Advanced Programming in Python

Lecture 11:
The rest of Python

Chalmers/GU CSE (DAT515/DIT515)

Version 20231128

Aarne Ranta

Plan

One of the goals of this course is to *cover all of Python*, at least in the reading mode.

This lecture will cover the rest of Python constructs, some quite new, some older:

- :=
- match
- yield
- & | ^ ~
- x: int
- raise
- re
- async

None of these is necessary in the labs, and they will not be asked in the exam!

https://docs.python.org/3/reference/grammar.html

$:=$

Assignment expression **x := e**

Using the "walrus symbol" **:=**

- returns the value of e
- assigns it to variable x

In many other languages (C, Java, …) ordinary assignments **x = e** do the same. Hence they are expressions, not just statements.

But Python forbids this to avoid unwanted errors, such as

```
if x = 42:
    print(x)  # always prints 42
```

This is a syntax error, because **x = 42** is a statement.

https://peps.python.org/pep-0572/

https://upload.wikimedia.org/wikipedia/commons/8/8f/Walrus_2_%286383855895%29.jpg

The three equality signs: = == :=

```
x = 7

x == 42

if x == 42:
    print(x)

if x := 42:
    print(x)

if x := 0:
    print(x)
```

Assignment statement, sets the value of x to 7

Equality expression, value False

Nothing is printed, because x is 7

Prints 42, sets the value of x to 42

Quiz: what is printed here?

match

Pattern matching: **match** and **case**

```
def http_error(status):
    match status:
        case 400:
            return "Bad request"
        case 404:
            return "Not found"
        case 418:
            return "I'm a teapot"
        case 200|201|202:
            return "Some kind of success"
        case x if 300 <= x < 400:
            return "Some kind of redirection"
        case _:
            return "Something's wrong with the Internet"
```

Matching with integers

disjunctive patterns

if conditions to patterns

matching anything not yet covered

Could be mimicked with an if-elif-else block but would be more complicated.
https://peps.python.org/pep-0636/

Matching lists

```
while True:
    command = input('> ')

    match command.split():
        case ['reverse', s]:
            print(s[-1::-1])
        case ['reverse', *s]:
            print('cannot reverse multiple words')
        case ['echo', *s]:
            print(*s)
        case ['quit'|'bye']:
            print('bye')
            break
        case _:
            print('try again')
```

A dialogue interpreter; cf. Lab 1.

Testing with different sequences of words

- Just one word: reverse
- Many words, cannot reverse
- Many words: just print

- quit or bye: quit the dialogue

- any other input: invalid

Also other structures can be matched, including objects of your own classes.

yield

Generators: **yield** and **next**

```
def fibonacci():
    lo, hi = 1, 1
    while True:
        yield lo
        lo, hi = hi, lo+hi

fs = fibonacci()
while not input():
    print(next(fs), end='')
```

This generates an "infinite" list of Fibonacci numbers

- instead of building a list with **append()** and **return**ing it

To test: get one number at the time, the **next** one, by pressing just enter. Any other input terminates.

Common use: reading large files line by line (this is what standard **open()** actually does)

https://realpython.com/introduction-to-python-generators/

& | ^ ~

Binary numbers and bitwise operators

```
x = 0b101010

int(x)  # == 42

bin(42)  # == '0b101010'

20 & 10  # == 0b10100 & 0b1010 == 0b0

20 | 10  # == 0b11110 == 30

20 ^ 10  # == 0b11110 == 30

~ 20  # -21
```

Literals for binary numbers

value shown as decimal (= base 10)

converted to binary

bitwise and (x * y)

bitwise or (x + y)

bitwise xor (x + y mod 2)

bitwise negation (1 - x)

https://realpython.com/python-bitwise-operators/

x: int

Type hints

```
def greeting(name: str, n: int) -> str:
    return n * 'Hello ' + name


# a violation
print(greeting('world', '3'))
```

```
$ python3 lecture11.py
TypeError: can't multiply sequence by
non-int of type 'str'

$ mypy lecture11.py
lecture11.py:59: error: Argument 2 to
"greeting" has incompatible type "str";
expected "int"  [arg-type]
Found 1 error in 1 file (checked 1 source
file)
```

Good for documenting functions in an API.
Also used in **static type checking**

- not native in Python
- but performed by preprocessing
  - with mypy
  - in pycharm, built in as warnings

https://docs.python.org/3/library/typing.html

Normal execution: the first error found when running the code is reported

With static type checking: all errors found are reported before running the code

https://realpython.com/python-type-checking/

raise

Defining and raising exceptions

```
class AsciiException(Exception):
    def __str__(self):
        return 'non-ascii characters in string'


def get_username():
    name = input('username: ')
    if any([ord(c) > 127 for c in name]):
           raise AsciiException
    return name
```

A user-defined exception is a subclass of the **Exception** class.

- the __str__() method defines the error message

To make the execution with a certain exception, **raise** it.

https://docs.python.org/3/tutorial/errors.html

Defining and raising meaningful exceptions is better than returning error strings or **None** values.

Other functions can then catch them in **try-except** blocks.

re

Regular expressions

```
import re

re.match('\d+', '123abc')
# <re.Match object; span=(0, 3), match='123'>
```

Inherited from the Perl language, reflecting the origin of Python as a scripting language.
https://docs.python.org/3/howto/regex.html#regex-howto
https://docs.python.org/3/library/re.html

re.match(<pattern>, <str>) matches in the beginning
re.search(...) returns the first match
re.findall(...) returns a list of all matches
re.finditer(...) yields all matches

async

Asynchronous IO

```
import asyncio

async def hello(i):
    print(f"hello {i} started")
    await asyncio.sleep(4)
    print(f"hello {i} done")

async def main():
    task1 = asyncio.create_task(hello(1))
    await asyncio.sleep(3)
    task2 = asyncio.create_task(hello(2))
    await task1
    await task2

asyncio.run(main())
```

Asynchronous programs: tasks running at the same time without blocking each other

https://docs.python.org/3/library/asyncio.html

Simple example from

https://stackoverflow.com/questions/50757497/simplest-async-await-example-possible-in-python

Much more in

https://realpython.com/async-io-python/