
```


Control Structures - Selection

Selection is handled by: **`if`** -> **`elif`** -> **`else`**

```
if boolExpr{1}:
  statements{1}
elif boolExpr{2}:
  statements{2}
...
else:
  statements{n}
```


Control Structures - Iteration

Iteration is handled by: **`while`** and **`for`**

```
while boolExpr:
  statements
```

```
for value in iterator:
  statements
```

Control Structures - Iteration

C style **for** loops can be approximated with the `range()` function (`range()` is inclusive on its lower-bound (default `0`) and exclusive on its upper-bound (with a default step-size of 1))

```
  for value in range(90, 100, 2):
    print(value)
# 90
# 92
# 94
# 96
# 98
```


Control Structures - Iteration

break and **continue** can be used in loops just as in C: - **break** will end the loop - **continue** starts the next iteration of the loop

else can be used on loops, the else case is executed if the loop exits normally (without a **break**)

```
for value in range(a + 1, b):
    if is_prime(value):
        print(f"At least one prime between {a} and {b}")
        break
else:
    print(f"No primes between {a} and {b}")
```


Control Structures - Iteration

Example (compute $pow = k^n$):

```
# Method 1 ... while
pow = i = 1
while i <= n:
    pow = pow * k
    i += 1
# Method 2 ... for
pow = 1
for _ in range(n):
    pow *= k
# Method 3 ... built-in operator
pow = k ** n;
# Method 4 ... operator
from operator import pow as power
pow = power(k, n);
```

Control Structures - Selection

*selection can also be done with: **match** -> **case**

```
match var:
  case option{1}:
    statements{1}
  case option{2} | option{3} | option{4}:
    statements{2}
  ...
  case option{n}:
    statements{n}
  case _:
    statements{default}
```

match / **case** in python can do anything a C **switch** / **case** can do plus much more.

Added in Python 3.10 - missing in older Python 3 installations

Terminating

Normal termination, call: **sys.exit()**

The **assert** or the **raise** keywords can be used for abnormal termination:

Example:

```
import sys

if not is_valid:
  print("something wasn't valid", file=sys.stderr)
  sys.exit(1)
```

```
assert data is not None, "data was None, it shouldn't be"
```

```
def func (a):
  if not isinstance(a, int):
    raise TypeError(f"\'{a}\' is not of type <int>")
```


Simple I/O example

```
import math
x = float(input("Enter x: "))
y = float(input("Enter y: "))
pythagoras = math.sqrt(x**2 + y**2)
print(f"Square root of {x} squared + {y} squared is {pythagoras}")
```

source code for pythagoras.py

Simple I/O example - Reading Lines #1

```
sum = 0
for line in stdin:
    sum += int(line)
print(f"Sum of the numbers is {sum}")
```

source code for sum_stdin.0.py

Simple I/O example - Reading Lines #2

```
sum = 0
for line in stdin:
    try:
        sum += int(line)
    except ValueError as e:
        print(e)
print(f"Sum of the numbers is {sum}")
```

source code for sum_stdin.1.py

Simple String Manipulation Example

```
try:
    line = input("Enter some input: ")
except EOFError:
    print("could not read any characters")
    exit(1)
n_chars = len(line)
print(f"That line contained {n_chars} characters")
if n_chars > 0:
    first_char = line[0]
    last_char = line[-1]
    print(f"The first character was '{first_char}'")
    print(f"The last character was '{last_char}'")
```

source code for line_chars.py

Simple String Comparison Example

```
last = None
while True:
    try:
        curr = input("Enter line: ")
    except EOFError:
        print()
        break
    if curr == last:
        print("Snap!")
        break
    last = curr
```

source code for snap_consecutive.py

Creating A Gigantic String

```
if len(sys.argv) != 2:
    print(f"Usage: {sys.argv[0]}: <n>")
    exit(1)
n = 0
string = "@"
while n < int(sys.argv[1]):
    string *= 2
    # or `string += string`
    # or `string = string + string`
    n += 1
print(f"String of 2^{n} = {len(string)} characters created")
```

source code for exponential_concatenation.py