Python (1. Einleitung) **Prüfungsstoff**

1. Du kennst den Namen des Informatikers, der die Programmiersprache „Python“ entwickelt hat
2. Du kannst beschreiben, worauf sich der Name „Python“ bezieht.
3. Du kannst angeben, welche Art von Programmen nötig sind, um ein Python-Programm zu schreiben und um ein Python Programm auszuführen.
4. Du kennst das Zeichen, das Python für Kommentare verwendet.
5. Du kannst die drei Fehlerkategorien beim Programmieren nennen und jeweils ein Beispiel angeben.
6. Du kennst die Syntax („Aufbau“) und die Semantik („Bedeutung“) der folgenden, in Python eingebauten Funktionen `bin(...)`, `oct(...)`, `hex(...)`, `int(...)` und kannst entsprechende Ausdrücke auf Papier evaluieren (auswerten).

Python (2. Arithmetik) **Prüfungsstoff**

1. Du kannst Arithmetische Ausdrücke mit folgenden Operatoren auswerten:

   - Grundrechenarten (`+`, `-`, `*`, `/`)
   - Modulo-Operator (`%`)
   - Ganzzahl-Division (`//`)
   - Potenzen und Wurzeln (`**`)

2. Du kennst die Operator-Hierarchie (Klammern, Potenzen, „Punkt vor Strich“)

3. Du weisst, dass ohne Klammern die Operationen gleicher Stufe von links nach rechts ausgewertet werden. **Ausnahme:** Ohne Klammern werden beim Potenzieren die Operanden von rechts nach links ausgewertet. (`2**3**2` ist 2^9 und nicht 8^2.)

4. Du kennst die folgenden Datentypen

   - ganze Zahlen (`int`)
   - Fliesskommazahlen (`float`)

   und kannst angeben, welchen Datentyp ein arithmetischer Ausdruck hat, wenn der Datentyp der Operanden bekannt ist.

Python (3. Variablen) **Prüfungsstoff**

1. Du kannst gültige (und damit auch ungültige) Variablennamen erkennen. Als Python-Schlüsselwörter müssen nur diejenigen erkannt werden, die bisher im Unterricht behandelt wurden.

2. Du kannst Zeichenketten von Variablen unterscheiden.

3. Du kennst den Zuweisungsoperator „=“ und weisst, wie Zuweisungen ausgewertet werden.

4. Du kannst die erweiterten Zuweisungen (`+=`, `-=`, `*=`, `/=`, `//=`, `%=`, `**=`) interpretieren.

5. Du kannst Mehrfachzuweisungen interpretieren.

6. Du kannst Daten von der Shell (Konsole) mit der Funktion `input("...")` einlesen und weisst, dass als Ergebnis jeweils ein String (Zeichenkette) zurückgegeben wird.

7. Du kannst eine Zeichenkette, die eine Zahl darstellt, mit der Funktion `int(...)` bzw. `float(...)` in den entsprechenden Datentyp umwandeln.

8. Du verstehst die Semantik (Bedeutung) der `print(...)`-Anweisung. und weisst, dass auch mehrere, durch Kommas getrennte Argumente, nacheinander ausgegeben werden können.

9. Du kannst kannst Programme schreiben, die ...

   - Daten von der Shell einlesen,
   - Zeichenketten in den richtigen Datentyp umwandeln (siehe oben),
   - Berechnungen ausführen,
   - Ergebnisse ausgeben.

1. Du kannst die beiden Konstanten `True` und `False` des Python-Datentyps `Boolean` (`bool`) aufzählen und in Programmen richtig schreiben.

2. Du kannst Ausdrücke mit den Vergleichsoperatoren `<`, `<=`, `>`, `>=`, `==` und `!=` auswerten.

3. Du kannst Ausdrücke mit den logischen Operatoren `not()`, `or`, `and` auswerten.

4. Du kennst die Hierarchie bei der Auswertung von logischen Ausdrücken, von Vergleichsausdrücken und von gemischten Ausdrücken mit arithmetischen Operatoren in Python:

    1. `(...)`
    2. `**`
    3. `*`, `/`, `//`, `%`
    4. `+`, `-`
    5. `!=`, `==`, `<`, `<=`, `>`, `>=`
    6. `not(...)`
    7. `and`
    8. `or`

5. Du kannst Ausdrücke auswerten, in denen die Warheitswerte in entsprechende ganze Zahlen umgewandelt (*gecastet*) werden. *Beispiele:*

Ausdruck	Wert
`int(True)`	`1`
`int(False)`	`0`
`True + 2`	`3`
`10 * False`	`0`
`(5 < 7) + (3 != 4)`	`2`

Python (5. Verzweigungen) **Prüfungsstoff**

1. *Voraussetzung:* Wahrheitswerte
2. Syntax und Semantik der bedingten Anweisung (`if`)
3. Syntax und Semantik einer einfachen Verzweigung (`if ... else`)
4. Syntax und Semantik einer mehrfachen Verzweigung (`if ... elif ... else`)
5. Du kannst mit den oben genannten Anweisungen Programme von angemessenem Schwierigkeitsgrad schreiben, in denen bedingte Anweisungen oder Verzweigungen realisiert werden müssen. Dazu gehört, dass du mit den Vergleichoperatoren und den logischen Operatoren umgehen kannst.

1. Du kannst `for`-Schleifen mit einer Laufvariablen (`i`, `j`, `k`, ...) und den folgenden Varianten der `range(...)`-Funktion interpretieren:

   - `range(n)`
   - `range(m,n)`
   - `range(m,n,d)`
   - `range(n,m,-d)`

   wobei $m \leq n$ und $d > 0$.

2. Du kannst `for` mit einer Elementvariablen und einer Liste (`[...]`) interpretieren.

3. Du kannst `while`-Schleifen interpretieren und

4. Du kannst Endlosschleifen erkennen.

5. Du kannst Schleifen mit `break` abbrechen.

6. Du kannst die Abarbeitung eines Schleifenkörpers mit `continue` überspringen.

7. Du kannst verschachtelte `for`- und `while`-Schleifen interpretieren.

Python (7. Listen und Tupel) **Prüfungsstoff**

1. Einfache und verschachtelte Listen definieren
2. Zugriff auf Listenelemente (auch bei verschachtelten Listen)
3. Zugriff auf Listenelemente mit negativen Indizes
4. Die Länge einer Liste mit der Funktion `len()` bestimmen
5. Einzelne Elemente einer Liste verändern
6. Elemente am Ende einer Liste hinzufügen
7. Ein Element vom Ende der Liste löschen
8. Ein Element von der i-ten Position löschen
9. Listen verketten
10. Listen vervielfachen
11. Zugriff auf Teillisten („slices“)
12. Kopien von Listen (als Referenz oder elementweise)
13. Reihenfolge der Elemente einer Liste umkehren
14. Elemente einer Liste aufsteigend oder absteigend sortieren
15. Elemente einer Liste (mit Zahlen) addieren
16. Mehrfachzuweisungen
17. Listen elementweise durchlaufen.
18. Listen indexgesteuert durchlaufen.
19. List Comprehensions (Listen mit for-Schleifen „von innen“ erzeugen) Listen
20. Definition von Tupeln
21. Zugriff auf Tupel-Elemente:
22. Operationen und Funktionen für Tupel: *Bei Tupeln sind alle Listenoperationen erlaubt, die das Original-Tupel nicht verändern.*

1. Vorteile des Funktionskonzepts:
   - Wiederverwendbarkeit von Programmcode
   - Zentrale Wartung und Verbesserung von Programmcode
   - Durch Funktionen werden Programmdetails verborgen und somit Programme übersichtlicher bzw. besser lesbar
2. Syntax einer Funktionsdefinition
3. Formale und aktuelle Parameter
4. Rückgabe von Werten mit `return`
5. Schreiben einfacher Funktionen mit einem Rückgabewert
6. Benannte Parameter und Parameterreihenfolge
7. Auswertung von Ausdrücken mit Funktionen
8. Gültigkeitsbereich (*scope*) lokaler Variablen
9. Default-Werte für Parameter
10. Listen als Funktionsparameter
11. Einfache Rekursionen (Funktionen, die sich selber aufrufen)

1. Darstellung von Zeichenketten mit (dreifachen) Hochkommas oder Anführungszeichen
2. Escape-Sequenzen:
   - Zeilenschaltung (*Newline*) (`\n`)
   - Anführungszeichen `\"`
   - Apostroph (`\'`)
   - Backslash (`\\`)
3. Zugriff auf einzelne Zeichen und Teilstrings mittels Listen-Syntax
   - `s[0]`, `s[-1]`, ...
   - `s[1:5]`, `s[5:]`, `s[:4]`, ...
   - `s[1:5:2]`, `s[::-1]`, ...
4. *Merke:* Python-Strings sind *immutable* (unveränderlich): es können keine einzelnen Zeichen(bereiche) ersetzt, eingefügt oder gelöscht werden.
5. Operatoren für Zeichenketten: `+` und `*`
6. Funktionen für Zeichenketten:
   - `len(<string>)`
   - `list(<string>)`
   - `ord(<character>)`
   - `chr(<int>)`
   - `str(<objekt>)`
7. String-Methoden:
   - `str.lower()`
   - `str.upper()`
   - `str.capitalize()`
   - `str.replace(<str1>, <str2>)`
   - `str.join(<str-liste>)`
   - `str.split(<separator>)`
   - `str.strip(`*`characters`*`)`
   - `str.lstrip(`*`characters`*`)`
   - `str.rstrip(`*`characters`*`)`
   - `str.zfill()`
   - `str.count(`*`value`*`)`
8. Der Gebrauch der `str.format()`-Methode mit Platzhaltern (ohne spezielle Formatierung).

1. Du kannst die Spezifikation zur Format-Methode für die folgenden Ausdrücke interpretieren:

   - `fill = <any character>`
   - `align = "<" | ">" | "^"`
   - `sign = "+" | " "`
   - `width = digit+`
   - `grouping_option = "_" | ","`
   - `precision = digit+`
   - `type = "b" | "d" | "e" | "f" | "o" | "x" | "X"`

2. Du kannst die `print()`-Funktion mit den Keyword-Argumenten `sep` und `end` interpretieren.

3. Du kannst Dateien zum Lesen oder zum Schreiben öffnen und kennst die Methoden `write()`, `read()` und `close()` für Dateiobjekte.

   Darüber hinaus kannst du den Inhalt von Dateien zeilenweise mit einer `for`-Schleife auslesen.

4. Du kannst Zeichenketten, die Zahlen darstellen, mit `int()` bzw. `float()` in den entsprechenden Datentyp umwandeln.

5. Du kannst einfache Programme zum Schreiben und Lesen von Dateien schreiben.

1. Du weisst, dass Python bei Dictionaries die Elemente aus Effizienzgründen in einer scheinbar zufälligen Reihenfolge speichert. (→ Hashtabellen)
2. Du kannst ein Dictionary mit Werten initialisieren:

   `D={<key1>: <value1>, <key2>: <value2>, ...}`
3. Du kannst ein leeres Dictionary mit `D=dict()` anlegen.
4. Du kannst Schlüssel-Wert-Paare mit `D[<key>]=<value>` zu einem Dictionary hinzufügen.
5. Du kannst mit dem Schlüssel `D[<key>]` oder mit `D.get(<key>)` auf den zugehörigen Wert im Dictionary zugreifen.
6. Du kannst den zu einem Schlüssel gehörenden Wert mit `D[<key>]=<newvalue>` ändern.
7. Du kannst mit `len(D)` die Anzahl der Elemente eines Dictionaries bestimmen.
8. Du kannst mit `if <key> in D:` oder `if <key> not in D:` testen, ob ein Schlüssel in einem Dictionary (nicht) vorhanden ist.
9. Du kannst ein Schlüssel-Wert-Paar mit `D.pop(<key>)` oder `del D[<key>]` aus einem Dictionary entfernen.
10. Du kannst mit `D.update(E)` ein Dictionary `D` mit einem Dictioary `E` aktualisieren.
11. Du kannst ein Dictionary mit `for <var> in D:` oder `for <var> in D.keys():` schlüsselweise durchlaufen.
12. Du kannst ein Dictionary mit `for <var> in sortetd(D.keys()):` nach sortierten Schlüsseln durchlaufen.
13. Du kannst ein Dictionary mit `for <var> in D.values():` werteweise durchlaufen.
14. Du kannst ein Dictionary mit `for (<keyvar>,<valuevar>) in D.items():` paarweise durchlaufen.
15. Du kannst mit `D.copy()` eine vom Original unabhängige Kopie des Dictionaries `D` herstellen und weisst, dass die Zuweisung `E = D` nur eine Referenz auf `D` in `E` speichert.
16. Du kannst Python-Programme schreiben, die einfache Zählaufgaben (Zeichenketten, Listenelemente) mittels eines Dictionaries erledigen.

1. Du kannst aus dem Umstand, dass die Elemente von Mengen einfach und ungeordnet auftreten, die nötigen Konsequenzen für die Darstellung von Mengen in Python ableiten.

2. Du bist in der Lage, Mengen mit

   - {<e1>, <e2>, <e3>, ...}
   - set(<list>) set([3, 2, 1, 3, 3]) => {1, 2, 3}
   - set(<string>) set('HELLO') => {'H', 'L', 'E', 'O'}

   aus einzelnen Elementenen, aus Listen oder Zeichenketten zu erzeugen und kannst die resultierende Datenstruktur (bis auf Reihenfolge) beschreiben. Die scheinbar willkürliche Reihenfolge bei der Ausgabe von Mengen ist irrelevant und muss nicht vorhergesehen werden.

3. Du kannst die leere Menge mit `set()` erzeugen.

4. Du bist in der Lage, die folgenden Python-Ausdrücke für geeignete Mengen `A`, `B`, und Elemente `e` auswerten.

   - `A.union(B)` $(A \cup B)$
   - `A.intersection(B)` $(A \cap B)$
   - `A.difference(B)` $(A \setminus B)$
   - `A.add(element)` $(A \cup \{e\})$
   - `A.discard(element)` $(A \setminus \{e\})$
   - `A.issubset(B)` (gilt $A \subset B$?)
   - `A.isdisjoint(B)` (gilt $A \cap B = \{\}$?)

5. Du kannst Python-Ausdrücke der Form `len(M)`, `sum(M)`, `sorted(M)` für geegnete Mengen `M` interpretieren.

6. Du kannst den Operator `in` im Zusammenhang mit Mengen interpretieren:

   `5 in {1,2,3} => False`

   `5 in {3,4,5} => True`

7. Du kannst mit einer `for`-Schleife über Mengen iterieren.

1. Du kannst erkennen, ob die Ausdrücke zum Importieren von Modulen syntaktisch korrekt sind.

2. Du erkennst, ob die Schlüsselwörter `'import'`, `'from'`, `'as'` und `'*'` syntaktisch korrekt angeordnet sind. Konkret:

   - `'import' <module> ['as' <var>]`
   - `'from' <module> 'import' <var> ['as' <var>] (,<var> ['as' <var])*`
   - `'from' <module> 'import' '*'`

   Wobei `<module>` für einen Modulnamen (ohne die Endung `.py`), `<var>` für einen gültigen Bezeichner (Variablennamen) stehen. Die Metasymbole `[...]` stehen für optionale Syntaxelemente, die runden Klammern `(...)` für eine Zusammenfassung von Elementen und der Kleene-Stern (`*`) für 0, 1, 2, ... -fache Wiederholung des vorangehenden Elements. Man beachte, dass `'*'` ein Python-Schlüsselwort ist während `*` dazu dient, die Syntax zu beschreiben.

   Bemerkung: Eine formale Beschreibung der Syntax von Programmiersprachen erfolgt oft durch reguläre Ausdrücke.

3. Du kannst das Konzept der *Namensräume* auf Python-Programmen anwenden, indem du erkennst, welche Variablen, Funktionen und Objekte sich in unterschiedlichen Namensräumen befinden und welche im gleichen Namensraum sind.

1. Du kannst die folgenden Begriffe erklären:

   - Klasse
   - Instanz (Objekt)
   - Instanzvariable, Klassenvariable
   - Instanzmethode, Klassenmethode
   - Konstruktor
   - Vererbung
   - Kapselung
   - Überschreiben von Methoden

2. Du kannst einfache OOP-Programme in der Programmiersprache Python interpretieren. Um dies zu tun, kennst die Bedeutung der folgenden Schlüsselwörter:

   - `class <Name>`
   - `class <Name>(<Elternklasse>)`
   - `__init__(...)`
   - `super()`

   Die Namen der Spezialmethoden wie `__str__()` oder `__add__()` werden, falls nötig, zur Verfügung gestellt.

3. Du weisst, dass bei der Definition von Instanzmethoden als erstes Argument jeweils ein formaler Objektname (üblich ist `self`) angegeben werden muss.

4. Du kannst eine einfache Klasse aufgrund einer vorgegebenen API (Programmierschnittstelle) implementieren.

5. Du weisst wie Variablennamen innerhalb der Vererbungshierarchie aufgelöst werden.

6. Du weisst, dass jede Kindklasse automatisch die Variablen und Methoden ihrer Elternklasse erbt und dass diese Variablen und Methoden auch überschrieben oder durch neue Variablen und Methoden ergänzt werden können.

7. Du kannst die wesentlichen (im Unterricht behandelten) Vorzüge von OOP aufzählen.