Syntax

!/usr/bin/env python3

-- *coding:something* -- (not needed if using utf-8/ASCII

```
/n  <- comment
end line with \  joins next line,
    works in string literls " ... \ ... "
grouping (...) {...} [...] span lines
ignore blank lines
```

INDNT DEDENT NEWLINE

```
python program after first two lines is valid if 0
or more statements spanning multiple lines
```

Identifiers

```
start: _, letters, not numbers.
continue: _, letters, numbers.
_... means semi-private or 'private to module'
__...__ means special to python
__... means private
```

Literals

```
decimals: start with 1-9 and continue with 0-9
binary: start with 0b
octal: start with 0o
hex: start with 0x
can include _ in all as grouping separator
```

Floating point literals

```
    .19 ok
    19. ok
    1.9 ok
    0.9 ok
    can include _
    e, E ok 1.9e2 == 1900
 Imaginary:
     decimal, float + j/J  e.g. 10+2j
```

String

```
'....' -can have escaping lile /n
"...."
'''....''' -can span lines
"""....""""" -can span lines
prefix
r'...' no excapes
f'...' f-string formatting
rf'...'
b'...' not a string, bytes [0-255]
```

Grouping

Parenthesis

```
-()     empty tuple
-(expr) math i.e. (12+4)*2
-(expr,expr(,)) tuple with optional comma
-(comprehension) a for a in __
                   for b in __
                   if a > 10
                   "gives you generator"
```

Square Braces

```
[]  list
    -list comprehension
```

Curly Braces

```
{} - dictionary (dict)
{...} -can be set, set comprehension,
   dict if key:val, or dict comprehension
```

expression list

```
expr,expr,*expr,expr,*expr
   -star takes sequence and includes it in a list,
   set, tupple, ** unpacks a dictionary
```

Yeild expression

```
(yeild expr) or (yeild from expr)

a.b -attribute(attr) looks up b in a

a[expr list] - element access, like key or index  a[1,3]

a[list of slices] slice is expr:expr:expr
   (lower bound : upper bound : optionally the stride)

a(argument list) function column with expr,
   list of expr, key=expr, *expr, **expr

a(comprehension) function column way to feed parameters i

await expr
```

Math operators

- a**b power
- +a, -a, ~a unary operators that does nothing, negates, and bitwise negation
- a*b - if numbers does multiplication
  - if seq, duplicates such as [1]*6 = [1,1,1,1,1,1]
- a@b - matrix multiplication
- a/b - float devision
- a//b - floor devision
- a%b - for numbers is modulo
  - for sequence this is format operation (old-style/printf) “%s” % name

Math operators continued

- a+b - addition
  - concatination [1,2]+[3,4]=[1,2,3,4]
- a-b - subtraction
- a«b
- a»b - binary shift
- a&b - binary bitwise and
- a|b - bitwise or
- aˆb - bitwise xor
- a<b a<=b a==b a!=b a>=b a>b, a is b, a is not b, a in b, a not in b, a if b else c (trinary conditional)
- lambda parameters: expr

Simple Statements

- (don't have sweets)
- pass
- expr list
- assignment . . . =
- assert expr
- assert expr, expr (second expr explains the assertion error)
- del lookup - removes items or elements from sets
- return expr list
- yeild expr
- yeild from expr

Simple Statements Cont.

- raise
- raise exc
- raise exctype
- raise exc from exc
- break
- continie
- import . . . as . . .
- from . . . import
- global name list
- nonlocal namelist

Compound Statemnets

```
Suite of statements attach to a Compound statements
Token generated in a NEWLINE, for each indent on new line,
and each dedent

A)  ~: statements ; statements
B)  ~:
     statement
     statement


if condition: suite
elif cond: suite        (conditional)
else: suite     (conditional)
```

Compound statements cont.

```
while cond: suite
else: suite (executed if you didn't use a break,
aka condition failed)

for (target list) in (iterables);
    suite
else: suite

try: suite
except exectype as e: suite
except exceptype: suite
execpt : suite
else: suite  (if suite is succesfull)
finally: suite (runs reguardless)
```

Compound statements cont.

```
with expr(as e): suite
    -executes expression then looks into it and
    calls the entermethod, then calls exit method
    -good for going into and out of a file
with expr,exper:suite (nested)
```

Function Definition

```
@decorator
def name(parameter): suite
    name
    name=value
    name:type
    name:type=value
    *args
    **kwargs
Corutine
async def name(parameters):suite
async for
async with
async await
x async for
    x in ..

... yeild ...
(anywhere in function turns it into a generator)
```

Class definitions

```
@decorators
class name(...): suite
```