Basics

Indentation combines structures

```
parent statement:
    statement block 1...
        ...
    parent statement:
        statement block 2...
        ...

next statements ...
```

Base Types *integer*, *float*, *boolean*, *string*, *bytes*

```
int    433 0 -134 0b010  0o234 0xA3
                  binary octal hexa
float 3.34 0.0  -1.4e-3
                    x10^-3
bool  True False
str   "one\nTWo"  """A\tB\tc
                  C\t"""
bytes b"todo:\xce\770"
             hexa octal
```

Identifiers for *variables*, *functions*, *modules*, *classes* ... *names*

Definition: `[a-zA-Z][a-zA-Z0-9\]*`

- Diacritics should be avoided
- Language keywords are forbidden
- snake_case **or** CamelCase

Variable assignment or **binding** of a *name* with a *value*

```
x = 3.2 + 7 + cos(z)
a = b = c = 0
x,y = 1,2 # multiple assignments
a,b = b,a # swap values

a,*b = seq # unpacking sequence
*a,b = seq # in item and list

x += 3 # x = x + 3
x -= 4 # x = x - 4
... *= /= %= ...
x = None # undefined
del x    # remove name x
```

Simple Statements

Boolean Logic

```
Comparisons: < > <= >= == !=
a and b (shortcut evaluation)
a or b  (shortcut evaluation)

not a   (logical not)
True, False (constants)
```

Basic Operators

```
Operators: + - * / //  %   **
                      div mod a^b
Membership: op in seq
            op not in seq
Identity: type1 is type2
          type1 is not type2
(1+4.3)*2 -> 10.6
"Hello" + " World"->"Hello World"
abs(-2.3) -> 2.3
round(3.23,1) -> 3.2
pow(3,4) -> 81
```

Bitwise Operators or how to deal with Zeros and Ones

```
Operators:
  << # bitwise shift to the left
  >> # bitwise shift to the right
  &  # bitwise AND
  |  # bitwise OR
  ^  # bitwise XOR
  ~  # bitwise NOT
```

Shifting right by 1 means integer division by 2 without rest

Lambdas are used to create anonymous functions

```
lambda [parameters]: expression

x = lambda a: a + 10 -> x(5) = 15

def func(n):
    return lambda a : a * n

mydoubler = func(2)
mytripler = func(3)
mydoubler(11) -> 22
mytripler(11) -> 33
```

Complex math: is integrated

```
a, b = 2, 3
z = complex(a,b)
z = a + bj
z.real -> realpart
z.imag -> imaginary part
```

for complex functions: **import** cmath

Math package brings more power: **from** math **import** *sin*, *cos*, ...

```
sin(pi/4) -> 0.707
sqrt(81) -> 9.0
log(e**2) -> 2.0
ceil(12.5) -> 13
floor(12.5) -> 12
```

Conversions

```
int("14") -> 14 int("3f",16) -> 63
int(13.23) -> 13
float("-3.23e3") -> -3230.0
round(14.56,1) -> 14.6
bool(x) # False for 0, None,  empty
    # container or False; True for
    # everything else
str(x) -> "..." # __str__()
chr(64) -> '@' ord('@') -> 64
repr(x) -> "..." # literal x
bytes([34,67,12]) -> b'"C\x0c'
list("abs") -> ['a', 'b', 's']
dict([(4,'four'),(2,'two')])
    -> {4:'four',2:'two'}
set(['one','two']) ->{'one','two'}
','.join(['one','two'])->'one,two'
'some spac es'.split()
    -> ['some','spac','es']
[int(x) for x in ('1','10',-2)]
    -> [1,10,-2]
```

Containers

Types *list*, *tuple*, *str*, *dict*, *set*, *frozenset*

```
list  [1,4,5]  ["s",11,4.2]
tuple (1,4,5)   "s",11,4.2
str bytes  # ordered seq of chars
dict  {"key": "value"}
set   {"key1", "key2"}
```

tuple, *str* and **keys** in *dict*, *set* are immutable. *frozenset* is an immutable *set*.

Sequence Container Indexing or how to slice

```
lst=[1, 2, 3, 4, 5]
lst[0]   -> 1  lst[1]  -> 2
lst[-1]  -> 5  lst[-2] -> 4
lst[start:end(exclusive):step]
lst[:-1] -> [1,2,3,4]
lst[::2] -> [1,3,5]
lst[::-2]-> [5,3,1]
lst[:] -> [1,2,3,4,5] shallow copy
```

On mutable sequences remove with `del lst[1]` or slices with `del lst[:3]`

Integer Sequences with **range()** are immutable sequences of type *int*

```
  range([start, ] end [, step])
# defaults 0   exclusive   1
range(5) -> 0 1 2 3 4
range(2,10,3) -> 2 5 8
```

Generic Operations on Containers. For *dict* and *set* these operations use **keys**

```
len(c) -> items count
min(c)  max(c)  sum(c)
sorted(c) -> sorted list copy
enumerate(c) -> iterator on
             (index, value)
zip(c1,c2) -> iterator on tuples
           containing c_i items
all(c) -> True if all c items eva
          luate to True
any(c) -> True if at least one
       item in c evaluates True
```

Specific Operations on *ordered sequence containers*

```
reversed(c) -> inverse iterator
c.index(val) -> position
c*2 -> duplicate
c + c2 -> concatenate
c.count(val) -> events count
```

Copy Containers with module: *import* copy

```
copy.copy(c) -> shallow copy of c
copy.deepcopy(c) -> deep copy
```

List Operations

```
lst.append(val) # add item at end
lst.extend(seq) # add seq of item
lst.insert(idx,val) # add item at
    # index
lst.remove(val) # remove first
    # item with val
lst.pop([idx]) # remove and return
    # item at idx. Default last.
lst.sort() # sort list inplace
lst.revers() # reverse list
    # inplace
```

Dictionary Operations

```
d[key] = value
d[key] -> value
d.update(d2) # update associations
d.keys() -> key iterator
d.values() -> value iterator
d.items() -> key,value iterator
d.pop(key[,default]) -> value
d.popitem() -> (key,value)
d.get(key[,default]) -> value
d.setdefault(key[,default])->value
d.clear() # delete all keys,values
del d[key] # delete key
```

Set Operations see **Dictionary Operations** for explanation of methods

```
Operators:
|   # union
&   # intersection
- ^ # difference/symmetric diff.
< > <= >= # inclusion relations

s.update(s2)    s.copy()
s.add(key)      s.remove(key)
s.discard(key)  s.clear()
s.pop()
```

Compound Statements

The If Statement is used for conditional execution

```
if expression: ...
(elif expression: ...)*
[else: ...]
```

```
if bool(x)==True <-> if x:
if bool(x)==False <-> if not x:
```

The While Statement is used for repeated execution as long as an expression is true

```
while expression: ...
[else: ...]
```

If the expression is true execute the **while** statement and if false execute the **else** statement, if present and the loop terminates. A **break** statement in the **while** loop terminates the loop without executing the **else** clause.

The For Statement is used to iterate over elements of a sequence or other iterable objects.

```
for targets in expression_list: ...
[else: ...]
```

The for statement is executed once for every item(s) provided by the iterator, resulting from the expression_list. Don't remove or add items from/to the list the iterator is from instead create a copy:

```
for x in a[:]:
    if x < 0: a.remove(x)
```

The Try Statement specifies exception handlers and/or cleanup code for a group of statements

```
try: ...
(except [expression [as id]]:...)+
[else:...]
[finally:...]
or just
try:...
finally:...
```

The finally clause, if present specifies a 'cleanup' handler which is always executed.

The With Statement is used to wrap the execution of a block with methods defined by a context manager without the need for a try…except…finally block.

```
with item (, item)*: ...
item is: expression [as target]
```

Function Definitions

```
def func ( [parameters] ):...
default parameters: p [= expr ]
*p  # remaining positional args
**p # remaining dict (kw)args

myclass.func -> func object
r = myclass.func() # call func
```

Class Definitions

```
class name ([inheritance]): ...
    def __init__([parameters]): ...
```

The **__init__()** function initializes the parameters as instance attributes.