**CSPC21 MIDTERMS**

---

## Python Functions

- A function is a block of code which only runs when it is called.
- You can pass data, known as parameters, into a function.
- A function can return data as a result.

## Creating a Function

- In Python a function is defined using the ***def*** keyword.

**Example**

```
def my_function():
  print("Hello form a function")
```

## Calling a Function

- To call a function, use the function name followed by parenthesis:

**Example**

```
def my_function():
  print("Hello form a function")

my_function()
```

## Creating a Function

- Information can be passed into functions as arguments.
- Arguments are specified after the function name, inside the parenthesis. You can add as many arguments as you want, just separate them with a comma.
- The following example has a function with one argument (fname). When the function is called, we pass along a first name, which is used inside the function to print the full name:

**Example**

```
def my_function(fname):
  print(fname + " Refnes")

my_function("Emil")
my_function("Tobias")
my_function("Linus")
```

- Arguments are often shortened to args in Python documentations.

## Parameters or Arguments

- The terms parameter and argument can be used for the same thing: information that are passed into a function.
- From a function's perspective:
  - A parameter is the variable listed inside the parenthesis in the function definition.
  - An arguments is the value that is sent to the function when it is called.

## Number of Arguments

- By default, a function must be called with the correct number of arguments. Meaning that if your function expects 2 arguments, you have to call the function with 2 arguments, not more, and not less.

**Example**

**This function expects 2 arguments, and gets 2 arguments:**

```
def my_function(fname, lname):
  print(fname + " " + lname)

my_function("Emil", "Refsnes")
```

- If you try to call the function with 1 or 3 arguments, you will get an error

**Example**

**This function expects 2 arguments, and but gets only 1:**

```
def my_function(fname, lname):
  print(fname + " " + lname)

my_function("Emil")
```

## Arbitrary Arguments, *args

- If you do not know how many arguments that will be passed into your function, add a * before the parameter name in the function definition.
- This way the function will receive a tuple or arguments, and can access the items accordingly:

**Example**

**If the number of arguments is unknown, add a * before the parameter name:**

```
def my_function(*kids):
  print("The youngest child is " + kids[2])

my_function("Emil", "Tobias", "Linus")
```

## Keyword Arguments

- You can also send arguments with the key=value syntax.
- This way the order of the arguments does not matter.

**Example**

```
def my_function(child3, child2, child1):
  print("The youngest child is " + child3)

my_function(child1="Emil", child2= "Tobias", child3= "Linus")
my_function("Emil", "Refsnes")
```

## Arbitrary Keyword Arguments, **kwargs

- If you do not know how many keyword arguments that will be passed into your function, add two asterisk: ** before the parameter name in the function definition.
- This way the function will receive a dictionary of arguments, and can be access the items accordingly:

**Example**

**If the number of keyword arguments is unknown, add a double ** before the parameter name:**

```
def my_function(**kid):
  print("His last name is " + kid["lname"])

my_function(fname="Tobias", lanme= "Refsnes")
```

### Default Parameter Value

- The following example shows how to use a default parameter value.
- If we call the function without argument, it uses the default value:

**Example**

```
def my_function(country = "Norway"):
  print("I am form " + country)

my_function("Sweden")
my_function("India")
my_function()
my_function("Brazil")
```

### Passing a List as an Argument

- You can send any data types of argument to a function (string, number, list, dictionary, etc.), and it will be treated as the same data type inside the function.
- E.g. if you send a list as an argument, it will still be a List when it reaches the function:

**Example**

```
def my_function(food):
  for x in food:
    print(x)

fruits = ["apple", "banana", "cherry"]

my_function(fruits)
```

### Return Values

- To let a function return a value, use the return statement.

**Example**

```
def my_function(x):
  return 5 * x

print(my_function(3))
print(my_function(5))
print(my_function(9))
```

### Pass Statemet

***function*** definitions cannot be empty, but if you for some reason have a ***function*** definition with no content, put in the ***pass*** statement to avoid getting an error.

**Example**

```
def myfunction():
  pass
```

- The next Python code block contains an example of a function without a return statement. We use the pass statement inside of this function. pass is a null operation. This means that when it is executed, nothing happens. It is useful as a placeholder in situations when a statement is required syntactically, but no code needs to be executed:

### Python Dictionaries

- Dictionaries are used to store data values in key: value pairs.
- A dictionary is a collection that is ordered*, changeable, and does not allow duplicates.

### Dictionary Items

- Dictionary items are ordered, changeable, and does not allow duplicates.
- Dictionary items are presented in key: value pairs, and can be referred to by using the key name.

### Creating Python Dictionaries

- Creating a dictionary is as simple as placing items inside curly braces {} separated by commas.
- An item has a key and a corresponding value that is expressed as a pair (key: value).
- While the values can be of any data type and can repeat, keys must be of immutable type (string, number, or tuple with immutable elements) and must be unique.

```
#empty dictionary
my_dict = {}

#dictionary with integer keys
my_dict = {1: 'apple', 2: 'ball'}

#dictionary with mixed keys
my_dict = {'name': 'John', 1: [2, 4, 3]}

#using dict()
my_dict = dict({1: 'apple', 2: 'bal'})

#from sequence having each item as a pair
my_dict = dict([(1, 'apple'), (2, 'ball')])
```

- As you can see from above, we can also create a dictionary using the built-in dict() function.

### Duplicates Not Allowed

- Dictionaries cannot have two items with the same key:
- Duplicate values will overwrite existing values:

```
thisdict = {
  "brand": "Ford",
  "model": "Mustang",
  "year": 1964,
  "year": 2020
}
print(thisdict)
```

### Dictionary Length

- To determine how many items a dictionary has, use the len() function:

- Print the number of items in the dictionary:

```
print (len(thisdict))
```

**Accessing Elements from Dictionary**

- While indexing is used with other data types to access values, a dictionary uses keys. Keys can be used either inside square brackets [] or with the get() method.
- If we use the square brackets [], KeyError is raised in case a key is not found in the dictionary. On the other hand, the get() method returns None if the key is not found.

```
#get vs [] for retrieving elements
my_dict = {'name': 'Jack', 'age': 26}

 #Output: Jack
print(my_dict['name'])

#Output: 26
print(my_dict.get('age'))

#Trying to access keys which doesn't exist throws error
#Output None
print(my_dict.get('address'))

#KeyError
print(my_dict['address'])
```

**Output**

```
Jack
26
None
Traceback (most recent call last):
   File "<string>", line 15, in <module>
      print(my_dict['address'])
KeyError: 'address'
```

**Changing and Adding Dictionary Elements**

- Dictionaries are mutable. We can add new items or change the value of existing items using an assignment operator.
- If the key is already present, then the existing value gets updated. In case the key is not present, a new key (key: value) pair is added to the dictionary.

```
#geChanging and adding Dictionary Elements
my_dict = {'name': 'Jack', 'age': 26}

 #update value
my_dict['age'] = 27

#Output: {'age': 27, 'name': 'Jack'}
print(my_dict.get)

#add item
my_dict['address'] = 'Downtown'

# Output: {'address': 'Downtown', 'age': 27, 'name': 'Jack'}
print(my_dict)
```

**Output**

```
{'age': 27, 'name': 'Jack'}
{'address': 'Downtown', 'age': 27, 'name': 'Jack'}
```

**Removing Elements from Dictionary**

- We can remove a particular item in a dictionary by using the pop() method. This method removes an item with the provided key and returns the value.
- The popitem() method can be used to remove and return an arbitrary (key, value) item pair from the dictionary. All the items can be removed at once, using the clear() method.
- We can also use the del keyword to remove individual items or the entire dictionary itself.

```
#Removing elements from a dictionary

#create a dictionary
squares = {1: 1, 2: 4, 3: 9, 4: 16, 5: 25}

#remove a particular item, return its value
 #Output: 16
print(squares.pop(4))

#Output: {1: 1, 2: 4, 3: 9, 5: 25}
print(squares)

#remove an arbitrary item, return (key: value)
#Output: (5, 25)
print(squares.popitem())

#Output: {1: 1, 2: 4, 3: 9}
print(squares)

#remove all items
squares.clear()

#Output: {}
print(squares)

#delete the dictionary itself
del squares

#Throws Error
print(squares)
```

**Output**

```
16
{1: 1, 2: 4, 3: 9, 5: 25}
 (5, 25)
{1: 1, 2: 4, 3: 9}
{}
Traceback (most recent call last):
   File "<string>", line 30, in <module>
      print(squares)
NameError: name 'squares' is not defined
```

**Good luck, CS 1-2!**